Haskell-Coloured Petri Nets

Claus Reinke

Languages and Programming Group,
School of Computer Science and Information Technology,
University of Nottingham, Jubilee Campus, Nottingham NG8 1BB, UK
czr@cs.nott.ac.uk

Abstract. Coloured Petri Nets are a high-level form of Petri Nets, in which transition inscriptions in some programming language operate on individual tokens, i.e., tokens attributed with values of the inscription language. We introduce the variant of Haskell-Coloured Petri Nets (HCPNs) and show that they have a simple mapping to Haskell. HCPNs can thus be used for system modelling in preparation of system implementation in Haskell, following a process of stepwise refinement in which all intermediate specifications are executable Haskell programs. Similar mappings can be used to introduce functional Petri Nets as graphical specification languages on top of other functional languages.

1 Introduction

Petri Nets [9, 10, 1] were introduced in the 1960s as an extension of automata to model distributed systems with concurrent activities and internal communication. Due to their exclusive focus on monolithic, global system states and global state transformations, automata are of limited use for such systems. Petri Nets overcome these limitations by offering means to refine global system states into collections of local states and to express local state transformations. Derived from automata, Petri Nets inherit the advantages of a graphical formalism: specifications can be understood intuitively but are backed up by a formal theory.

Coloured Petri Nets (CPN) [5, 6] are a high-level form of Petri Nets, in which graphical specifications of communication structure (Nets) are combined with textual specifications (inscriptions), in some programming language, of data objects and their manipulations.

Petri Nets are widely used to model distributed and concurrent systems. Application areas include simulation, planning and performance evaluation of manufacturing systems, industrial processes, transport systems and network protocols as well as workflow modelling (selection of topics from "Petri Net Research Groups" [1]). Numerous tools for the simulation or analysis of the different types of Petri Nets have become available, both commercial and free (see "Tools on the Web" [1]).

Similar to functional languages, Petri Nets represent a shift in emphasis from global towards local operations and from control-flow towards data-flow. Programming in functional languages and modelling with high-level Petri Nets thus requires a similar mind-set, and yet each formalism has its own specialities (for instance, graphical structuring of system behaviour versus textual denotation of values). It seems only natural for these two specification formalisms to complement each other.

Indeed, one of the oldest and most popular toolsets for Coloured Petri Nets, Design/CPN [3], is strongly associated with a functional programming language – ML. Design/CPN is currently developed and supported by Kurt Jensen's CPN group at the University of Aarhus in Denmark. A variant of Standard ML was chosen both as a major implementation language and as the inscription language and "the choice of Standard ML has never been regretted" [3].

In spite of this positive experience in the Petri Net community, Petri Nets with functional inscription languages have not yet become a standard tool in the functional programming community. One reason might be a limited awareness of high-level Petri Nets and their practical applications (see "Industrial Applications" [3] or the programmes and proceedings of the regular Petri Net events [1] for samples).

A more serious obstacle to experimentation with functional Petri Nets is the limited availability of implementations. Traditionally, the construction of a complete toolset for any type of Coloured Petri Nets has involved startup costs in the range of 3-4 years. In this paper, we show that the startup costs can be reduced drastically (down to 1-2 weeks) by embedding functional CPNs into their functional inscription languages. CPNs are thus readily available to functional programmers, who can use their favourite functional languages as the inscription languages.

In the next sections, we provide an overview of existing work in functional CPNs, summarize relevant Petri Net terminology and introduce the variant of Haskell-Coloured Petri Nets (HCPNs). HCPNs are shown to have a simple mapping to Haskell. We emphasize simplicity of the mapping over other aspects, so that most of the mapping should carry over with small modifications to other functional languages, as long as inscription and implementation language coincide.

2 Related Work

Several variants of high-level Petri Nets with functional inscription languages have been proposed and implemented. Among these, Design/CPN

[3], using a variant of Standard ML as inscription language, seems to be the most successful. Based on licensing information, the developers claim a user community of "500 different organisations in 40 different countries, including 100 commercial enterprises". The year 1999 marks the 10th anniversary of the Design/CPN toolset and sees the second workshop dedicated to the practical use of Coloured Petri Nets and Design/CPN.

Design/CPN is freely available as an integrated toolset, supporting graphical editing, simulation, and analysis, targeting system modellers using Petri Nets, for whom the functional inscription language is only one part of a toolset and who are usually not interested in the internals of code generation and simulation. From the perspective of functional programmers using Petri Nets, especially if they are interested in experiments with the combination of Petri Nets and functional languages, this approach has severe drawbacks:

- The availability of the tool on different platforms is limited and depends entirely on the work of a single group.
- The implementation of the inscription language is built into the system – users cannot choose among the implementations of SML'97 that are available for a variety of platforms.
- The inscription language is integrated into the system – users cannot use their favourite functional language, unless it happens to be the particular variant of ML that comes as part of the system.
- The potential for system modifications and extensions is limited – the source code for the tool set is not available, and while a programming interface to many of its inner workings is provided, no standalone functional source code is generated from Net models.

We are aware of two other implementations of Petri Nets with functional inscription languages: GRAPH ([11], 1988-1992), a graphical specification language for hierarchical process-systems with process descriptions in the functional language KiR [8], was supported by a rather complete implementation, including a graphical editor, interactive simulation, distributed execution on a local network, and support for analysis. Unfortunately, core graphical components were built on a now obsolete vendor-specific library, and the system is no longer available.

K2 ([2], 1995-1999) is a coordination language based on Petri Nets. Its development has focussed on an efficient distributed implementation of the coordination layer, although a graphical editor is also available. The interface between coordination layer (Net structure) and computation layer (inscription language) is C-based, so that any language with an

interface to C could be used as an inscription language. A connection to a functional inscription language was planned.

We make two general observations: firstly, that the development of a first complete implementation of Coloured Petri Nets seems to take about 3-4 years (later revision cycles seem to be shorter – version four of Design/CPN has just been announced). Secondly, the core application domains of CPNs seem to be modelling and simulation-based analysis of distributed systems, but most applications do not require a distributed implementation of the simulation (witness the success of Design/CPN).

3 From Finite Automata to Coloured Petri Nets

This section gives a brief summary of terminology in the hierarchy of Petri Nets [10, 5, 6, 1]. For a gentle introduction, we start with two simple Net types whose structural elements can also be found as parts of high-level Nets, so that the explanation of high-level Nets merely needs to describe the integration of an inscription language into the Net structure. For more detailed introductions and surveys, the reader is referred to the extensive literature (unfortunately, access to the Petri Net bibliography with more than 6000 entries at [1] is restricted, but the answers to frequently asked questions at the same site list introductory material and surveys).

At the lower end, *condition/event Nets* can be seen as a distributed and non-sequential variant of finite automata: instead of a finite set of global system states, there is a finite set of local *conditions*. Graphically, each condition is represented as a circle, marked with a *token* if the condition is true, and empty otherwise. The state of the system as a whole is composed from the states of all local conditions, as given by the token distribution or *marking*. The global state changes of automata are replaced by local *events*, which depend on and change local conditions. Graphically, each event is represented as a rectangle, with incoming arcs from *pre-conditions* and outgoing arcs to *post-conditions*. An event is *enabled* when all its pre-conditions are true and all its post-conditions are false. The *occurrence* of an event validates its post-conditions and invalidates its pre-conditions.

From here, the generalization towards high-level Nets is straightforward: the graphical representation in terms of circles, tokens, boxes, and arcs remains, but the interpretation and use of these elements evolve. As an intermediate step, *place/transition Nets* shift the focus from propositional conditions to the availability of resources: each circle represents a *place* where resources may be located, the number of resources is indicated

by the number of anonymous tokens on the place. Boxes represent *transitions*, which consume resources from *input places* and produce resources on *output places*. A transition is enabled when tokens are available on all its input places. The *firing* of a transition consumes one token from each input place and produces one token on each output place.

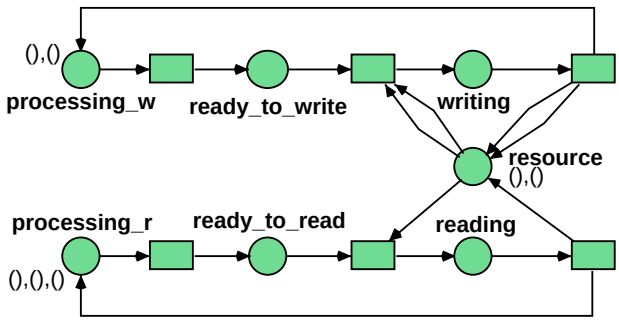

Fig. 1. A reader/writer-problem as a place/transition Net

For illustration, consider the reader/writer-problem modelled in Fig. 1. There are two writers and three readers, occasionally needing access to a shared `resource`. Mutually exclusive, one writer or up to two readers can access the `resource` concurrently. These are modelled by two identical process life cycles for readers and writers, each instantiated several times by having several tokens on each cycle (using `()` for tokens). The two tokens on `resource` represent access permissions, all of which are needed for a writer to proceed. Note how naturally and concisely the Net structure models sequences of events, concurrent events, and conflicts over shared resources (for comparison, a finite automaton modelling the same system would suffer from an exponential increase in the number of states while obscuring the modular structure of the system).

Finally, in high-level Nets, such as Coloured Petri Nets, anonymous tokens and resources are replaced by individual (or coloured) tokens, annotated with typed values of some *inscription language*. Places are annotated with the type (colour set) of tokens they can carry, and because multiple tokens of the same value may be available, places are usually organized as *multi-sets*. Transitions and arcs are annotated with program code that operates on the token values. Input arcs are annotated with variables or patterns which scope over code attributed to the transition and to output arcs, output arcs carry expressions that define the values of output tokens in term of the values of input tokens.

Transition code, also parameterized by the values of input tokens, can consist of additional variable bindings, which scope over guards and output expressions, and of boolean-valued *guards* that determine whether a given multi-set of tokens is appropriate for the firing of the transition (example Nets are given in Fig. 3 and 4). A transition is enabled when tokens are available on each input place for which the transition guard evaluates to true. An enabled transition can fire by consuming the tokens from its input places and producing tokens on its output places.

4 Haskell-Coloured Petri Nets in Haskell

Our aim is to provide a simple implementation of CPNs with functional inscription languages in functional implementation languages.

Our approach is to separate Net structure and Net inscriptions and to embed both in the implementation language (see Fig. 2). To simplify the embedding and to make F-Coloured Petri Nets a true extension of the chosen functional language, the same language F is used for inscription and implementation.

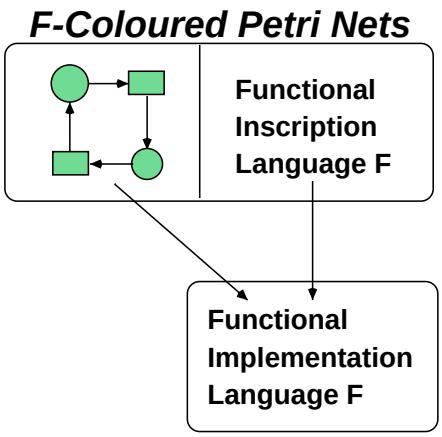

Fig. 2. Embedding functional Petri Nets in their inscription language

The idea is to reuse the implementation of the implementation language for the implementation of the inscription language, so that only a simulation of the Net structure and means to connect the Net simulation and the execution of inscription code need to be provided.

For this paper, we choose Haskell as our inscription and implementation language, leading to Haskell-Coloured Petri Nets (HCPN), but the ideas should carry over with little change to other functional languages.

4.1 An Example of HCPNs– Reviewing

We introduce Haskell-Coloured Petri Nets by means of examples. The operational semantics of the Net structure is given by the firing rule of CPNs, and the semantics of the inscription language is given by the language definition [7]. What needs to be specified is the combination of

Nets and inscriptions, i.e., which elements of the inscription language are allowed to appear where in Nets and how their evaluation is integrated into the firing rule. As we strive for a simple embedding, we have to ensure that all uses of the inscription language in Nets have a straightforward mapping to the implementation language.

To illustrate the use of HCPNs for the modelling of concurrent activity, we use a simplified example of reviewing. Fig. 3 shows what might be a small part of a workflow model for the organization of a workshop.

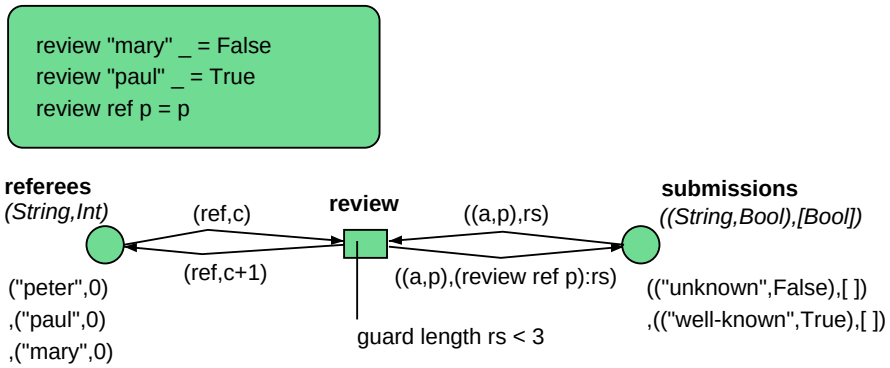

Fig. 3. The process of reviewing as a HCPN

Given a collection of referees and a collection of submitted papers, referees select papers for reviewing until each submission is accompanied by three reviews. Referees are represented by name and number of papers reviewed, submissions by author name, a boolean value indicating acceptability, and a list of referee recommendations. Reviewing of different papers by different referees can proceed concurrently, as the single transition may be enabled for different multisets of tokens. In the model shown in Fig. 3, concurrency is limited by the minimum of the number of submissions and referees available. In the given initial marking, this corresponds to two independent instances of the transition in each step.

Reviewing a paper takes some time, and we would expect each referee to work on one of the papers concurrently – three concurrent activities instead of two! Note that in the standard variants of Petri Nets, there is no concept of global time, only a concept of local *causal relationships*, but we can still make the intermediate state of referees reading a paper explicit as part of the Net structure, as shown in Fig. 4. Reviewing is split into separate start and end transitions and these administrative actions

on the pool of submissions have to be synchronized, but the reading of papers can proceed concurrently.

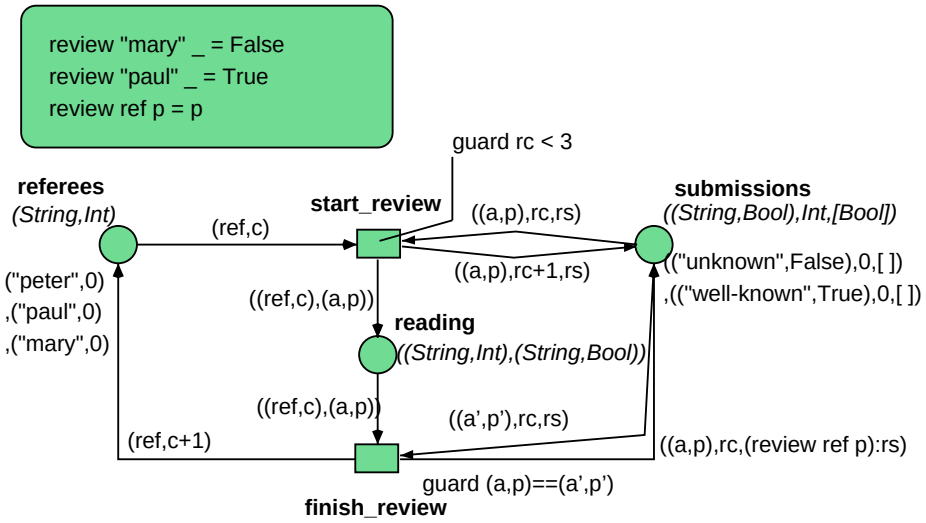

Fig. 4. An alternative HCPN model of reviewing

If we take the non-strictness of function application in Haskell into account, some of this is already captured in our first model: when a referee starts reviewing a submission, there is no need to finish this review before returning the token representing the submission. The result of the review is only "promised", and a place for it is reserved in the list of reviews belonging to that paper. However, the second model makes this explicit in the Net and captures that most referees can only read one paper at a time. In practice, it might well be necessary to lift the result of `review` to a `Maybe`-type to model that not all promised reviews are just delivered.

Note that we use a guard to compare submissions in `finish_review` instead of simply using the same variable names on both input arcs. While more natural, this kind of non-linear pattern is not easily supported by our embedding, and we chose simplicity over a nicer design here.

4.2 The Embedding

The embedding consists of rules that describe how elements of HCPNs are mapped to Haskell code that simulates the Nets and of general support code used in the mapping. This section gives a rough overview of the

mapping. The following sections describe the general support code and the Haskell code generated for the example HCPN in Fig. 4, filling in the details of the mapping.

Apart from information about the graphical representation, HCPNs consist of the following elements, all of which must be mapped to Haskell code: optional global declarations; places with names, optional types, and optional initial markings; transitions with names, optional guards, and arcs from input places and to output places, carrying optional inscriptions. Place types and arc inscriptions default to (), so that place/transition Nets as the one in Fig. 1 can be specified conveniently.

Any global declarations are already Haskell code. Places are mapped to labelled fields in a Net-specific record-like data type, using place names as labels. For simplicity, multisets of tokens on places are represented as lists, so the type of a field in the record structure is the type of lists of the corresponding place type. Net markings are then just the elements of this record-like type, listing the values of tokens on the places in the corresponding fields (for an example, see *mark* and *Mark* in Fig. 6).

Transitions cannot be mapped directly to functions from the types of input tokens to the types of output tokens. There are two problems with this idea: first, in a given marking, a single transition might be enabled for several combinations of input tokens or not at all, and second, this mapping would give transition functions of different types, making it difficult to administrate the simulation of the overall Net structure.

A simple solution to the first problem is to lift the result type of transition functions to lists of results to capture the non-determinism of transition enabling. Our current solution to the second problem is to model each transition as a function from complete Net markings to complete Net markings, so that all transition functions for a given Net have the same type.

4.3 General Support Code

The *SimpleHCPN* module listed in Fig. 5 provides support code for single-step simulation with a textual trace. As Net markings will be given in the form of records with a field for each place in the Net, the type of markings will differ from Net to Net. To abstract from such Net-specifics, types in the support module are parameterized by a *marking* type.

The main simulation functions, *run* and *enabled*, are parameterized by a Net (a collection of transitions) and a marking (a record of token multi-sets on places in the Net). In each step, *run* prints the current marking, and calls *enabled* to determine the list of enabled transitions. One

```
module SimpleHCPN where

import List
import Random

data Net marking         = Net {trans :: [Transition marking]}
                           deriving (Show)
data Transition marking = Transition {name :: String
                                     , action :: marking -> [marking]
                                     }
                           deriving (Show)

select           :: [a] -> [(a, [a])]
select []        = []
select (x : xs) = [(x, xs)] ++ [(y, x : ys) | (y, ys) <- select xs]

enabled                 :: Net m -> m -> [(String, m)]
enabled net marking = [(transName, nextMarking)
                       | (transName, ms) <- [(name t, action t marking)
                                             | t <- trans net
                                             ]
                       , nextMarking <- ms
                       ]

choose             :: [(String, m)] -> IO (String, m)
choose enabledTs = do n <- getStdRandom (randomR (0, length enabledTs - 1))
                      return (enabledTs !! n)

run                :: Show m => Net m -> m -> IO ()
run net marking =
  print marking >>
  if null enabledTs
    then putStrLn "no more enabled transitions!"
    else do print enabledTs
            (transition, nextMarking) <- choose enabledTs
            putStrLn transition
            run net nextMarking
  where
    enabledTs = enabled net marking
```

Fig. 5. A support module for HCPNs

of the enabled transitions is chosen at random (*single-step simulation*) and used to produce the marking for the next step, until no transitions are enabled. *enabled* calculates a list of transitions that are enabled in the current *marking*, conveniently combined with the possible next markings. For each transition t in the Net and for each of the follow-on markings generated by that transition, the result of *enabled* contains a pair of transition name and follow-on marking.

To achieve such a simple simulation loop, the simulation code needs to operate on variable-sized collections of transitions. In combination with the restrictions of Haskell's static type system, this forces us to construct transition actions of homogeneous type, which is the reason why transition actions operate on complete markings instead of their local environment only. *Net*s are then represented as lists of *Transition*s, each described by a name and an action (mapping *marking*s to lists of *marking*s).

The auxiliary function *select* is used to select tokens out of multi-sets, represented as lists. Applied to a multi-set, it generates a list of pairs, each representing one possibility to select a single token from the multi-set (the second component in each pair is the remaining multi-set for that selection). The remaining operation, *choose*, uses the standard *Random* library to choose one of the currently enabled transitions (together with a follow-on marking).

For *multi-step simulation*, sets of concurrently enabled transitions would have to be calculated and chosen at each step (not shown here). To check for concurrent enabling of a multi-set of transitions, the current marking has to be threaded through all transitions, consuming all input tokens before any output tokens are produced to avoid dependencies between transition firings in a single multi-step.

4.4 Haskell Code for the Example

Using the HCPN support module, mapping a HCPN model to Haskell code that simulates the execution of the Net model is pleasingly straightforward. Fig. 6 gives the Haskell code for the Net shown in Fig. 4.

The definition of *review* is simply copied from the Net, the type *Mark* is the record-like type representing markings, $mark :: Mark$ represents the initial marking, and $net :: Net\ Mark$ collects the transitions in the Net.

Transitions are mapped to functions from markings to lists of markings because transitions may be enabled for zero, one, or many selections of tokens on their input places. The auxiliary function *select*, in combination with the use of records to represent markings, and list comprehensions to handle the non-determinism inherent in transition firing,

```
module Review2 where

import SimpleHCPN

review "mary" _ = False
review "paul" _ = True
review ref p    = p

data Mark = Mark {referees :: [(String, Int)]
                 , submissions :: [((String, Bool), Int, [Bool])]
                 , reading :: [((String, Int), (String, Bool))]
                 } deriving (Show)

t_start_review    :: Mark → [Mark]
t_start_review m =
  do
    (((a, p), rc, rs), other_submissions) ← select (submissions m)
    ((ref, c), other_referees) ← select (referees m)
    if rc < 3
      then return m {reading = ((ref, c), (a, p)) : reading m
                     , submissions = ((a, p), rc + 1, rs) : other_submissions
                     , referees = other_referees
                     }
      else fail "guard in start_review failed"

t_finish_review    :: Mark → [Mark]
t_finish_review m =
  do
    (((ref, c), (a, p)), other_reading) ← select (reading m)
    (((a', p'), rc, rs), other_submissions) ← select (submissions m)
    if (a, p) == (a', p')
      then return m {submissions = ((a, p), rc, (review ref p) : rs) : other_submissions
                     , referees = (ref, c + 1) : referees m
                     , reading = other_reading
                     }
      else fail "guard in finish_review failed"

net = Net {trans = [Transition {name = "finish_review", action = t_finish_review}
                   , Transition {name = "start_review", action = t_start_review}
                   ]
          }

mark = Mark {reading = []
            , submissions = [(("unknown", False), 0, []), (("well-known", True), 0, [])]
            , referees = [("peter", 0), ("paul", 0), ("mary", 0)]
            }
```

Fig. 6. Haskell code for the HCPN in Fig. 4

improves code readability. Using Haskell's **do** notation to write down the list comprehensions, the code for a transition closely follows the graphical representation, and failure of pattern-matches or guards is handled implicitly (returning an empty list of follow-on markings).

Each transition action consists of three parts: selecting tokens from input places (uses record field selection and *select*), evaluating the guard to check transition enabling, and placing tokens on output places (uses record field updates). Inscriptions on input arcs have to conform to Haskell's pattern syntax and match the type of the input place, guards are just Haskell's boolean expressions, and inscriptions on output arcs are expressions matching the type of output places.

5 Further Work

For the present paper, we have focussed on a simple, directly useable embedding. The main part of the embedding is a straightforward translation of Net structures and inscriptions into a functional program, complemented by a small amount of support code. Petri Net modelling and simulation (with textual trace) can therefore be used with minimal startup costs. Data types with labelled fields, list comprehensions and **do** notation have been used to make the Haskell code more readable, but the ideas should translate to other functional languages, as long as care is taken to design inscriptions with a good match in the implementation language.

For larger Nets, it becomes impractical to draw Nets and to translate them into functional programs by hand. In line with the idea to minimize the implementation effort needed to get started with functional Petri Nets, we have reused the efforts that have gone into existing tools [1]. We chose Renew 1.0[12], which comes comes with source code (Java) and a flexible license. Renew's graphical editor was used to produce the HCPN graphs (disabling the syntax check), and a new export function was added to translate the internal graph representation into Haskell programs according to the mapping described in Sect. 4.

It took only a few days to set up an environment for Haskell-Coloured Petri Nets, including graphical editing, program generation from graphical specifications, and simulation with a textual trace. The estimate of 1-2 weeks, given in the introduction, would already include further work, e.g., for reading information from textual traces back into the graphical editor to provide graphical representations of intermediate states, or to integrate the Haskell simulator to provide interactive, graphical simulations.

A shortcoming of this minimalistic approach is that type and syntax checking do not take place during editing, but are delayed to the compilation of the generated Haskell code. The generated code closely resembles the Net structure, but it might still be difficult to map error messages referring to the final Haskell code back to graphical Net elements.

Such lack of integration shows that the embedding presented here cannot be a complete replacement for year-long implementation efforts. The limitation of Net inscriptions to facilities offered by the host language has already been mentioned, and efficiency problems can be expected for large nets or large numbers of tokens: representing multi-sets and choices as lists, and recomputing the set of enabled transitions in each step (even though the effects of transition firing should be localized) are weak spots.

Apart from language independence, the main advantage of our embedding is that it offers an economical approach to functional Petri Nets. They can be used now with minimal effort, and further investments into optimizations or better tool support can be made incrementally when they are needed. Embeddings such as the one presented in this paper are also good vehicles for further research and experimentation.

Our own motivation here stems from experience with Design/CPN [4], and we hope to combine research results in functional language design and Petri Nets. To give a simple example of the issues that need to be addressed: neither Design/CPN nor its inscription language ML offers a systematic approach to input/output. On the one hand, the Design/CPN manual contains warnings that side-effecting ML code must be used with care, and must not have any effect on the enabling of transitions, but ML as an inscription language offers no control over side-effecting code. On the other hand, Design/CPN offers no explicit support for interactions between a Net model and an external environment.

The second problem was discussed on the Design/CPN mailing-list [3], and two alternative solutions were proposed – either to introduce special transitions, connected to external events, or to use special places as communication ports to the environment. The motivation behind these proposals is that the simulation of CPN models is often fast enough that the Nets could be used as prototypical controllers for the simulated system. Without any organized connection to the environment, however, the prototyping and modelling phase has to be followed by a complete rewrite before the simulation can be converted into reality.

Because the source code of Design/CPN is not available, experiments with the proposed solutions depend on the developers, who are always helpful but have only limited time available. In this case, the proposals

did not receive high priority for the implementation, forcing us to abandon the idea of a real system controlled by a simulated Net. As far as we know, such extensions are still not implemented.

In contrast, code involved in input/output is marked by the type system in languages such as Haskell or Clean, and the embedding has to be explicitly adapted for such code to be executed. No unexpected effects can occur, and a systematic approach to Net simulation with input/output and side-effects can be sought. Given the embedding outlined in this paper, it is now possible to experiment with such pragmatically motivated extensions without delay. Foreign function interfaces could then be used to control, for example, business processes by workflow models given as high-level Petri Nets with Haskell inscriptions.

6 Conclusions

We have shown that Coloured Petri Nets with a functional inscription language can be embedded in the inscription language in a pleasingly simple way. Using this or a similar embedding, high-level Petri Nets can be used as a modelling tool by functional programmers. This offers the chance to close a significant gap in the use of functional languages – the lack of a graphical specification language. It is not yet clear whether functional Petri Nets can close this gap completely, but such Nets have already proven their worth in the specification of distributed, concurrent and communicating systems. Initial, abstract Net specifications can be refined until all relevant system aspects are modelled, followed by or interspersed with a replacement of Net structures with functional code.

Equally important, such embeddings can be used to experiment with variants of high-level Nets, and to combine research results from Petri Net and functional language research. The localization of communications and changes to system states in Petri Nets corresponds nicely to the careful distinction between external communication and local program transformations in pure functional languages. Both formalisms advocate a disciplined approach to locally comprehensible system specifications, and their combination can offer an attractive alternative to existing graphical specification models for object-oriented development processes.

Preliminary experience with advanced embeddings supporting, e.g., composition of complex Nets from sub-Nets indicates that limitations of Haskell's static type system might complicate the further development. So far, however, the idea to separate the inscription language and the simulation of Net structures has paid off: by implementing the Net sim-

ulation on top of a functional language and reusing the implementation of the implementation language for the implementation of the inscription language, the implementation overhead has been minimized.

We hope that the simplicity of our embedding encourages functional programmers to use Petri Net models, simulated by functional code, in the development of functional programs and to model and simulate complex system behaviour, not only in Haskell or ML.

References

1. World of Petri Nets: Homepage of the International Petri Net Community. `http://www.daimi.aau.dk/PetriNets/`, 1999.
2. Claus Aßmann. Performance Results for an Implementation of the Process Coordination Language K2. In K. Hammond, A.J.T. Davie, and C. Clack, editors, *Implementation of Functional Languages (IFL '98), London, UK*, volume 1595 of *LNCS*, pages 1–19. Springer-Verlag, September 1998.
3. CPN group at the University of Aarhus in Denmark. Design/CPN – Computer Tool for Coloured Petri Nets. `http://www.daimi.aau.dk/designCPN/`, 1999.
4. W. Hielscher, L. Urbszat, C. Reinke, and W. Kluge. On Modelling Train Traffic in a Model Train System. In *Proceedings of the Workshop and Tutorial on Practical Use of Coloured Petri Nets and Design/CPN, Aarhus, Denmark*. Available online (`http://www.daimi.aau.dk/CPnets/workshop98/`) and as technical report PB-532, Department of Computer Science, University of Aarhus., June 1998.
5. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Springer Verlag, 1997. Three Volumes.
6. Kurt Jensen. Recommended Books and Papers on Coloured Petri Nets. `http://www.daimi.aau.dk/~kjensen/papers_books/rec_papers_books.html`, 1999. Includes links to online articles covering introductions, theory, and practice.
7. S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A Non-strict, Purely Functional Language. Technical report, February 1999. Available at `http://www.haskell.org`.
8. Werner E. Kluge. A User's Guide for the Reduction System π-RED$^+$. Technical Report 9419, Institute of Computer Science and Applied Mathematics, Christian-Albrechts-University, Kiel, December 1994.
9. Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.
10. W. Reisig. *Petri Nets, An Introduction.* EATCS, Monographs on Theoretical Computer Science. Springer Verlag, 1985.
11. Jörg Schepers. Using Functional Languages for Process Specifications. In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 89–102, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.
12. Theoretical Foundations Group, Department for Informatics, University of Hamburg. Renew – The Reference Net Workshop. `http://www.renew.de/`, 1999.