

Tool Support for Haskell-Coloured Petri Nets (*extended abstract*)

Claus Reinke

`claus.reinke@talk21.com`

Abstract. Haskell-Coloured Petri Nets (HCPNs) are one instance of a class of high-level Petri Nets which combines graphical description of concurrent system structure (causal dependencies, concurrency, conflict over shared resources, data transport) with functional specification of data manipulation. In an earlier paper [6], we introduced HCPN to demonstrate how any functional language supporting a few common features (such as records and list comprehensions) can support Petri nets with inscriptions in that language via a straightforward embedding, thus giving functional programmers easy access to a rich hybrid graphical/textual specification formalism.

We report on work in progress to provide tool support for the specific example of HCPN, including a concrete example implementation of our embedding together with support for graphical editing and simulation. Graphical user interfaces add their own complexities, but the focus continues to be on simple implementation, supporting easy experimentation with language design options, with a view on strengthening the functional programming aspects of this hybrid specification formalism. After about a month, the tools have already reached a state in which they can be used for small examples and demonstrations, e.g., for teaching purposes. To bring project work, workflow systems, and general system modelling within reach, we will add support for hierarchical nets.

By providing Haskellers with their own graphical modelling language, we also hope to encourage parallel implementations of the ideas in other functional languages (especially those concerned with concurrent systems, such as Erlang). The aspect of Haskell that most complicates the development (apart from the nitty-gritty of graphical user interfaces) is its lack of support for runtime reflection, for which workarounds are just becoming available.

1 Introduction

Since their introduction in the 1960s as a concurrent extension of automata, Petri nets [5, 7] have become one of the most successful examples of how a domain-specific language can serve as an interface between computer science and domain experts. Domain-engineers use the seemingly informal “token game” view of nets for simulation, analysis, performance prediction, documentation and communication of models of concurrent and distributed systems in areas ranging over manufacturing systems, telecommunication or transport networks, office document

workflow, or - more recently - biological pathways. Meanwhile, computer scientists take a formal view of nets to implement simulation and analysis tools, to study the theoretical properties of pragmatically interesting classes of nets or the interaction between the various extensions arising from industrial use. For further information, see the Petri Net Homepage [1].

The common language of Petri nets allows practitioners and researchers to cooperate on progress in application domains and formal background and to harness insights from either of the two for benefit of the other. The common language also serves as a separator, avoiding spill-over of specialist knowledge: computer scientists are not forced to become experts in, say, biotechnology or business processes; nor are bioscientists forced to become computer experts if they want to employ computers for modelling, simulating, analysing, designing, and documenting the biotechnological processes they are interested in. To see how unusual this situation is, the reader may want to contrast it with standard software development practice, where only computer experts do the programming, and where acquisition and proper encoding of domain knowledge is a major cost and failure factor as these experts in the wrong domain try to deliver pre-packaged software to their “users”.

We suggest to combine Petri nets with functional languages in such a way that, for a given functional language F , F -Coloured Petri nets have a simple implementation by a straightforward embedding into F . This offers functional programmers easy access to the world of Petri nets, combining the expressive textual notation of functional programs for specifying data manipulation aspects with the tailor-made graphical notation of Petri nets for specifying aspects of concurrent system structure (causal dependencies, concurrency, conflict over shared resources, synchronous/asynchronous communication between subsystems).

Our earlier paper [6] focussed on the embedding, introducing Haskell-Coloured Petri nets (HCPN) as an illustrative example, and gave a more detailed introduction to Coloured Petri nets as well citing some earlier work on combinations of Petri nets and functional languages. The present paper focusses on HCPN as a practical hybrid graphical/textual modelling language, and specifically reports on work in progress to provide tool support that will enable Haskell programmers to add this modelling notation to their repertoire. In the following sections, we briefly recall the definition of HCPN and their embedding in Haskell, before focussing on the existing tools for graphical editing and simulation of HCPN. As Haskell programmers tend to be unfamiliar with high-level Petri nets and their use in modelling concurrent systems, we use our tools to illustrate a typical HCPN modelling process via a concrete, though simple, example. We highlight some unusual aspects of the implementation and outline our plans for further development.

2 Haskell-Coloured Petri Nets

Petri nets are commonly introduced as bipartite directed graphs, with place nodes holding resources, transition nodes consuming and producing resources,

and arcs between these two kinds of nodes specifying dependencies between transitions and places, and the resources on them. A transition is *enabled* if sufficient resources are available on its input places, and an enabled transition can *fire* by consuming resources along each input arcs while synchronously producing resources along each output arc. High-level nets, and especially *Coloured Petri nets* (CPN), move from this basic model with anonymous resource tokens to one with individual resource tokens: tokens become data objects in some inscription language, places hold multisets of such tokens, and transitions may manipulate tokens according to code in the inscription language, as well as transport them between places.

Haskell-Coloured Petri nets [6], then, are a form of high-level Petri nets using Haskell as the inscription language: places have types, tokens on them are expressions of those types, transition input arcs may be labelled with patterns of those types, transition output arcs may be labelled with expressions of those types, and transitions may be labelled with boolean-valued guard expressions (variables in input arc patterns scope over guard and output arc expressions of the same transition), all net inscriptions may refer to Haskell declarations and definitions associated with the net. The firing rule is adapted accordingly: a transition is *enabled* if each input place holds a token matching the input arc pattern, and the transition guard evaluates to `True` under the bindings created by those matches; an enabled transition may *fire* by consuming the enabling tokens from its input places and producing tokens on its output places that correspond to the output arc expressions instantiated with the enabling bindings.

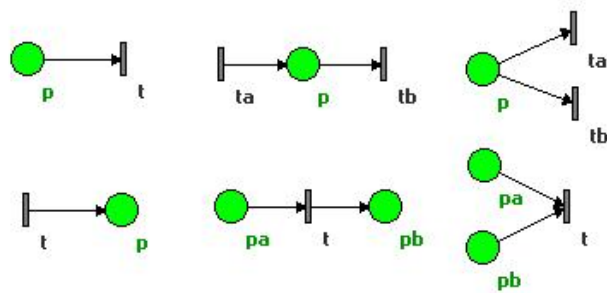


Fig. 1. Consumer, producer, causal dependency, conflict, synchronisation

From a modelling perspective, it is more useful to look at the basic actions in nets together with ways to compose nets in order to form complex nets (Fig. 1). There are two elementary actions: consuming a resource from a place, and producing a resource on a place. These actions and more complex nets can be composed in various ways: they can be completely independent of each other (modelling concurrent activities), they may share places (modelling producer/consumer relationships and asynchronous communication between sub-

nets, or modelling conflicts over shared resources), and they may share transitions (modelling synchronisation of actions and synchronous communication between subnets).

3 Embedding HCPN in Haskell

HCPN have been defined to allow for a straightforward embedding in Haskell [6], i.e., the elements of a Haskell-Coloured Petri net can be mapped to a Haskell program that simulates the token game of the net.

The places of a net form a static, heterogenous collection, as each place may have a different type – this maps to a record type, with a field for each place in the net. Every place holds a multiset of data objects belonging to the place type – this maps to a list of the place type. Markings of a net with tokens then simply are instances of the record type.

Simulating a HCPN involves finding the transition instances that are enabled in the current marking (a transition may be enabled for multiple choices of input tokens, each choice determining a separate transition instance), selecting one of them at random, and changing the marking according to the firing rule for that transition instance. Since the firing rule for HCPN is tightly interwoven with aspects of Haskell semantics (pattern matching, variable binding, guard evaluation), we chose to keep the simulation loop simple while moving all the interesting work into the code for each transition.

To account for transitions being enabled several times or possibly not at all, the code for a transition maps a net marking into a list of follow-on markings (the list is empty if the transition is not enabled). By using Haskell’s `do`-notation for working in the list monad, the transition code can be written to reflect the three phases of the firing rule: checking for tokens matching the input arc patterns, evaluating the guard under the binding established by the matches, and updating the marking to consume input tokens and produce output tokens.

A helper function `select` returns all possible ways of selecting one element from its input list, so each input arc maps to a line of code: reading the place field from the marking record, splitting the place marking into a token and the remaining multiset, and matching the token against the input arc pattern. The transition guard maps to a conditional, and updating the marking maps to record updates. Failure in the list monad, through pattern match failure or the guard evaluating to `False`, simply results in an empty list being returned, and variable bindings are handled by placing the guard and output arc expressions in the scope of the pattern matches.

Figures 2 and 3 show an example HCPN transition using most features, and its translation into Haskell code (markings for places used in input- or output-arcs are factored out into `let` bindings so that they can be more easily threaded through the code for all input arcs). Figure 4 outlines the structure of the Haskell code for a complete net, using the same example: following imports of support code, any Haskell declarations are copied from the net, followed by the type `Mark` of markings and the code for each of the transitions; the net structure

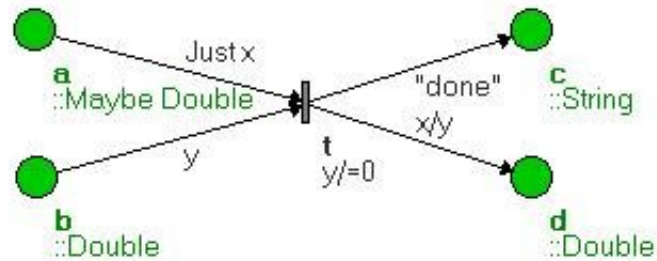


Fig. 2. Example transition with input-arc patterns, guard, and output-arc expressions

```

t :: Mark -> [Mark]
t m =
  do
    let b_marking = b m
        a_marking = a m
        d_marking = d m
        c_marking = c m
        (y, b_marking) <- select $ b_marking
        (Just x, a_marking) <- select $ a_marking
        if y/=0
        then return m{
            b = b_marking
            , a = a_marking
            , d = (x/y) : d_marking
            , c = ("done") : c_marking
          }
        else fail "guard failed"

```

Fig. 3. Haskell code generated for the example transition

is then given by the list of transitions (code plus name and additional info for simulation traces), and the initial marking as a record of the type `Mark`; both net and marking are passed to the generic simulation loop. For a more detailed discussion of the embedding, we refer the reader to our earlier paper [6].

```

.. -- imports
-- declarations (none here)
-- markings
data Mark = Mark {
    d :: [Double]
  , c :: [String]
  , b :: [Double]
  , a :: [Maybe Double]
} deriving Show
-- transition actions
t :: Mark -> [Mark]
..
-- transitions
net = Net{trans=[ Trans{name="t",info=Nothing,action=t} ]}
-- initial marking
mark = Mark{ d = [] , c = [] , b = [] , a = [] }

main = simMain "transition.hcpn" showMarking net mark

```

Fig. 4. Schematic of generated code, here for the transition example net

4 An Example – The Starving Philosophers

System modelling with Coloured Petri nets has many similarities to functional programming, in that there is a focus on local state and transformations, as opposed to the global state and transitions of state machines and procedural programming. Also, models are built from subnets that are composed in various ways to reflect their interactions. An introduction into net modelling techniques is beyond the scope of this paper, but we will try to give some of the flavour by showing a couple of steps in modelling a variant of Dijkstra’s Dining Philosophers problem.

A popular approach to modelling a concurrent system is to find the sequential subsystems and model them as finite automata (which are just a sequential subclass of nets), then to connect the subnets to model aspects such as causal dependencies, conflict over shared resources, synchronous or asynchronous communication, or concurrency. In our case, each of three philosophers goes through the same cycle: think, take one fork, take other fork, eat, put down forks. We model thinking, eating, and waiting for the other fork as states, and fork taking and putting down as transitions. The three subnets for the three philosophers

are still clearly visible in the complete net of Figure 5 (note that place types and arc labels default to `()`, and that the initial marking has one `()` token on each of the three `forkN` places and on each of the three `philN_ready` places).

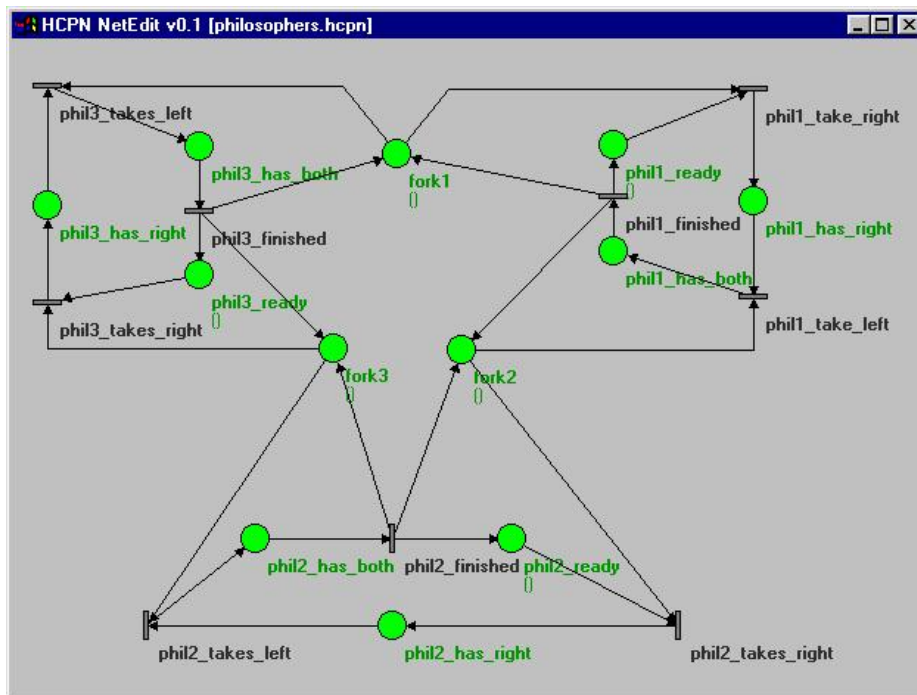


Fig. 5. Philosophers as a plain Place/Transition Net

Without the fork places, we would have one net consisting of three independent finite automata – note that such concurrent composition is possible without obscuring the structure of the subnets (as it would happen when building the product automaton). Adding the fork places brings the shared resources into the model, and synchronising the taking and putting down of forks with the corresponding transitions in the philosopher subnets models the conflicts between the philosopher actions (note how the compositions of Figure 1 reappear again and again).

So far, the model is a plain Petri net, with anonymous tokens only, expressing the whole problem in the structure of the net. This is often inconvenient, and leads to unmanageably large nets, which is where functional abstraction and individual (“coloured”) tokens come into play. For instance, we can introduce a sum type to distinguish the three forks (for simplicity, we use `Int`), change the type of the fork places, and change the arc labels to select and return the appropriately identified fork. Having now two ways to distinguish forks, by place

and by colour, we can drop one of them and fold the three fork places into one without changing the firing behaviour of the model. We can then proceed to do the same for the philosophers, folding the three subnets into one while identifying the philosopher tokens as `Ints`. The result of such folding is shown in Figure 6 (now, the initial marking has forks 0,1,2 on the `forks` place, and philosophers 0,1,2 on the place `phil_ready`; the forks to work with are computed from the philosopher id).

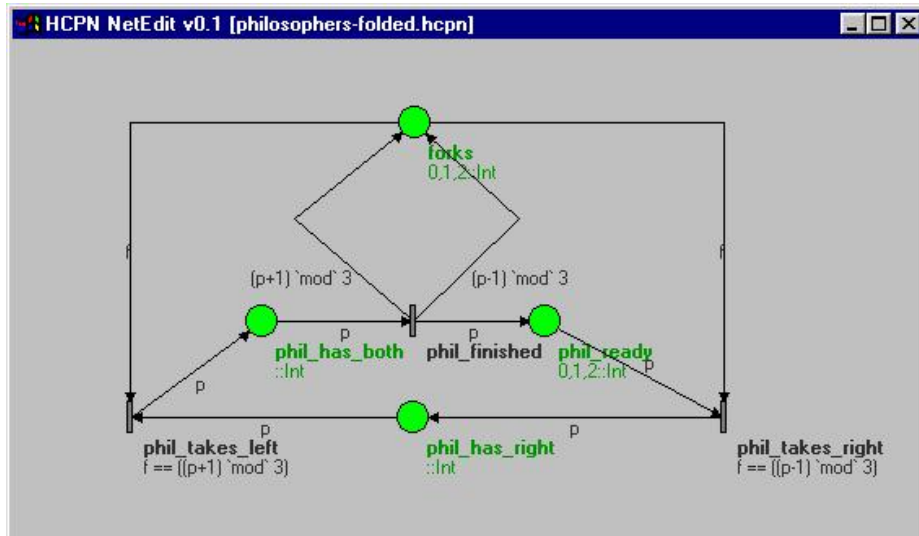


Fig. 6. Philosophers folded – replacing structure by colour

Such trade-offs between net structure and token colour are typical for system modelling with coloured nets: when modelling for analysis, the aim would be to express the relevant concurrent system aspects in the net structure while covering other aspects as data manipulation in the inscription language; when modelling for design, one might start with an abstract low-level model of the system structure, then refine that model by adding colour.

5 HCPN NetEdit and NetSim

As given, HCPN can be seen as graphical syntactic sugar for a class of Haskell programs that simulates concurrent systems – a wide-spectrum embedded language (in contrast to the more typical domain-specific embedded languages). Even for the tiny example in the previous section, however, it would be tedious to work with the textual representation of HCPN, and a graphical editor is needed to take full advantage of this hybrid graphical/textual modelling language, with simulation code generated automatically from the net representation. Moreover,

the simulation should use the same graphical format, and there should be no need to inspect the generated code.

Since HCPN were first introduced, the situation of GUI libraries for Haskell has improved again, following the infamous cycle of first-too-many-then-too-few libraries. One library in particular, wxHaskell [3, 2] has set out to avoid short-term research into functional approaches to GUI specification in favour of pragmatic issues such as portability and long-term maintainability, by providing a thin binding to the popular wxWidgets library. Building on wxHaskell, it has been relatively straightforward to write a useable graphical editor for HCPN, if not as easy and convenient as it could have been (like many other GUI libraries, wxHaskell provides more support for GUI widgets than for drawing, and at the time of writing, the binding is still under development, approaching its 1.0 release).

Instead of detailing the fairly standard programming of the graphical editor, we will focus on some non-standard issues raised by the interaction of the usage envisioned in the first paragraph of this section with the chosen implementation of HCPN as an embedding compiler (generating from nets Haskell programs of a certain shape). Just to summarize, HCPN NetEdit v0.1 supports a keyboard/mouse-driven approach to editing HCPN (no menus, minimal interference of GUI widgets with the modelling task at hand, some automatic layout of labels to avoid wasting time with label placement), allowing nets to be created, saved, loaded, nodes to be selected, moved (connected arcs follow), deleted, and annotated with names, types, guards, connected with arcs, which may be labelled, and segmented to help layout, and Haskell code for simulation to be exported. There is a growing list of standard graphical editing functionality that would be nice to add, but NetEdit is fairly useable as it stands, with only few items on the todo-list falling into the category of necessary additions.

The graphical simulation part, NetSim, and its integration in the toolchain are more interesting - due to the approach of using a domain-specific embedding compiler from a hybrid graphical/textual input language to Haskell. The main problem is the lack of runtime support for reflection in Haskell: the whole point of embedding HCPN in Haskell by generating Haskell code from nets was to avoid having to re-implement Haskell, but in the context of a graphical development environment, we would like to compile and load the generated code behind the scenes, and it should operate on the net we are editing, to visualise the intermediate markings of the net simulation. Another problem of the code generation approach is that error detection is deferred until the generated code is compiled, making it difficult to relate error messages back to the graphical view of nets.

So we would want to compile and run generated code from within our editing session - runtime code generation and loading. This is not usually supported in Haskell: although (frustratingly) all interactive Haskell implementations need to do this, none of that functionality is available to the Haskell programmer without hacking the implementations. There is some support for such implementation-specific hacks (Hugs Server API, GHC as a package, hs-plugins [8, 4]), but Haskell

lacks a standard runtime reflection API for exposing this functionality in a portable and type-safe way.

While we await progress on this front, we have chosen the following work-arounds:

- instead of the graphical simulation code modifying the net currently edited, it starts its own copy of the HCPN GUI code, loads the HCPN model, and animates the simulation on that copy.
- instead of compiling and loading the exported code at runtime, we start GHCi as an external process in which we load and run the exported code, making it look as if starting the simulation was just like starting another window (albeit slowly, due to GHCi startup and load time).
- instead of analysing the static correctness of HCPN during editing (which would have to go beyond parsing into scope analysis and typing), we capture the error messages from GHCi on loading the exported code; the structure-preserving embedding makes it possible to relate source code line numbers back to the net elements (place, transition, guard, input/output arc) for which that code segment was generated, but GHC's source locations are not very precise, and the whole scheme causes an unfortunate delay in providing useful feedback.
- at the time of writing, Haskell also lacks a portable way of starting a subprocess while interaction with its stdio streams (this is planned for forthcoming releases of the hierarchical libraries), but wxHaskell provides a portable solution to this (wxHaskell provides several non-graphical widgets).

6 Further work

hierarchical nets This is a must for modular construction of larger models, but is sometimes treated in a more or less ad-hoc manner. We would like to make an impact here, by using functional programming not just inside nets, but over nets, to support functional abstraction over and composition of nets. The challenges are two-fold: find an appropriate semantics, and implement within the constraints of our chosen embedding.

state-space generation Instead of choosing one possible execution sequence non-deterministically for graphical simulation, it is often useful to generate the whole state space for a net. The problem of deciding whether certain (un-)desirable markings are reachable from the initial marking is undecidable for coloured nets in general, but that does not mean that model-checking is always infeasible in practice.

Problems: we do not require equality on token types as we want to support functions and abstract data types, also we would like to add input/output to interface net models with the outside world.

exotic net types The practical success of high-level Petri nets has led to a variety of pragmatically motivated extensions, such as test arcs (non-consuming read of token), inhibitor arcs (disabling transitions on presence of tokens), timed nets (to model real-time properties, or to analyse performance characteristics of the processes modelled), stochastic nets (to ease modelling of scheduling issues), hybrid nets (continuous transitions in addition to the standard ones), self-modifying nets, etc. etc..

Problems: while undoubtedly useful, and demonstrably implementable, it is not always clear what the semantic properties of such extensions are (though net theory has been making good progress on investigating these), or what the desirable combinations from a language design perspective are.

7 Conclusions

Lots remains to be done, but the existing toolset is already useable, and with a few more improvements of basic GUI interactions, we expect to move on to more interesting design issues, such as functional abstraction over nets. The existing toolset, thanks to the simplicity of the embedding, provides a good basis for exploration of the design space opened by the combination of functional programming and Petri nets, beyond simply using functional programs inside nets as has been the focus in earlier implementations of this combinations.

Applications range widely, effectively opening the field of successful applications of Petri nets to functional programmers: teaching operating system and concurrency concepts (as opposed to using functional languages only in compilers and abstract machine courses), workflow modelling and performance prediction (could profit from more control over scheduling probabilities and from modeling time more explicitly), more recent developments include modeling, documenting and archiving biological pathways.

We plan to make a first public release of our HCPN toolset in connection with IFL 2004. In the meantime, snapshots are available at <http://www.cs.kent.ac.uk/~cr3/HCPN/>.

References

1. World of Petri Nets: Homepage of the International Petri Net Community. <http://www.daimi.aau.dk/PetriNets/>, 1999.
2. Daan Leijen. wxhaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, September 2004.
3. Daan Leijen. wxHaskell, a portable and native GUI library for Haskell. <http://wxhaskell.sourceforge.net/>, 2004.
4. André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging Haskell In. In *ACM SIGPLAN 2004 Haskell Workshop*. ACM Press, September 2004.
5. 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.

6. Claus Reinke. Haskell-Coloured Petri Nets. In P. Koopman and C. Clack, editors, *Implementation of Functional Languages (IFL '99)*, Lochem, The Netherlands, volume 1868 of *LNCS*, pages 165–180. Springer Verlag, 2000.
7. W. Reisig. *Petri Nets, An Introduction*. EATCS, Monographs on Theoretical Computer Science. Springer Verlag, 1985.
8. Don Stewart. hs-plugins, Dynamically Loaded Haskell Plugins. <http://www.cse.unsw.edu.au/~dons/hs-plugins/>, 1997.