Translating HCL Programs into Petri Nets

Ricardo Massa Ferreira Lima[*]Rafael Dueire Lins[†]

key-words: Formal Methods, Parallel Programming, Petri Nets

Abstract. *Haskell$_\#$* is a parallel extension to lazy functional language Haskell. The sequential part of programs is declared using standard Haskell. This reduces code development costs and increases its reliability by reusing existing and previously tested (or formally verified) Haskell modules. The structure of a process network (of possibly heterogeneous processors) is defined by *Haskell$_\#$ Coordination Language* (HCL), also used for task-to-processor allocation. In this work, we present an environment for analyzing formal properties of the communication structure of *Haskell$_\#$* programs. This environment contains a compiler that translates an HCL program into Petri nets, a well established family of formal specification techniques for modelling non-deterministic concurrent systems. Petri nets allow the analysis of a wide spectrum of properties, such as liveness, boundedness, and deadlock-trap.

1 Introduction

The concept of what is a functional language has evolved with time, but its main feature is that a program is a set of definitions of higher-order functions – functions are not only passed as parameters to other functions, but can also be the result of the evaluation of a given function. According to this definition, LISP [16] was the first functional programming language.

Two other features are also used to classify functional languages: the existence (or absence) of destructive assignment and their evaluation mechanism. If a language has no destructive assignment, the value of any sub-expression is static, it is said to be "pure" or to enjoy the property of *referential transparency*. If the arguments to a function are passed by value, i.e. all the arguments are evaluated before the function itself, it is called "strict". Non-strict languages may be implemented using either a *data-driven* (also known as dataflow) or a *demand-driven* approach. Demand driven languages evaluate arguments as required by the function. If the result of the argument is recomputed each time it is needed, this evaluation mechanism is called *call-by-name*. Conversely, if the result is shared the evaluation mechanism is called *call-by-need*, and the language is said to be *lazy* or *procrastinating*. Non-strict languages have the advantage that only expressions which must be evaluated to give the program result actually are evaluated. They allow the use of infinite data structures, such as infinite sets and lists, which are described by a formation law. These structures can be seen as intensionally defined.

LISP and SML [26] are examples of strict impure functional languages. HOPE [5] and OPAL [7] are strict pure functional languages. While Miranda[1] [37], Haskell [14], pH [29], and Id [28] are instances of pure functional languages.

An overview of the relationship between parallelism and functional programming is presented in [25]. The recent book [10] presents research directions in parallel functional programming. The idea of parallel functional programming dates back to 1975 when Burge [4] suggested the technique of evaluating function arguments in parallel, with the possibility of

[*]Centro de Informática - UFPE `rmfl@cin.ufpe.br`
[†]Departamento de Eletrônica e Sistemas - CTG - UFPE `rdl@cin.ufpe.br`
[1]Miranda is a trademark of Research Software Ltd.

functions absorbing unevaluated arguments and perhaps also exploiting speculative evaluation. In general, the parallelism obtained from referential transparency only has fine granularity not yielding good performance. The search for ways of controlling the degree of parallelism of functional programs by means of automatic mechanisms, either static or dynamic, had little success[12, 19, 21]. On the other hand, explicit parallelism with annotations to control the evaluation demand of expressions, the creation/termination of processes, the sequential and parallel composition of tasks, as well as the mapping of these tasks onto specific processors have been proposed by many authors [6, 13, 32, 36].

If compared with compilers that exploit implicit mechanisms, such a strategy produces better performance levels. However, it has two main limitations:

1. they are geared towards shared memory architectures, which brings problems of portability;
2. the computation and communication structure are intertwined not allowing the understanding of these elements in isolation [3].

In order to overcome these difficulties, we developed $Haskell_{\#}$[24], a parallel extension to the lazy functional language Haskell[14], which offers a process hierarchy on top of communication structure and sequential computation components. These two levels, are defined at independent stages of the development process. The computation of sequential tasks is declared using the standard Haskell functional language, which makes possible a reduction in the cost of development and an increase in reliability because of the reuse of existing and previously tested or formally verified Haskell modules. The structure of communication network (of possibly heterogeneous processors) is defined by means of the coordination language, called $Haskell_{\#}$ Coordination Language (HCL), also used for task-to-processor allocation.

The development of $Haskell_{\#}$ suffered strong influence from Occam's[15] computation model, a language based on Hoare's CSP (Calculus of Sequential Processes)[11]. The decision to follow Occam's computation model, had as its goal to make possible the automatic analysis of formal properties and, thus, to help the programmer to reason about the application under development. In this work, we present an environment for analyzing formal properties of the communication structure of $Haskell_{\#}$ programs. This environment contains a compiler that translates an HCL program into Petri nets, a well established family of formal specification techniques for the modelling of concurrent and non-deterministic systems. It was first introduced by C. A. Petri[31] in the 1960's to model and analyze communication systems. Since then, Petri nets have been studied extensively (e.g. [30], [34], [27]) because of their wide applicability in several areas, such as, computer science, electronics, chemistry and business management. Petri nets have many advantages, including their well-developed mathematical foundation; compact representation of system specifications; several tools for both qualitative and quantitative analysis; and graphical "nature" that aids the understanting of complex systems. The formalism behind Petri nets makes possible the analysis of a wide spectrum of properties, such as liveness, boundedness, deadlock-trap, etc[27]. In particular, our compiler translates the HCL description into the format of Petri nets used by the computational tool INA[35], capable of simulating and automatically analyzing the formal properties of the systems modelled by Petri nets.

A $Haskell_{\#}$ application contains a number of instances (processes) of conventional Haskell modules, which cooperate for perform a given activity. Figure 1 depicts the $Haskell_{\#}$ programming environment. After the program development, there are two possibilities: either to generate the executable code; or to translate the HCL program into Petri nets. In the former situation, the system will execute in a high-performance environment composed of heterogeneous loosely coupled processors. In the second case, in order to identify the formal properties and help programmers to reason about the specified topological network structure, the resulting Petri net model is simulated and/or analyzed through the computational tool INA. $Haskell_{\#}$ provides an integrated environment for execution and analysis of formal properties of distributed systems.

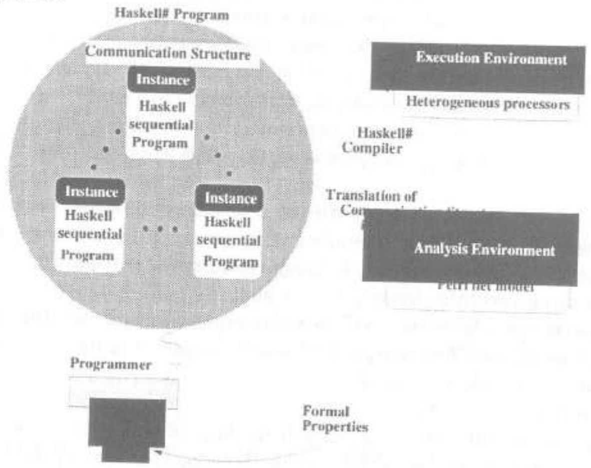


Figure 1: $Haskell_{\#}$ Programming Environment

2 Haskell

Haskell is a general purpose, pure functional programming language incorporating many recent innovations in programming language research, including higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, automatic garbage collection, pattern-matching, list comprehension, a module system, monads, and a rich set of primitive datatypes, including arrays, arbitrary and fixed precision integers, and floating-point numbers[14]. Haskell has now become *de facto* standard for the non-strict (or lazy) functional programming community, with several sequential compilers available. $Haskell_{\#}$ uses the Glasgow Haskell Compiler (GHC)[9] to implement the sequential pieces of Haskell code.

There are two other parallel versions of Haskell: Concurrent Haskell[20] and GUM[36]. Concurrent Haskell is a concurrent extension to Haskell, which provides a more expressive substrate to build sophisticated I/O-performing programs, notably ones that support graphical user interfaces for which the usefulness of concurrency is well established. The goal of the designers of Concurrent Haskell is to obtain implicit, semantically transparent parallelism, but the version available now uses explicit parallelism. GUM is a portable, massage-based parallel implementation of Haskell using PVM[8] communications harness. GUM is available for both shared-memory (Sun SPARCserver multiprocessors) and distributed-memory (network workstations) architecture. GUM's performance figures demonstrate speedups relative to sequential compiler technology.

3 *$Haskell_{\#}$ Coordination Language* - HCL

This section presents *$Haskell_{\#}$ Coordination Language* (HCL), developed for instantiating and distributing processes over the physical nodes and establishing a connection between them through unidirectional, point-to-point synchronous channels.

3.1 Functional Processes

$Haskell_{\#}$ processes are instances of Haskell functional modules. It is possible define more than one instance of a particular Haskell module, that will present the same behaviour. All processes are created at the start of the *Haskell*$_\#$ program execution and finish together at the end of the *Haskell*$_\#$ program execution.

Because *functional processes* are instances of simple Haskell modules, it is necessary to provide an interface (communication port) from which they can send/receive messages to/from other functional processes. In *Haskell*$_\#$, each communication port is strongly typed. In addition, it has strict semantic on communication. As a result, the types of ports are restricted to ground types, such as basic types (integer, float points etc.) and structured types in *head normal form*[18] (lists, trees, arrays etc.). Functions, IO operations, channels and other ports are not valid types. Eventually, a port declaration should define the *direction* (input or output) of communication.

A functional process is defined by means of a *module* declaration, which defines the instances of a given functional Haskell module and its respective input/output ports. Figure 2 presents the syntax for functional module instance definition. According to the syntax, the port names of a given **module** declaration are shared by all instances defined in its scope. There is a direct correspondence between the order ports appear in the **input/output** declarations and the type of *main* function of the Haskell module. For instance, the declarations:

input $a :: t_1,\ \{b, c\} :: t_2,\ d :: t_3$
output $e :: t_4,\ f :: t_5$

define the input and output ports of a Haskell module, in which the type of *main* function is given by: $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow IO(t_4, t_5)$ In this case, the *main* function should receive three parameters with types t_1, t_2 e t_3 mapped, respectively, onto ports a, (b, c), e d of the **input** declaration. The function should return an IO operation that, whenever executed, will produce the tuple (t_4, t_5), from which the first component is mapped onto output port e, while, the second, onto port f. It should be noted that port b and c are associated with the same input parameter (t_2). This indicates that both ports are waiting for the second parameter of the *main* function.

The declaration of input/output ports are optional. Based on these declarations, it is possible

Syntax			Comments
module	→	**module** *mod input output instances*	
input	→	**input** *portsI*	input ports declaration - optional
output	→	**output** *portsO*	output ports declaration - optional
instances	→	**instances** *processes*	functional modules instances declaration
mod	→	*id*	functional module name
portsI	→	*port , portsI*	set of input ports
	|	*portNDs , portsI*	
	|	*port*	
	|	*portNDs*	
portsO	→	*port , portsO*	set of output ports
	|	*port*	
port	→	*id :: types*	type port definition
portNDs	→	{ *portND* } :: *types*	type ports definition
portND	→	*id , portND*	port for non-deterministic choice
	|	*id*	
processes	→	*process processes*	
	|	*process*	
process	→	*id*	name of a functional process instance
id	→	*Haskell function identifier*	
type	→	*Haskell data type*	

Figure 2: Syntax for functional module instance definition

to classify functional processes into four categories:

1. **conventional processes:** contains input and output ports; their execution start only when other processes provide the input data they expect;
their execution terminate as soon as other processes consume the output data they produce;
2. **root processes:** contain no input ports; their execution start together with the global program;
3. **terminal processes:** contain no output ports; they have no restriction for terminating their execution;
4. **root and terminal processes:** join the features of *root* and *terminal* processes.

Applications containing only *root and terminal* processes can be thought as a set of parallel independent processes, whereas applications without any *terminal* or *root* processes can be seen as a network of cooperative processes. At this point, one can deduce that *if conventional processes have no autonomy to start, then applications that contain only conventional processes will always enter in deadlock.* To avoid this problem, the **start** declaration was included in HCL. It defines which processes have autonomy to start their execution. The **start** declaration also provides the initial values to processes with input ports. It is important to emphasize that *root* processes should be declared in the scope of a *start* constructor. Finally, the set of start declarations define initial state of the system. Thus, only one start declaration should appear. The syntax of a **start** declaration is presented in the Figure 3.

3.2 Communication Channels

Similarly to the Occam[15] model, *Haskell*$_\#$ channels are point-to-point, unidirectional and synchronous. In addition, the strict communication semantics of *Haskell*$_\#$ restricts the values exchanged between processes to those already evaluated.

The concept of channel is related to the concept of communication port, which represents the interface of functional processes. A channel is statically declared through the connection of two ports of the same type, opposite direction using the **connect** constructor. The syntax of a **connect** declaration is defined in Figure 4.

Syntax			Comments
start	→	**start** *process args*	
args	→	*value args*	sequence of initialization values
	|	*value*	
process	→	*id*	process name
value	→	*valid Haskell value*	
id	→	*identifier of Haskell function*	

Figure 3: Syntax of a **start** declaration

Syntax			Comments
connect	→	**connect** *process.port_out* **to** *process.port_in*	
process	→	*id*	process name
port_out	→	*id*	output port
port_in	→	*id*	input port
id	→	*Haskell function identifier*	

Figure 4: Syntax of communication channel declaration

A set of semantic errors users perform during **connect** declarations are detected by the *Haskell*$_\#$ compiler:

error 1 connection between two ports with the same direction;
error 2 connection between two ports with different types;
error 3 the same port participating of more than one connection;
error 4 connection between two ports belonging to the same process.

Haskell$_\#$ does not allow dynamic channel creation or communication in both directions. One could argue that this is too restrictive. We want to provide a model of channel that makes possible to statically analyze formal properties of the process network. In addition to this, the strict rules force programmers to have a better understanding of system under development and to specify precisely what they want to do.

3.3 Process Termination

A $Haskell_{\#}$ application terminates only when all its processes have finished. Due to this synchronous semantics, $Haskell_{\#}$ processes terminate only when all their values in the output ports had been consumed by other processes.

At this point, it is important to present the concepts of *repetitive* and *nonrepetitive processes*. *Nonrepetitive processes* are those that start, execute some activities and terminate. *Repetitive processes* never reach the termination state. They start, execute some activities and return to the initial state in order to perform new tasks. Note that the "actors" concept[2] does not apply to *repetitive processes*, because they have no memory and keeps the same behavioural pattern through different interactions. An operating system is a classic application of *repetitive processes*. This class of processes is also useful in monitoring and controlling activities.

Termination is a global property, being inherited by all processes in the application. Thus, a given application can contain either *repetitive processes* or *nonrepetitive processes*, but never both simultaneously. The termination property is declared using the syntax in Figure 5.

Syntax			Comments
application	→	**application** *app_name property*	
app_name	→	*id*	termination application
property	→	**repetitive**	
	|	**nonrepetitive**	nontermination application
id	→	*Haskell function identifier*	

Figure 5: Syntax for declaring the termination property of a $Haskell_{\#}$ application

3.4 Mapping of Functional Processes

The execution environment of $Haskell_{\#}$ applications is composed by a network of possibly heterogeneous processors. HCL allows programmers to define the configuration of the machine required by a group of processes. If more than one machine with the required configuration is available in the network, the compiler selects one of them. Thus, programmers informs the machine configuration required by a group of processes, but say nothing about machines physical addresses. This does not restrict programmers to make a physical mapping, by defining machine addresses as its configuration.

A file named *nodeclasses* is used to define the set of features to be used in order to load processing nodes. This file should follow the syntactic rules presented in Figure 6. There are two or more features in each category. For instance, the category *speed* could have *fast* and *slow* as its possible features. The features of a particular processing node is specified inside a file called *nodeid* according to the rules presented in Figure 7. Processes to processors

Syntax			Comments
features	→	*feature features*	set of node features
	|	*feature*	
feature	→	*id , feature*	features belonging to a category
	|	*id* \n	
id	→	*Haskell function identifier*	

Figure 6: Syntax for declaring the categories of machines belonging to a network

mapping is performed statically. Therefore, functional processes are mapped into a processing node of a network when the application starts until it ends. The **alloc** declaration is used to

Syntax			Comments
nodes	→	*node nodes*	
	|	*node*	
node	→	*node_address features* \n	
node_address	→	*id*	processing node address
features	→	*id , features*	set of features of processing nodes
	|	*id*	
id	→	*Haskell function identifier*	

Figure 7: Syntax for declaring the features of processing nodes inside the file *nodeid*

Syntax			Comments
allocs	→	*alloc allocs*	
	|	*alloc*	
alloc	→	**alloc** (*features*) *processes* \n	
features	→	*id , features*	set of choose features
	|	*id*	
processes	→	*id , processes*	set of processes to be mapped into processing node
	|	*id*	
id	→	*Haskell function identifier*	

Figure 8: Syntax to declare the processing mapping

define the mapping scheme. Its syntax is presented in Figure 8. For instance, consider that a programmer decides to classify machines of a network according to the following parameters:

- performance: *fast* or *slow*;
- memory capacity: *thin* or *wide*;
- physical position: *local* (inside of the processing centre) or *alien* (outside of the processing center);

Using these parameters a programmer could, for example, define that processes with little memory demand but with heavy performance restrictions will run on a *fast* and *thin* machine. The programmer could also define that coarse grained processes ought to be mapped onto an *alien* machine.

Compiler can detect the following errors in **alloc** declarations:

1. mapping into a machine with features that does not belong to the execution environment;
2. the number of machines required exceed the number of existing processors in the execution environment.

4 Translating HCL into Petri Nets

This section describes the mechanism used to translate a program written in *Haskell Configuration Language* into Petri nets. The translation is useful to analyse the formal properties of the communication structure of $Haskell_{\#}$ applications. The behaviour of individual processes are not considered here. In addition to this, we assume that the required resources exist. The process mapping does not take part in the translation scheme, because the absence of resources is detected during compilation (see Section 3.4).

4.1 Functional processes

At the abstraction level HCL programmers work, the internal behaviour of individual processes are not relevant. Process interfaces (input and output *ports*) are the important elements here. Therefore, processes are represented through their input and output *ports*.

Due to the absence of process behaviour in the Petri net model, the time consumed in process execution is null. Thus, after receiving the last message through its input ports, a process gets ready to send its first message through an output port.

The leftmost ports of an **input/output** declaration have higher priority over rightmost ports. There is no dependency between ports in a nondeterministic choice and they get ready to receive simultaneously. Together, these ports can depend on events related to ports in other processes. We call *dependency graph* the directed graph in which nodes are process ports and vertices represent the dependency between them. Ports connected to the vertices target depend on the ports connected to its source (Figure 9). The ports of processes are

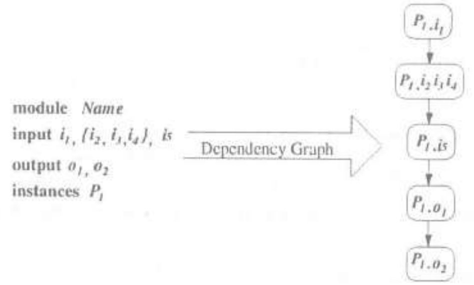


Figure 9: Generation of the dependency graph of a functional process

modelled as *places* in Petri nets. In its turn, ports in a nondeterministic choice are translated into an individual place. It indicates that they will be ready to receive simultaneously. Places are labelled according to the following rule: *process_name.port_name*. Place labels are of practical use and have no formal meaning.

Figure 10 presents the compilation scheme for translating $Haskell_\#$ processes into Petri nets.

4.2 Communication Channels

$Haskell_\#$ communication channels are point-to-point, undirectional and synchronous. These features should be observed in the generated Petri net. However, differently from $Haskell_\#$, channels in the resulting Petri net model have no type. Two reasons can justify this decision:

1. for communication modelling and analysis using Petri nets the values exchanged (or their types) are not important, only the flow of data is taken into account.
2. Due to the static type checking performed by $Haskell_\#$ compiler, HCL are considered to be type correct.

Synchronous communication will be modelled in Petri nets as a transition in which the input arcs are connected to places representing the input and the output ports of the communication channel. In their turn, the output arcs of this transition are connected to the ports that follows it in the *dependency graph* of the two communicating processes. In *repetitive* applications, after sending its last message, processes return to the initial state and a new iteration is triggered. On the other hand, all processes in *nonrepetitive* applications reach a termination state after sending their last message.

Figure 11 presents the compiling scheme for translating communication channels into Petri nets for a *nonrepetitive* application. Here, the communication between ports $P_1.i_1$ and $P_2.o_1$ are deterministic, whereas ports $P_1.i_2$ and $P_1.i_3$ are waiting in a nondeterministic choice for a message from port $P_2.o_2$ and $P_3.o_1$, respectively. Since the application is *nonrepetitive*, all its

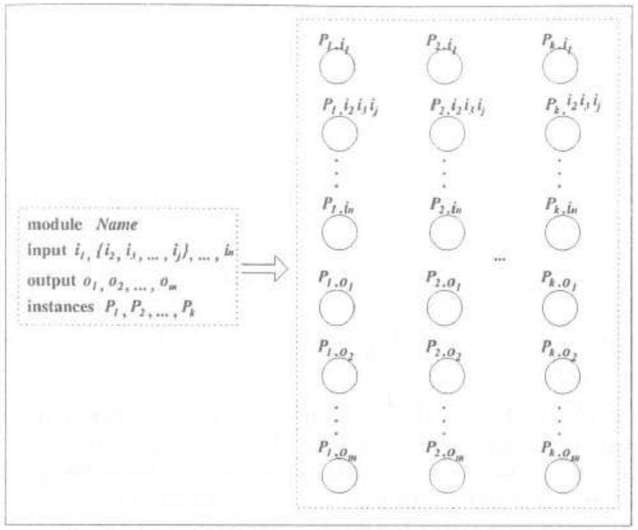


Figure 10: Translation of functional process into Petri nets

processes have a termination state defined by places labelled according to the following rule: *process_name.final*.

Figure 12 presents the translation of communication channels into Petri nets in *repetitive* applications. Note that final states are replaced by ready to restart states. It is indicated by return places labelled according to the following rule: *process_name.return*. Restart action is represented by a transition. Its input place is the return place and its output place is represented by the root node of *dependency graph*.

We emphasize that labels attributed to places have no formal meaning, being only of practical use.

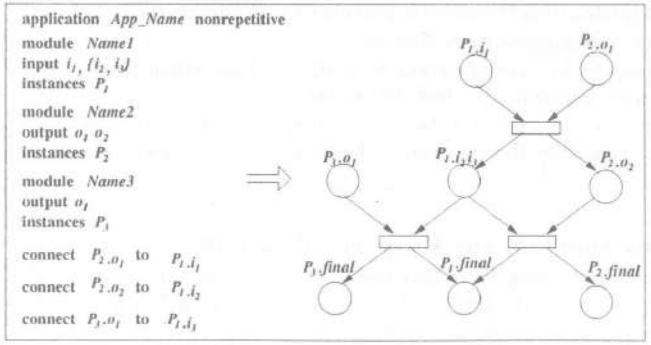
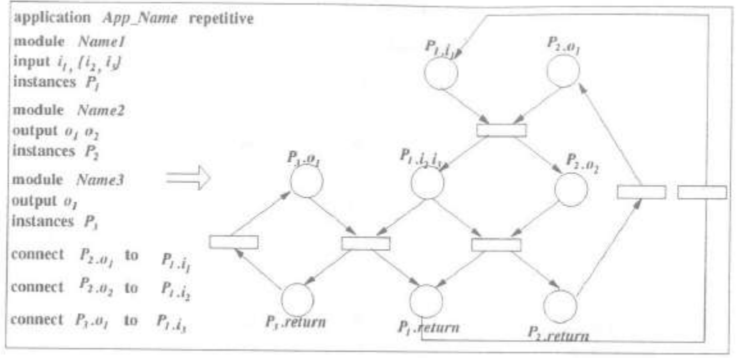


Figure 11: Translation of communication channels of a *nonrepetitive* application into Petri nets



```
application App_Name repetitive
module Name1
input i1, [i2, i3]
instances P1
module Name2
output o1 o2
instances P2
module Name3
output o1
instances P3
connect P2.o1 to P1.i1
connect P2.o2 to P1.i2
connect P3.o1 to P1.i3
```

Figure 12: Translation of communication channels of a *repetitive* application into Petri nets

4.2.1 Initial State

The initial state of a *Haskell*$_\#$ application is given by the union of the initial states of all its processes. **start** declarations (see Section 3.1) define the processes which have autonomy to start executing. Since, at this abstraction level, processes consume no time, processes in scope of a **start** declaration will be initialized in one of the following states:

1. processes with output ports will be initialized in state *ready_to_send_first_message*;

2. processes without output ports in a *repetitive* application will be initialized in state *ready_to_reinitialise*, represented by the *return* place;

3. finally, processes without output ports in a *nonrepetitive* application will be initialized in state *finalized*, represented bu the *process_name.final* place.

Other processes should wait for the input messages in order to start executing. As a result, they are initialized in state *ready for receive first message* represented by the root node of *dependency graph*.

The state of a process is modelled in Petri nets by means of a token inside the places used to represent process ports and its *final* state (in *nonrepetitive* applications) or its *ready to reinitialized* state (in *repetitive* applications). In a given process, only ports in a nondeterministic choice can be simultaneously ready for use.

Figure 13 (a) describes the compiling scheme for translating the initial state for the possible type of processes in a *repetitive* application:

- P_1 belongs to the scope of a **start** declaration and has output ports;
- P_2 does not belong to any **start** declaration;
- P_3 belongs to the scope of a **start** declaration and has no output port;

Figure 13 (b) describes the compiling scheme for the three cases above in a *nonrepetitive* application.

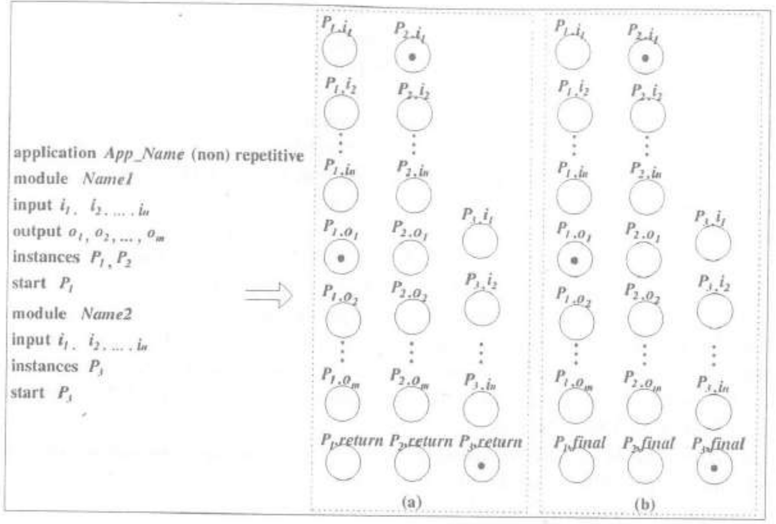
5 An Environment for Property Analysis

After studing existing Petri nets tools we decided to use the *Integrated Net Analyzer* (INA)[35], developed by the group of Prof. Peter H. Starke in *Humboldt-Universität* of Berlin. This choice was motivated by the following reasons:

1. automatic generation of Petri nets for the INA format is simple;

2. a large number of properties of Petri nets can be analyzed by INA;

3.a number of reduction rules, allows to transform a given Petri net model into a simple one, keeping



```
application App_Name (non) repetitive
module Name1
input i1, i2, ..., in
output o1, o2, ..., om
instances P1, P2
start P1
module Name2
input i1, i2, ..., in
instances P3
start P3
```

Figure 13: Translation of process initial state into Petri nets

the properties of the original model;

4. the behaviour of a Petri net can be simulated through INA.

In order to develop a compiler, based on the strategies described in Section 4, which translates an HCL description into a Petri net "code" readable by INA, we used Lex[23] and Yacc[17]. Users of our *Haskell*$_\#$ programming environment can decide either to simulate, or to apply reduction rules and/or analyze formal properties of resulting model Petri net model through the interactive environment of INA.

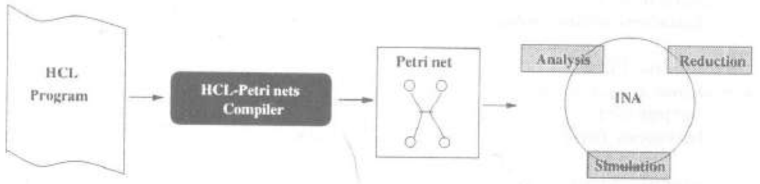


Figure 14: Properties analysis environment

6 Example

This section presents the *Haskell*$_\#$ analysis of a small HCL program. After describing the HCL program, its translation into Petri net is detailed. Eventually, the resulting Petri net model will be analyzed through INA.

The example system is composed by four functional processes. Two signal generator processes ($signal_1$ and $signal_2$), one filter process ($filter$) and a process to print the filtered signal ($printer$). $printer$ has two input ports, i_1 and i_2, which wait concurrently for a signal sent by processes $signal_1$ and $signal_2$, respectively. If both signals arrive together, the choice is performed nondeterministically. Independently of the choice performed, the $filter$ process applies a filter in the received signal and sends the resulting channel for the process $printer$. Eventually, the process $printer$ prints the received value.

Applications can be defined as: *repetitive* or *nonrepetitive*. Both alternatives will be studied. Figure 15 presents the code for both HCL programs.

Applying the compilation rules to functional process (Section 4.1) are generated the Petri nets of Figures 16 (a) and (b) for the *repetitive* and *nonrepetitive* applications, respectively. Through the rules for translating communication channels (Section 4.2) we get the Petri nets of Figures 16 (c) and (d) for the *repetitive* and *nonrepetitive* applications, respectively. Finally, appling compiling rules for process initialization (Section 4.2.1), the Petri nets of Figures 16 (d) and (e) are generated from, respectively, the *repetitive* and *nonrepetitive* applications. Figure 17 presents the list of properties calculated by INA for both applications.

The interpretation of the properties calculated by INA (Figure 17) must be done adequately but it is beyond the scope of this paper. The property on Figure 17 shows that the *repetitive* application is *live* and *reversible*, whereas the *nonrepetitive* application does not possesses these properties. Based on this information the programmer could decide to define the application as *repetitive*, because it avoids that the system stop after processing a signal sent from $signal_1$ or $signal_2$. In addition, it guarantees that the system will execute the following sequence of steps an unlimited number of times:

1. process $filter$ chooses, nondeterministically, a signal sent from processes $signal_1$ and $signal_2$;
2. the received signal is filtered by the $filter$ process;
3. $filter$ process send the filtered signal to the $printer$ process;
4. $printer$ process prints the filtered signal;
5. return to step 1.

The example described in this section showed how the properties analyzed from the Petri net model can help programmer to reason about the system under development in order to predict and to correct possible problems in the system implementation.

```
application SignalProcess (non)repetitive

module Signal
output o_1::t
instances signal_1, signal_2

module Filter
input i_1, i_2::t
output o_1::t
instances filter

module Printer
input i_1::t
instances printer

alloc thin signal_1
alloc thin signal_2
alloc thin printer
alloc wide filter

start signal_1
start signal_2

connect signal_1.o_1 to filter.i_1
connect signal_2.o_1 to filter.i_2
connect filter.o_1 to printer.i_1
```

Figure 15: A small HCL program

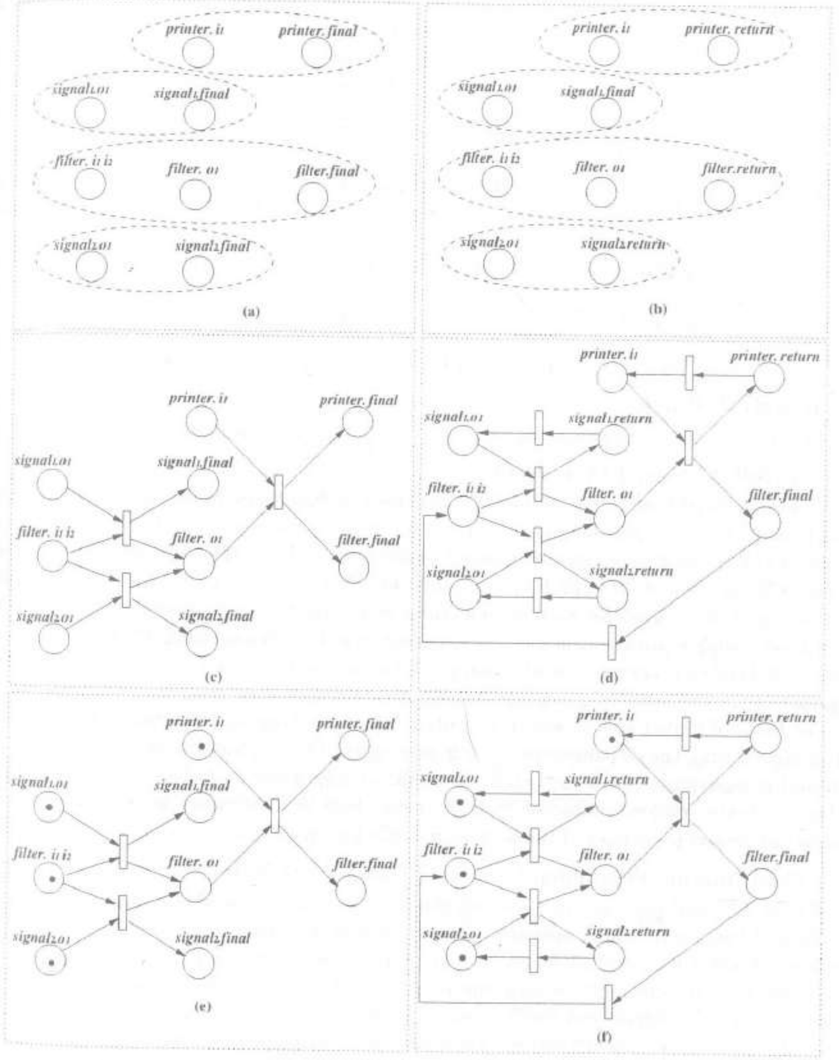


Figure 16: Translating HCL program of Figure 15 into Petri nets

Properties	repetitive	nonrepetitive
ordinay	yes	yes
pure	yes	yes
conservative	yes	no
static conflicts free	no	no
dynamic conflicts free	no	no
maked graph	no	no
state machine	no	no
free choice	no	no
extended free choice	no	no
satisfies the deadlock-trap property	yes	no
covered by place invariant	yes	yes
covered by transition invariant	yes	no
limited	yes	yes
structurally bounded	yes	yes
reversible	yes	no
dead transition in the initial state	no	no
live	yes	no
live e *safe*	yes	no

Figure 17: Properties of Petri net of Figures 16 (c) e (e)

7 Related Works

In this section we present an account of two works related to ours.

7.1 Haskell-Coloured Petri Nets

Coloured Petri Nets (CPN) 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. Claus Reinke presented in [33] a variant of CPN named *Haskell-Coloured Petri Nets* (HCPN) and show that they have a simple mapping onto Haskell. HCPN 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.

The focus of Reinke's work was to stimulate functional language programmers to use Petri net models during the development of their programs. The relationship between communication and computation is not exploited. It would be important to isolate these two elements and get a better understanding of each of them. Besides that, there is no concern on the analysis of formal properties of the system modelled by HCPN.

7.2 Coordinating Functional Processes Using Petri Nets

Claus Abmann proposes in [1] a coordination language, named *K2*, based on a variant of coloured Petri nets, which primarily defines process systems with deterministic behaviour, but also allows for controlled forms of nondeterminism. According to Abmann, net-based specification of cooperating processes combine the advantages of an underlying calculus (which facilitates formal analysis and verification of a sequential system behaviour) with graphical representations (which clearly expose structural dependencies amongst system components. There is an interactively controlled simulation mode that allows for stepwise execution and the possibility to display intermediate states of token distributions. The *K2* environment includes compilers to convert specifications into executable code.

A functional language named *kiR*[22] is used to specify processes behaviour. In his work, Abmann does not clarify the reasons for choose *KiR*. On the other hand, in $Haskell_{\#}$, we

extend Haskell, a *de facto* standard pure functional language for describing process behaviour.

In *K2*, places are used to hold tokens (values) to be exchanged between processes. The communication structure is represented by directed arcs. Processes behaviour are modelled by Petri net transitions. Each transition represents a functional program, in which the behaviour is associated with the Petri net, modifying its behaviour. Although Abmann asserts that his approach yields a process hierarchy, this hierarchy is weaker than the one offered by $Haskell_{\#}$.

K2 establishes a finite synchronous distance[2] between communicating processes. A process is enabled if there exist enough tokens in its input queues and enough free space in its output queues. This mechanism makes *K2* communication asynchronous. Similarly to Occam[15], $Haskell_{\#}$ adopts a synchronous communication model.

8 Conclusions

This work presented an environment for analyzing formal properties of the communication structure of $Haskell_{\#}$ programs. This structure is defined by $Haskell_{\#}$ Coordination Language (HCL), a language also used for task-to-processor allocation. This environment contains a compiler that translates an HCL program into Petri nets as specifyied by INA[35], a tool capable of simulating and automatically analyzing several formal properties of systems.

We believe that this hierarchical approach to concurrent system development brings the ability of reasoning about programs and better chances of proving their correctness.

References

[1] C. Abmann. Coordinating Functional Processes Using Petri Nets. *Implementation of Functional Languages, Springer-Verlag, LNCS 1268*, pages 162–183, September 1997.

[2] G. Agha. Actors: A Model for Concurrent Computation in Distributed Systems. *MIT Press, Cambridge, Massachussets*, 1986.

[3] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. *International Conference on Coordination Models and Language, Springer-Verlag, LNCS 1061*, pages 34–56, April 1996.

[4] W. H. Burge. Recursive Programming Techniques. *Addison-Wesley Publishers Ltd.*, 1975.

[5] R. M. Burstall, D. B. McQueen, and D. T. Sannella. Hope. *Technical Report CSR-62-80, Edinburgh University*, 1980.

[6] F.W. Burton. Functional Programming for Concurrent and Distributed Computing. *Computer Journal*, 30(5):437–450, 1987.

[7] K. Didrich, Fett A., C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. *J. Gutknecht, editor, Programming Languages and System Architectures, Zurich, Switzerland - Springer-Verlag LNCS 782*, pages 228–244, 1994.

[8] G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. *MIT Press, Cambridge*, 1994.

[9] GHC Team. The Glasgow Haskell Compiler User's Guide, Version 4.01. *http://www.dcs.gla.ac.uk/fp/software/ghc/4.01/users_guide/users_guide.html*, 1998.

[10] K. Hammond and Michaelson G. Research Directions in Parallel Functional Programming. *Springer-Verlag*, 1999.

[11] C. A. R. Hoare. Communicating Sequential Processes. *Prentice-Hall, C.A.R. Hoare Series Editor*, 1985.

[12] P. Hudak. Serial Combinators: "Optimal" Grains of Parallelism. *FPCA'85*, pages 382–399, September 1985.

[13] P. Hudak. Para-Functional Programming in Haskell. *Parallel Functional Languages and Compilers, B. K. Szymanski, Ed. ACM Press, New York*, pages 159–196, 1991.

[2] A synchronous distance between transitions *A* and *B* defines the number of occurrences of *A* before *B* is allowed to take place

[14] P. Hudak, S. P. L. Jones, and P. L. Wadler. Report on Programming Language Haskell: a Non-Strict, Purely Functional Languages. *Special Issue of SIGPLAN Notices*, 16(5), May 1992.

[15] Inmos. Occam 2 Reference Manual. *Prentice-Hall, C.A.R. Hoare Series Editor*, 1988.

[16] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, (3):184–195, 1960.

[17] S. C. Johnson. YACC - Yet Another Compiler Compiler. *Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey*, 1975.

[18] S. P. L. Jones. The Implementation of Functional Programming Languages. *Prentice-Hall, C.A.R. Hoare Series Editor*, 1987.

[19] S. P. L. Jones, C. Clack, and J. Salkild. GRIP - A High-Performance Architecture for Parallel Graph Reduction. *FPCA'87: Conference on Functional Programming Languages and Computer Architecture - Springer-Verlag LNCS 274*, pages 98–112, 1987.

[20] S. P. L. Jones, A. Gordon, and S. Finne. Concurrent Haskell. *POPL'96 - Symposium on Principles of Programming Languages, ACM Press*, pages 295–308, January 1996.

[21] O. Kaser, C.R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Equals - A Fast Parallel Implementation of a Lazy Language. *Journal of Functional Programming*, 7(2):183–217, March 1997.

[22] W. E. Kluge. A User Guide for the Reduction System π-RED. *Technical Report 9409, Institut für Informatik und Praktische Mathematik, Universität Kiel*, 1994.

[23] M. E. Lesk. LEX - A Lexical Analyzer Generator. *Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, New Jersey*, 1975.

[24] R. M. F. Lima, F. H. Carvalho Jr., and R. D. Lins. *Haskell*$_{\#}$: A Message Passing Extension to Haskell. *CLAPF'99 - 3rd Latin American Conference on Functional Programming* , pages 93–108, March 1999.

[25] R. D. Lins. Functional Programming and Parallel Processing. *2nd International Conference on Vector and Parallel Processing - VECPAR'96 - LNCS 1215 Springer-Verlag*, pages 429–457, September 1996.

[26] R. Milner, M. Tofte, and R. Haper. The Definition of Standard ML. *MIT Press, Cambridge, Massachussets*, 1989.

[27] T. Murata. Petri Nets: Properties Analysis and Applications. *Proceedings of IEEE*, 77(4):541–580, April 1989.

[28] R. S. Nikhil. Id (version 90.1) reference manual. *Technical Report CSG Memo 284-2, Lab. for Computer Science, MIT*, July 1991.

[29] R. S. Nikhil, Arvind V., and J. Hicks. pH Language Proposal (Preliminary). *Electronic communication*, September 1993.

[30] J. L. Peterson. Petri Net Theory and the Modeling of Systems. *Prentice-Hall, Englewood, N. J.*, 1981.

[31] C. A. Petri. Kommunikation mit Automaten. *Technical Report RADC-TR-65-377, Griffiths Air Force Base, New York*, 1(1), 1966.

[32] M. J. Plasmeijer and M. van Eekelen. Functional Programming and Parallel Graph Rewriting. *Addison-Wesley Publishers Ltd.*, 1993.

[33] C. Reinke. Haskell-Colored Petri Nets. *Implementation of Functional Languages*, 1999.

[34] W. Reisig. Petri Nets. *Springer-Verlag, Berlin*, 1985.

[35] S. Roch and P. Starke. Manual: Integrated Net Analyzer Version 2.2. *Humboldt-Universität zu Berlin, Institut für Informatik, Lehrstuhl für Automaten- und Systemtheorie*, 1999.

[36] P. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. P. L. Jones. GUM: A Portable Parallel Implementation of Haskell. *PLDI'96 - Programming Languages Design and Implementation*, pages 79-88, 1996.

[37] D. Turner. Functional Programming as executable Specifications. *Phil. Transactions of the Royal Society of London 312*, pages 363–388, 1984.