# Deadlock Detection in a Distributed Implementation of a Visualization System for Medical Measurement Signals

G. Lindner*
Physikalisch-Technische Bundesanstalt

M. Heiner, T. Kobienia†
Technical University of Cottbus

## ABSTRACT

In this paper we apply the methods proposed in [5], [7] for detecting anomalies in existing executable INMOS C [1] code for a parallel program using static and dynamic analyses of Petri nets. Our target architecture on which the application program is running is a transputer network. The algorithmic translation of the INMOS C program into hierarchical place/transition nets which preserves both control and message flow is performed by a Petri net generator. Every implemented process is converted into an equivalent net. After this, the net parts generated are visualized and linked, and supplements are added, using a powerful graphical Petri net editor. Hierarchical methods are furnished, e.g. function calls are levelled down to subnets, and communication interface objects are highlighted. Every net object in the graphical representation is referenced with its counterpart in the source code so that error localization is possible. In the next processing step, the output capabilities of the Petri net editor are used to produce several input files for analyzing tools. Finally, the process nets investigated separately are brought together by merging the communication interfaces into one whole top level net. Here, a renewed analysis is performed.

**Index Terms: Transputer network, visualization system, processes, Petri net generator, reachability graph, analysis.**

## 1. INTRODUCTION

The program under investigation is used at the PTB laboratories at the "*Hospital Benjamin Franklin*" in Berlin to visualize medical data (e.g. ECG, EEG, MCG) on-line in different selective graphics modes [13]. Our program was developed to read continuously data from the multichannel acquisition system under stringent real-time conditions. A great number of processing steps is performed (e.g. calculating of mean values, filtering and interpolation algorithm). After preprocessing on processor pipes, the computed data are displayed on two big monitors. One monitor displays an overview of all channels and a status window, while the other shows a reference channel and a special contour plot. Details of the hybrid target architecture used for the program are shown in figure 1. The processing nodes *processing0* and *processing1* are PowerPC601[1] high performance nodes, and all other nodes are T805.

Owing to their links, transputers may be connected into a great number of configurations, depending on the given application. They form a distributed memory architecture. The abstract programming model which the transputers support is the Communicating Sequential Process (CSP) model, based on the idea of independent parallel processes communicating through channels. Channels are one-way, point-to-point communication paths allowing processes to exchange data and to synchronize their activities. Every process may be formed by any number of parallel processes, so that the entire software system can be described by a hierarchy of communicating parallel processes. The communication between processes is synchronized. When data are exchanged between two processes, the output process does not continue before the input process has finished its communication task and vice versa. The software for the visualization system is written in the INMOS C language [1], which has been devel-

* Institut Berlin, D-10587 Berlin, Germany, glindner@chbrb.berlin.ptb.de

† Department of Computer Science, P.O.Box 101344, D-03013 Cottbus, Germany, mh@informatik.tu-cottbus.de, tk@informatik.tu-cottbus.de

[1]PowerPC601 is a trademark of International Business Machines.

oped as an extension of ANSI C. A great number of library functions is provided for channel-based input and output.

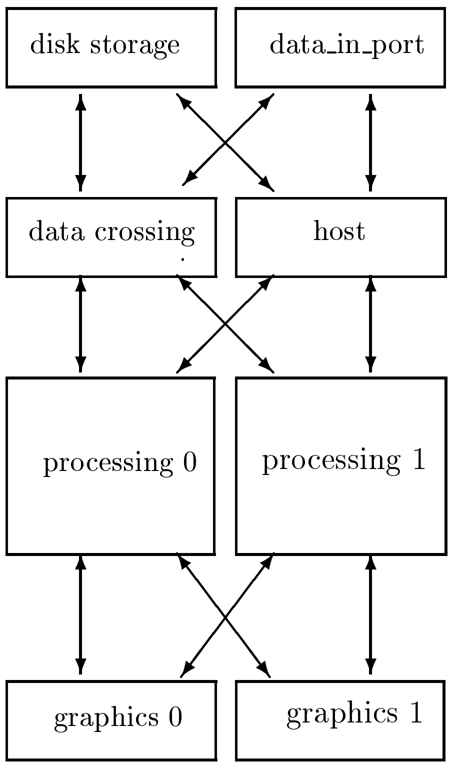


Figure 1: Processor node architecture of the application program

## 2. UNPREDICTABLE PROGRAM BEHAVIOUR

Besides the static configuration (e.g. number of active measurement channels, sampling frequency from data acquisition system, channel distribution in the windows available etc.), the user has the opportunity to control dynamically the state of the system by interactive keyboard commands. The control sequences must be routed throught the network against the data stream and influence the homogeneous data transfer between the different processing nodes. Non-reproducible system states can occur in a stochastic manner, leading sometimes to communication deadlocks, which are based on the message passing programming model along synchronized channels. Then, the whole system is stopped irreversibly. The program behaviour often critically depends on potentially unpredictable timing of the components caused by the keyboard commands.

## 3. NET-BASED QUALITATIVE ANALYSIS

Net-based qualitative analysis is a well-known approach. It can be used both as a pedagogical aid and as a tool for debugging and investigating the transputer application [9]. In the following, we apply the methods proposed in [5], [7] for detecting deadlocks in our application program. The source code was analyzed with a view to detecting errors in the software available, to correct it and to increase its reliability. In the following, the main components—*generator, editor and analyzer*—are shortly discussed.

### 3.1 Generator component

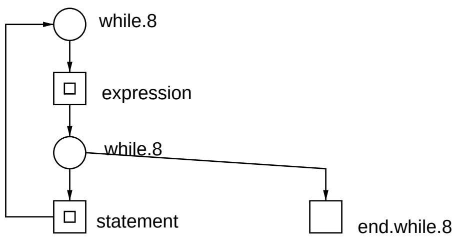
In the first instance, the program is translated into an appropriate hierarchical place/transition net [11]. The generation basically assembles general Petri net components for all important language means in accordance with the syntax tree. To generate Petri net components from the INMOS C source code, the generator uses the following simplified grammar:

```
process ::= function.definition.
  function.definition ::=
  function.header compound.statement.
  compound.statement ::= '{'statement.list'}'.
statement.list ::= statement.list ';' statement
  | statement.
statement ::= labeled.statement
  | if.statement
  | switch.statement
  | iteration.statement
  | jump.statement
  | compound.statement.
iteration.statement ::=
  | while.statement
  | for.statement
  | do.statement.
jump.statement ::=
  | break.statement
  | continue.statement
  | goto.statement
  | return.statement.
labeled.statement ::= label@':'statement.
```

Semantic actions are introduced for every grammatical rule. These actions generate the Petri net components for the INMOS C statements which can alter the control flow (such as if, do, while and switch statement) or which constitute a synchronuous communication. The following was defined to allow all Petri net components to be uniformly processed:

1. every Petri net component begins with a place,
2. every Petri net component ends with a transition.

All node names in the Petri net model are controlled by the appropriate Petri net components [3], while the suffix numbers in the node names refer to the source text line numbers. An example of a "*while-construct*" [8] is shown in figure 2. Besides this basic functionality, further information is produced, allowing the generated nets to be automatically laid out afterwards, and the program's structural complexity *NAP*[2] to be assessed.



*Grammar*:
while.statement ::=
while'('expression')'statement
*Program's structural complexity*:
NAP(while.statement) :=
NAP(statement) $*$ NAP(expression) + 1

Figure 2: While construct

**Synchronization concept:** The extended library functions of the INMOS C language (e.g. *ChanIn, ChanInInt, ChanOut, ChanOutInt, ProcSkipAlt*) are of particular importance for the generation of synchronization skeletons. The Petri net generator in use [8] produce net representations of the reduced or nonreduced control structure of a given sequential process. In the reduced operation mode, the Petri net model is generated for all sequential program parts, which include process interaction primitives mentioned above[3]. The basic construct of synchronous communication is shown in figure 3. In general, every communication in any INMOS C program is represented by two unique places in the corresponding net. The places may be thought of as communication channels between potential "senders" and "receivers". The place $p_{\text{send.0}}$ is referred to as a sending place. It holds a token if the sending process fires the transition *ChanOut1* showing that the sender starts waiting for synchronous communication. Firing the transition *ChanIn* represents that the control of *process2* passes the receive statement. After the receive statement ends, place $p_{\text{ack.1}}$ enables the transition *ChanOut2*. This means that the control of *process1* may exit from the suspended state. These places generated by the translation of the extended library functions are referred to as *communication places*. They are connected with *communication arcs*. A *communication arc* is not considered to be part of any process subnet. All noncommunication arcs are refered to as *sequential arcs*.

**Properties of the generated nets:** If we disregard any communication between processes, a distributed program can be modeled by a set of state machines, one state machine per process. A state machine is a Petri net such that each transition t has exactly one input place and exactly one output place, i.e., $|{}^\bullet t| = |t^\bullet| = 1$ for all $t \in T$[4]. Our distributed programs are characterized by many communications between the processes. The process subnets are merged on its communication interfaces to form the top level net. A top level net is composed of strongly connected Petri net process subnets possibly interconnected with communication arcs and places making the total Petri net strongly connected[5]. Additionally, it is worth noting that the whole net is 1-bounded (safe) per construction. The initial marking $M_0$ represents the situation in which a program modeled by the net is ready to start execution.

## 3.2 Editor component

The graphical Petri net editor [4] is used to visualize the net parts generated (and to add any supplements). Figure 4 shows a simple example of one such process. Hierarchical methods are used, e.g. function calls are levelled down to subnets. The next processing step brings the separate process nets together (linking) merging them at the

[2]NAP: **N**umber of **A**cyclic **P**aths

[3]That means, that any sequential program parts, which do not include process interaction primitives, are reduced to one transition.

[4]Here, ${}^\bullet t$ ($t^\bullet$) denotes the set of input (output) places of transition $t$.

[5]A net is strongly connected if there is a directed path from any node (place, transition) to any other node.

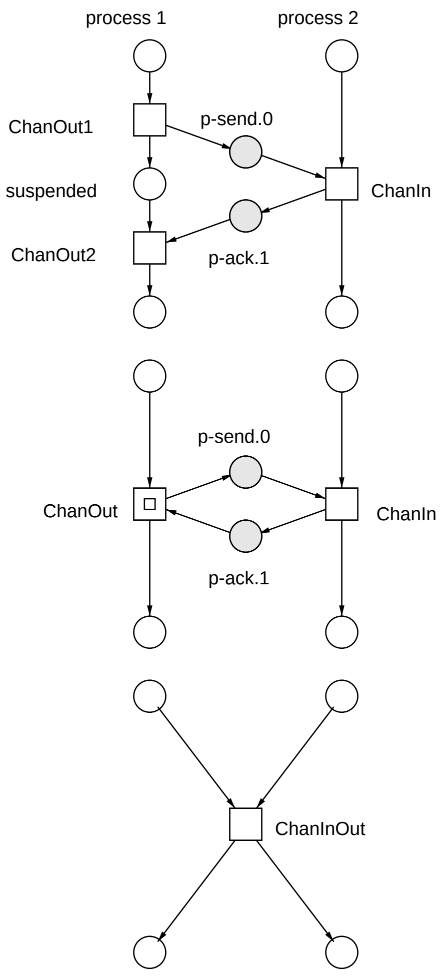


Figure 3: Synchronous communication

communication interfaces to form a whole top level net. Figure 5 shows the half part of our symmetrical composed top level net. The transition *send1* in figure 5 is a coarse transition and represents the whole sequential structure of the *send1* process.

## 3.3 Analyzing component

Finally, the output capabilities of the Petri net editor produce an input file for the analyzing tool *INA* [12]. Significant properties, such as liveness, absence of deadlocks, safety and boundedness are evaluated. There is no commonly accepted definition of a deadlock in a distributed environment [10]. In our context, the term *deadlock* describes a system state, where no transition is enabled.

The analysis is done bottom-up in a step-wise manner. At first, all processes are analyzed separately.

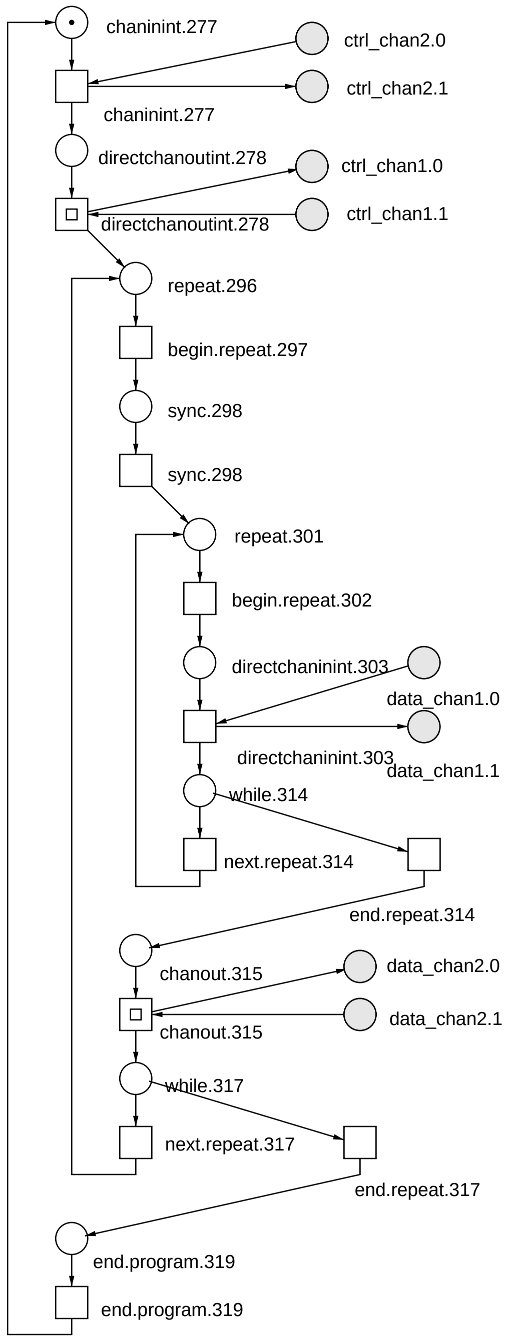


Figure 4: Reduced send1 process net

Afterwards, the top level net is analyzed step-wise starting with a smaller subnet (2-3 processes) which is extended step by step by further processes until the complete process system net has been composed and analyzed.

## 4. SYNCHRONIZATION SKELETON FROM THE APPLICATION PROGRAM

To investigate all processes *datain*, *send0*, *send1*, *processing0*, *processing1*, *graphics*, and *host*, the processing cycle

1. generation of the place/transition net
2. postprocessing in the editing environment
3. use of the analyzing tool

is iteratively used.

| |
|---|
| token type: black (for place/transition nets) |
| time option: no times |
| firing rule: normal |
| priorities : not to be used |
| strategy : single transitions |
| line length: 80 |

Table 1: Analyzing tool presettings

| process net name | generated states | no. of net objects (p , t) | no. of source lines |
|---|---|---|---|
| *datain* | 14 | 14 , 19 | 170 |
| *send0* | 20 | 20 , 20 | 96 |
| *send1* | 12 | 12 , 14 | 64 |
| *processing0* | 22 | 22 , 18 | 204 |
| *processing1* | 31 | 31 , 27 | 239 |
| *graphics0* | 131 | 131 , 151 | 733 |
| *graphics1* | 91 | 91 , 112 | 581 |
| *host* | 56 | 56 , 62 | 455 |

Table 2: Analyzing results of the process subnets

In error free case, the analysis tool [12] presettings as shown in table 1 lead to the following properties of one isolated process: "The net is not statically conflict-free, ordinary, homogenous, bounded, a state machine, has no dead transitions at the initial marking, has no dead reachable states, is live and safe and reversible (resetable)". Here, we are able to use the exhaustive search by constructing the complete reachability tree (state graph). Such analysis has an exponential time complexity in terms of the number of concurrent processes. In table 2 details from the analyzing sessions are shown. When the analysis of the individual nets has proved that the nets are free from anomalies, these are merged at their communication interfaces and the analysis of the overall net is carried out.

To handle the complex state space at the top level, we use the reduced reachability graph construction by the stubborn set method which allows us to prove at least the freedom of dead states. Moreover, the 1-boundedness can be shown efficiently by proving that the net is covered by place-invariants. More complicated properties of the total net could not be proved by classical Petri net analysis techniques because of the huge size of the complete reachability graph. Therefore[6] we started an additional analysis step based on (different versions of) temporal logics by the help of the analysis tools *PEP* [2] and *PROD* [14].

## 5. CONCLUSIONS

In this paper, we have described an approach to increase the reliability of existing transputer software. First, an INMOS C program is translated into a Petri net, which properly models the communication patterns and control flow of the source program. Then, the generated net is analyzed using both structural and dynamic Petri net analysis techniques. Ongoing investigations deal with improved analysis techniques based on temporal logics.

## REFERENCES

[1] *ANSI C Language and Libraray Reference Manual*, INMOS document number: 72 tds 37401 edition.

[2] E. Best and B. Grahlmann. *PEP-Programming environment based on Petri nets, documentation and user guide.* University Hildesheim, Dep. of CS, November 1995.

[3] G. Czichy. Implementation of a petri net generator for INMOS C programs. Technical report, GMD/FIRST, February 1992.

[4] G. Czichy. Design and implementation of an graphical editor for hierarchical petri net mod-

[6]encouraged by the positive results of another case study published in [6]

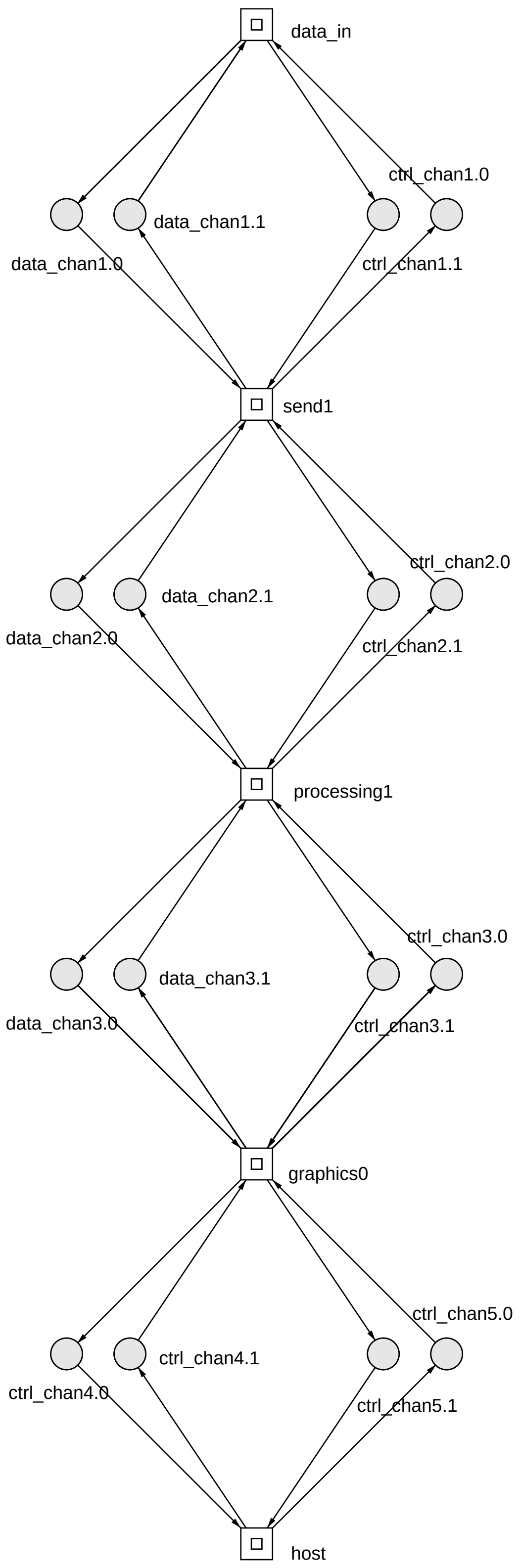


Figure 5: The half of the top level net

els. Master's thesis, Technical University Dresden and GMD/FIRST, June 1993.

[5] M. Heiner. Petri net based software validations, prospects and limitations. Technical report, ICSI-TR-92-022, Berkeley/CA, March 1992.

[6] M. Heiner and P. Deussen. Petri net based qualitative analysis - a case study. Technical report, Technical University Cottbus, December 1995.

[7] M. Heiner, G. Ventre, and D. Wikarski. A petri net based methodology to integrate qualitative and quantitative analysis. In *Information and Software technology*, volume 36, pages 435–441, 1994.

[8] T. Kobienia. Extended petri net generator for INMOS C programs. Technical report, Technical University Cottbus, February 1996.

[9] G. Lindner. The application of formal description methods to support the program development of a real-time visualization system for medical measurement signals based on transputers. In *Design of Complex Automatisation Systems*, pages 509–519. Technical University Braunschweig, June 1995.

[10] T. Murata, B. Shenker, and S. M. Shatz. Detecting of ada static deadlocks using petri net invariants. In *IEEE Transactions on Software Engineering*, volume 15, pages 314–325. IEEE, March 1989.

[11] H. P. Starke. *Analyse von Petri-Netz-Modellen.* B. G: Teubner, 1990.

[12] H. P. Starke. Ina - integrated net analyzer, 1992.

[13] D. Stollfuss and G. Lindner. A real-time visualization system for medical measurement signals based on transputers. *Physikalisch-Technische Bundesanstalt Annual Report*, pages 281–282, 1993.

[14] K. Varpaaniemi, J.Halme, K.Hiekkanen, and T.Pyssysalo. Prod reference manual. Technical Report 13, Helsinki University of Technology, Digital Systems Lab., August 1995. Series B.