# Using Petri Nets for Data Dependency Analysis

Fred Cruz Filho, Paulo Maciel and Edna Barros
Centro de Informática – Universidade Federal de Pernambuco
Recife, PE, Brazil, 50740-540

## Abstract

Due to design constraints, many digital systems are implemented as mixed hardware and software components. An informal choice of where implementing a part of the system — either in software or in hardware — can produce incorrect or unsatisfactory results. Co-design methodologies have been developed to aid the development of such systems. The partitioning phase is one of the tasks carried out by a co-design methodology. It divides the original system into components taking in account software or hardware implementation. This work presents a model to carry out data dependency analysis, which is one of the aspects considered for the partitioning algorithm of the PISH co-design methodology.

## 1 Introduction

Due to design constraints, many digital systems are implemented in an heterogeneous architecture composed by hardware and software components [1]. Software components are cheaper, more flexible and portable than hardware components. On the other hand, hardware implementation may yield a better performance. However, application specific circuits are expensive and should be used only when strictly necessary. The growing complexity of such systems makes the choice a difficult task, and an informal solution can lead to incorrect results.

Although such mixed systems have been designed since hardware and software first came into being, there is a lack of CAD tools supporting the development of such heterogeneous systems [2]. The partitioning phase is one of the most important tasks carried out by a co-design methodology. Its main task consists in identifying system's components and choosing between a software or a hardware implementation of each component. In the PISH co-design methodology, data dependency is an important aspect taken in account for the partitioning algorithm.

This work presents a model developed to carry out the data dependency analysis in the PISH methodology. This work is organized as follows. Section 2 depicts an overview of the PISH methodology. Section 3 introduces Petri nets concepts. Sections 4 and 5 show the proposed model as well as the proposed methodology for carrying out the data dependency analysis, respectively. Section 6 presents an example. Finally, section 7 concludes and gives some further works.

## 2 The PISH co-design system

The PISH co-design methodology is being developed at CIn/Universidade Federal de Pernambuco. It uses occam [3] as its specification language. Its main focus is to guarantee that the functionality of the original system is preserved on the partitioned system [4] by the use of a well-defined set of formal transformation rules applied to the specification. The workflow of the PISH methodology has the following phases (Fig. 1):

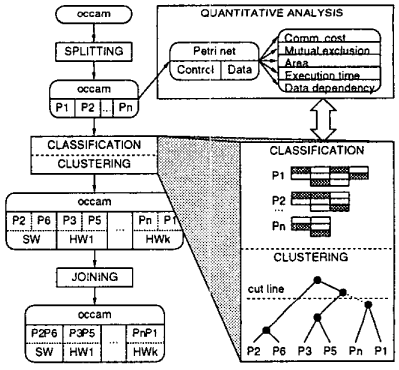


Figure 1: PISH methodology workflow

**Splitting.** The original specification is transformed by exhaustive application of algebraic rules, until all processes reach a *normal form* [5]. The result of this transformation is a parallel composition of ordinary processes, with the inclusion of special

0-7803-6583-6/00/$10.00 © 2000 IEEE

processes called *controllers*. *Controllers* perform communication between a pair of processes, sending the variables which are read and written and receiving the variables which are written.

**Quantitative analysis.** In this phase, a set of metrics such as communication cost [6], mutual exclusion degree [7], area [8], and execution time [9] are estimated. Most of these metrics are computed from the final Petri net model [10, 11] which was obtained from the occam specification.

**Classification.** Interacting with the quantitative analysis toolset, this phase analyzes the processes. According to criteria such as data dependency, parallelism, mutual exclusion and multiplicity [12], a set of implementation alternatives is captured. Data dependency analysis helps to discover situations where a parallel or pipeline implementation should be considered, for improving the performance of the final system. This result can also be used on the clustering phase to keep data-dependent processes together, avoiding synchronization costs. One implementation alternative is chosen for each process, either automatically or manually.

**Clustering.** The processes are grouped in clusters, generating a new configuration of the system. An hierarchical clustering approach is performed according to a metric which takes in account criteria such as similarity among elements and resource sharing. At the end of the clustering phase, a set of partitions is generated and one of them is allocated to a software component, whereas the others remain as hardware components.

**Joining.** The joining phase performs the inverse transformations of the splitting phase. This phase removes the unnecessary controllers of the system. It also tries to perform transformations considering the implementation alternatives chosen by the classification and clustering phases (pipelining, serialization).

occam is a programming language derived from CSP [13], first intended to implement concurrent algorithms. The occam subset considered in this work has the form:

```
P ::= SKIP | STOP
    | x := e | ch?x | ch!e
    | IF ( g1 P1, ..., gn Pn ) | ALT ( g1 P1, ..., gn Pn )
    | WHILE ( g P )
    | SEQ ( P1, P2, ..., Pn ) | PAR ( P1, P2, ..., Pn )
```

SKIP has no effect and always terminates successfully. STOP has no effect, but it never terminates (deadlock). The `x := e`, `ch?x` and `ch!e` are the assigment, input and output operation, respectively. IF and ALT select one of the following processes, based on their guard. If two or more guards are active on the same time, IF selects the left-most process and ALT randomicaly chooses one. WHILE executes the following process until its guard becames false. SEQ executes the following processes in a sequence, starting from the left-most. PAR executes the following processes concurrently.

# 3 Petri nets overview

Petri nets are a family of mathematical formalisms that model concurrent systems by implicit token-flow in a net, providing a sort of methods for qualitative and quantitative analysis.

**Definition 1** Petri net. *Let $P$ be a finite set of places, $T$ be a finite set of transitions and $I, O : P \times T \to \mathbb{N}$ be the input (pre-conditions) and output (post-conditions) matrices, respectively. $N = \langle P, T, I, O \rangle$ is a matrix-based definition for a place/transition net. Furthermore, $I$ and $O$ will be referred as multi-sets, where $p \in I^t(t) \Leftrightarrow I(p,t) > 0$, $p \in O^t(t) \Leftrightarrow O(p,t) > 0$ (and similarly for $I^p(t)$ and $O^p(t)$).*

**Definition 2** Marked Petri net. *Let $N = \langle P, T, I, O \rangle$ be a Petri net. Let $M : P \to \mathbb{N}$ be a marking vector associated to net $N$, where $M(p)$ is the number of tokens on a place $p$. A net $N' = \langle N; M \rangle$ is said a* marked Petri net *if $M$ is a marking vector of $N$. A special marking vector $M_0$ is defined as the* initial marking *of a marked Petri net.*

**Definition 3** Firing rule. *Let $N = \langle P, T, I, O, M \rangle$ be a marked Petri net. A transition $t$ is said* enabled *if $M(p) \geq I(p,t), \forall p \in P$. Firing transition $t$ produces a new marking vector $M'$, such as $M'(p) = M(p) - I(p,t) + O(p,t)$, denoted by $M[t\rangle M'$. A special empty transition $\lambda$ is such that $M[\lambda\rangle M$.*

**Definition 4** Reachable marking. *Let $N = \langle P, T, I, O, M \rangle$ be a marked Petri net and $M''$ be a marking vector. $M''$ is said* reachable *from $M$ if there exists a transition $t$ (possibly empty) and a marking $M'$, such that $M'[t\rangle M''$ and $M'$ is reachable from $M'$.*

**Definition 5** Marking coverability. *Let $N = \langle P, T, I, O \rangle$ be a Petri net, $M$ and $M'$ be marking vectors. $M$ is said* coverable *by $M'$, denoted by $M \leq M'$ (or $M' \geq M$), if $M'(p) \geq M(p), \forall p \in P$.*

Analysis methods for Petri nets properties can be grouped in three classes: reachability graph based methods, state equation based methods and reduction techniques. The reachability graph is dependent on the initial marking and generates all the possible states (markings) of the net. In counterpart, it demands a high computing power and, for certain classes of nets, it is potentially infinite.

State equation methods are based on the solution of simple linear algebraic equations. Although, such techniques may give only necessary, but not sufficient, conditions to infer conclusions about properties of generic nets.

Reduction rules can also be applied to the net for reducing its structure — decreasing the number of intermediary states — without modifying the net semantics. In order to obtain better results, it is fair to suggest the cooperative application of such techniques.

Reachability graph methods become inefficient mainly because of the freedom in the firing order of concurrent transitions. Such freedom can generate an explosion of the number of states. One of the techniques for reducing the state space is the stubborn-set method [14]. Such technique aims to construct a smallest possible subset from the state graph.

# 4 The data flow Petri net

The data flow Petri net is obtained by a direct analysis of the occam syntax. It produces a net that is safe, free of loops and free of place conflicts, by construction. In such model, places represents values (variables, constants and intermediate values) whereas transitions represents actions and operations. The presence of a token in a place models the availability of a value. Each occam construction is modeled as follows.

**Definition 6** Current and next nets. *Let $PN$ be the set of Petri nets and $OP$ be the set of* occam *processes. Let $TF : PN \times OP \rightarrow PN$ be the* translation function *from* occam *to Petri nets. Let $N = \langle P, T, I, O \rangle$ and $N' = \langle P', T', I', O' \rangle$ be Petri nets. If $N, N'$ are such that $N' = TF(N, op), \forall op \in OP$, then $N$ and $N'$ are defined as the* current net *and* next net, *respectively.*

**Combiners.** Combiners are occam processes that group other processes. SEQ/PAR, IF/ALT and WHILE are the combiners supported by this model.

SEQ and PAR combiners are translated in the same way. From the data flow point of view, only actions which are of the kind *read after write* are modeled as a sequence. The other kind of actions are modeled in parallel (*read after read*, *write after read* and *write after write*). Therefore, the control-flow has the task of ordering these actions. Due to space restrictions, only the SEQ/PAR constructor will be described formally.

**Definition 7** Parent and descendent processes. *A* parent process *is a combiner that groups other processes. A* descendent process *is a process grouped by a combiner.*

**Definition 8** SEQ/PAR translation. *Let $N, N'$ be a current and a next net, respectively. If SEQ/PAR is labeled as process $i$, its translation is then $P' = P \cup \{p_i.start, p_i.end\}$, $T' = T \cup \{t_i.start, t_i.end\}$, $I^{t\prime}(t_i.start) = \{p_i.start\}$, and $O^{t\prime}(t_i.end) = \{p_i.end\}$. If process $i$ is descendent from process $j$, then $O^{t\prime}(t_j.start) = O^t(t_j.start) \cup \{p_i.start\}$ and $I^{t\prime}(t_j.end) = I^t(t_j.end) \cup \{p_i.end\}$.*

The translations of IF and ALT combiners are similar (Fig. 2). From the data flow point of view, the results of every branch must be available at any moment. If some branch fails, due to a data dependency, the whole conditional process must fail. An arc is added from the place representing the guard expression to the *start* transition of the IF/ALT process.

WHILE combiner is translated in the same way of IF/ALT, except by a small difference (Fig. 2). The expression corresponding to the guard is evaluated twice (before and after translation of the guarded process). In this approach, the loop body is modeled once, with no feedback. The second translation of the guard expression supports the situations where some variable which is used in that expression is modified inside the loop.

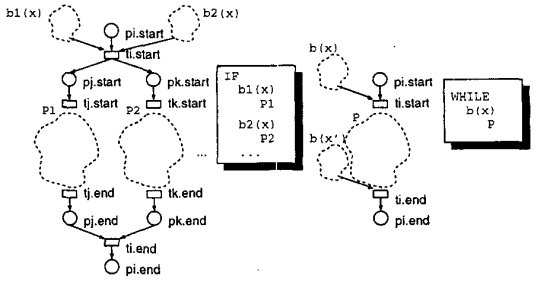


Figure 2: Conditional combiners

**Literals.** Literals are constant values implicitly declared by processes. All literals are represented by places attached to the *start* transition of the process

where they appear and to the transition representing the action which uses them (Fig. 3).

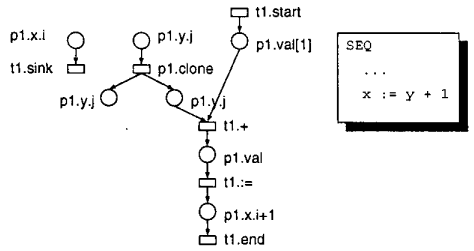


Figure 3: Variables, expressions and assignment

**Variables.** Variables are represented by their instances, where each instance is represented by one or more places. Each time a new value is assigned to a variable, a new instance of this variable is added to the model. Further accesses to this variable are performed to the new instance.

In the model, there must be a instance always ready to be read at any moment. Hence, any time a variable is read, a *clone* transition is attached to the place representing the current instance of this variable. Two extra places are attached to the *clone* transition: one is used in the operation being modeled and the other remains for further accesses (Fig. 3).

The writing of a variable is modeled by discarding its current instance, which is performed by a *sink* transition. A new place is created in the model and the action which assigns a value to that variable is attached to the new place.

**Declarations.** Two kinds of declarations are carried out by this model: channels, which provides communication between processes; and variables, which provides a local temporary place to store values.

Channels are modeled by a place that is attached only to the operations which perform some action on them. There will be exactly one place for each declared channel in the specification.

Variables are modeled as it has be seen earlier, but two special situations arise. When a variable is declared, it has the first instance initialized with an arbitrary initial value. This is modeled by attaching the place representing the first instance to the *start* transition of the process which declares the variable. When a process finishes, all values which were produced are cleaned. This is modeled by attaching the place representing the last instance of each variable declared inside the process to the *end* transition of the process.

**Expressions.** An expression is modeled by attaching its operands to a transition representing the operator. The expression result is modeled by adding a new place to the model, which is attached to the operator transition (3).

**Actions** The assignment operation is modeled by adding an *assign* transition to the model. The transition is attached to the place representing the value being assigned to and to the place representing the new instance of variable being assigned.

The input operation is the receiving a set of values from a channel. This is modeled by a transition representing the input action, which is attached to all places representing the values being received. It is also attached to the places representing the new instances of variables being assigned.

The output action is the sending of a set of values through a channel. This is modeled by a transition representing the output action, which is attached to all places representing the values being sent. It is also attached to the place representing the channel.

An automatic translation mechanism has been completely implemented. The tool reads an occam specification and produces an INA *net file*. It also generates all initial and target markings which will be used for the data dependency analysis.

# 5 Data dependency analysis

The *normal form* defined by the PISH methodology is a parallel composition of *simple processes*, as it has be seen in section 2. The data dependency analysis will be carried out considering the *simple process* granularity. Hence, each process will be tested against all the others for discovering data dependencies between pairs of processes.

**Definition 9** Enabled and disabled processes. *Let* $N = \langle P, T, I, O, M_0 \rangle$ *be a marked Petri net. A process* $i$ *is said enabled if* $M_0(p_i.start) = 1$ *and disabled if* $M_0(p_i.start) = 0$.

**Definition 10** Executable and non-executable processes. *Let* $N = \langle P, T, I, O, M_0 \rangle$ *be a marked Petri net. Let* $M$ *be a marking, such that* $M(p_i.end) = 1$. *A process* $i$ *is said executable if there exists a marking* $M'$, *reachable from* $M_0$ *such that* $M' \geq M$. *Otherwise, the process is said non-executable.*

The following algorithm performs the data dependency analysis:

1. For each process $P_i$ do:
   (a) Disable process $P_i$ and enable all the other processes
   (b) For each process $P_j \neq P_i$.
      i. If $P_j$ is non-executable, then mark $P_j$ as data-dependent on $P_i$
      ii. If $P_j$ is executable, then mark $P_j$ as data-independent on $P_i$

In a first approach, the state equation method was applied. The correctness of the state equation results is well known for live marked graphs, which is a superclass of the model net. This approach has shown ineffective, mainly for two reasons: the net is not live during the data dependency test (due to the disabled processes) and the computational cost for state equation raises proportionally to the number of nodes.

A second approach was chosen and it has demonstrated to be effective to perform the data dependency analysis. It is based on the reachability graph construction, using stubborn-set reductions, available in the INA tool [15].

# 6 Example – a vending machine

The method was applied to a vending machine controller, whose behavior is the following. The vending machine accepts coins from client until a soft drink is requested. If the money inserted is insufficient for the purchase of the drink or the chosen drink is not available, the money is refunded. Otherwise, the machine checks if there is sufficient coins to dispense the change. If it can give the change, it delivers both the drink and the change. If not, the client chooses between receiving the drink, discarding the change, or money refund. After the *splitting* phase, it is composed by 7 processes (Fig. 4).

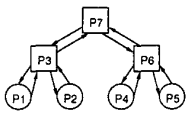


Figure 4: Vending machine processes

The complete data flow Petri net of the vending machine is composed by 460 places (including 35 global channel places) and 258 transitions. Table 1 depicts, for each process, the number of places and transitions which models the process. It also shows the number of states in the reachability graph, considering the stubborn-set reducing technique applied to the initial marking of the process.

Considering the results obtained for the initial marking of process $P_1$, the reachability analysis without any reduction generates much more than 100,000 states. On the other hand, applying only state space reductions leads to 93 states. Furthermore, applying cooperatively both structural and state space reductions on the net yields only 52 states.

Table 1: The vending machine data flow Petri net

| Process | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ |
|---|---|---|---|---|---|---|---|
| Local P | 83 | 84 | 59 | 39 | 102 | 44 | 14 |
| Local T | 58 | 46 | 33 | 25 | 59 | 27 | 10 |
| States | 93 | 142 | 108 | 178 | 209 | 174 | 162 |

The data dependency analysis produced the results shown on table 2. One important aspect of the result is the fact that despite of process $P_1$ is being attached to controller $P_3$ (and indirectly to $P_7$), it does not depend on any process. It is true, due to the fact that process $P_1$ only initializes variables with constant values, discarding any previous values of all variables it receives. In the PISH *normal form* every data dependency is a communication dependency, but not all communication dependency is a data dependency. The model can capture such difference.

Other characteristic of the model is also evident: it captures the transitivity of the data dependency. If a process $P_i$ depends on a process $P_j$ and $P_j$ depends on $P_k$, the analysis result contains the dependency between $P_i$ and $P_k$.

Table 2: Data dependency analysis results

| Process | Depends on |
|---|---|
| $P_1$ | — |
| $P_2$ | $P_1$, $P_3$ |
| $P_3$ | $P_1$, $P_2$, $P_7$ |
| $P_4$ | $P_1$, $P_2$, $P_3$, $P_6$, $P_7$ |
| $P_5$ | $P_1$, $P_2$, $P_3$, $P_4$, $P_6$, $P_7$ |
| $P_6$ | $P_1$, $P_2$, $P_3$, $P_4$, $P_5$, $P_7$ |
| $P_7$ | $P_1$, $P_2$, $P_3$, $P_4$, $P_5$, $P_6$ |

# 7 Conclusions and future works

This work has formalized a model for analyzing data dependencies between concurrent processes in the nor-

mal form of PISH methodology. The model produces results which are necessary to *classification* phase of PISH methodology and were formerly obtained in an non-formal way.

Furthermore, the model points only to real data dependencies between processes, not to communication (synchronization) dependencies. The *splitting* phase of PISH methodology introduces sequential controllers to a sequence of processes, even if they are data-independent. Based on the results of the data dependency analysis, it becomes possible to change the sequential controller to a parallel controller, improving the parallelism degree of the final system.

Extensions to this model should be developed in order to extract other kind of information, like variable lifetime (for quantifying the data dependency and register allocation). Integration to the control-flow model also under development should be considered for extracting more information (deadlock freeness, serialization strategies, etc).

## Acknowledgments

Prof. Peter Starke and Stephan Roch from Humboldt-Universität, developers of INA Tool, for their support.

# References

[1] R. Gupta and G. DeMicheli, "Hardware/software co-systhesis for digital systems," *IEEE Design & Test of Computers*, pp. 29–41, 1993.

[2] G. DeMicheli, "Hardware/software co-design: Application domains and design technologies," in *Hardware/Software Co-design*, G. DeMicheli and M. Sami, Eds., pp. 1–28. Dordrecht Kluwer Academic Publishers, 1996.

[3] Dick Pountain and David May, *A Tutorial Introduction to occam Programming*, BSP Professional Books, Oxford, UK, 1987.

[4] Edna Barros and Augusto Sampaio, "Towards provable correct hardware/software partitioning using OCCAM," in *Proceedings of 3rd International Workshop on Hardware/Software Co-Design*, Los Alamitos, 1994, pp. 210–217, IEEE.

[5] Leila Silva, Augusto Sampaio, and Edna Barros, "A normal form reduction strategy for hardware/software partitioning," in *Proceedings of 4th International Symposium on Formal Methods - Europe, FME'97*, Sringer Verlag, 1997.

[6] Paulo Maciel, Edna Barros, and Wolfgang Rosenstiel, "Computing communication cost by Petri Nets for hardware/software co-design," in *8th IEEE International Workshop on Rapid System Prototyping*, Chapel Hill, North Carolina, USA, June 24-26 1997.

[7] Paulo Maciel, Edna Barros, and Wolfgang Rosenstiel, "A Petri net approach for quantifying mutual exclusion degree," in *Proceedings of INCOM'98*, Nancy-Metz, France, June 23-27 1998.

[8] Paulo Maciel, Edna Barros, and Wolfgang Rosenstiel, "Estimating functional unity number in PISH co-design system by using Petri nets," in *Proceedings of IEEE 12th Symposium on Integrated Circuits and Systems Design*, Natal, Brazil, October 1999.

[9] Paulo Maciel, Edna Barros, and Wolfgang Rosenstiel, "A Petri Net approach to compute load balance in hardware/software co-design," in *High Performance Computing'98*, Boston, Massachusets, USA, April 5-9 1998.

[10] Paulo Maciel, Edna Barros, and Wolfgang Rosenstiel, "A Petri Net based approach for performing the initial allocation in hardware/software co-design," in *1998 IEEE International Conference on Systems, Man and Cybernetics*, San Diego, California, USA, October 11-14 1998.

[11] Paulo Maciel, Edna Barros, and Wolfgang Rosenstiel, "A Petri net model for hardware/software co-design," *Design Automation for Embeded Systems*, vol. 4, pp. 243–310, 1999.

[12] Edna Barros, *Hardware/Software Partitioning Using UNITY*, Ph.D. thesis, Tübingen Universität - Fakultät für Informatik, Germany, 1993.

[13] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[14] Antti Valmari, "The state explosion problem," in *Lecture Notes on Petri Nets I: Basic Models*, number 1491 in Lecture Notes in Computer Science, pp. 429–528. Springer-Verlag, 1998.

[15] Stephan Roch and Peter Starke, *INA - Integrated Net Analyzer, Version 2.2*, Humboldt-Universität zu Berlin - Institut für Informatik, Germany, 1999.