Derivation of Formal Representations from Process-based Specification and Implementation Models

Steven Vercauteren*, Diederik Verkest*, Gjalt de Jong† and Bill Lin‡
* IMEC Laboratory, Kapeldreef 75, B-3001 Leuven, Belgium
† Alcatel Telecom, F.Wellesplein 1, B-2018 Antwerpen, Belgium
‡ ECE Department, University of California, San Diego, La Jolla, CA

Abstract

In this paper we present a formal framework to verify timing properties of embedded systems. We propose a process calculus as an intermediate model to map between language-level constructs of process-based specification and implementation models, and Petri net operations. We present an elegant translation scheme to generate Petri nets starting from the intermediate process expressions. The approach has been applied to verify the freedom of deadlock in a QAM modem design, with promising results.

1. Introduction

As embedded systems become increasingly complex, they become increasingly prone to errors and difficult to design. From experience gained with several large scale design projects in digital communications [4, 8], we have encountered a number of subtle errors during both the specification and implementation phases, that were difficult to trace using traditional analysis methods such as simulation. Since most modern embedded systems are concurrent in nature, usually implemented using a heterogeneous architecture containing multiple hardware/software components, hard to detect errors can result from unanticipated interactions between the concurrent parts. Over the past years, a number of automated formal verification methods [5, 6, 12, 15] have emerged, that can address some of these verification problems. However, to leverage these methods, the specification model used to represent the system must be formal in the first place as ambiguity in specification semantics is in itself a major source of errors.

In this paper, we propose a formal model based on Petri nets [13] for reasoning about the behavior of a concurrent system. We have chosen the Petri net formalism, because it is well suited to model concurrency, choice, and causality, and because there is a wealth of formal verification techniques [15, 18] that can exploit the inherent partial order properties explicitly captured in the model. However, a major problem when using Petri nets as a model for verification, is the lack of a formal mapping between constructs in the (system) specification language itself and Petri net operations. We address this problem by providing such a formal mapping using a CSP [9]/CCS [10]-like process calculus as an intermediate model. The underlying idea is that many of the existing process-based specification languages (e.g. [2]) can be translated to what we call "intermediate code" of this process calculus, which can subsequently be translated into a Petri net representation by applying a syntax-directed mapping scheme. Once arrived at the Petri net level, existing formal verification methods can then be leveraged. Because the overall mapping is semantics preserving, these methods can be used to validate the various intermediate specifications of the design process.

The impact of this work on the design of embedded systems is twofold. At the specification phase, errors can be detected quickly without requiring extensive simulations. Secondly, when designing an embedded system, many problems occur at the hardware/software interface. Therefore, a design support tool has been developed [19] to assist the designer in the mapping of software components on programmable processors. Such a tool assembles the hardware/software interface by selecting and combining I/O scenarios from a library. Individual scenarios are correct by construction. However, when assembling these scenarios to construct the complete interface, subtle (timing) problems may arise, as will be shown in section 5. The formal framework presented in this paper allows to diagnose these problems, such as possible deadlocks. This necessitates the modeling of the complete hardware/software interface, including the processor core, run-time operating system, hardware and software device drivers, etc.

The remainder of this paper is organized as follows. Section 2 presents the basic concepts of the used process calculus. Section 3 reviews the basic definitions of Petri nets, and details the *syntax-directed* translation procedure of process expressions to Petri nets. Section 4 highlights the verification framework. Section 5 extensively describes a case study. Finally, conclusions are drawn in section 6.

2 Process Calculus

In our model we start from a set of given **atomic rendezvous actions**. These actions are taken to be indivisible, and form the "leaf" processes of the model. The occurrence of an action is called an event [1], and it is the result of two concurrent processes - in some cases the environment is implicitly assumed to be the counterpart process - both engaging in the execution of the event under consideration. Every event can thus be seen as a **binary** synchronization. Two types of events exist: *visible* events and *hidden* events. Visible events represent *external* synchronizations (i.e. which are externally observable); the set of all visible events is denoted as $\boldsymbol{Vis}$. Hidden events represent *internal synchronizations* and are denoted by the special symbol τ. The environment cannot prevent a process from engaging in a *hidden* event. Indeed, as τ denotes an internal synchronization and all synchronizations are binary, it cannot be synchronized upon by any other process.

Processes are built from other (leaf) processes by means of the following operators: *sequential composition*, *deterministic* and *non-deterministic choice composition*, *parallel composition*, *recursion* and *interrupt composition*.

Processes are denoted by *process expressions*. We also assume a set $\boldsymbol{Var}$ of *process variables* denoted by x, y, etc. The set $\boldsymbol{PE}$ of all possible process expressions is defined by the following production system, where R is used as start symbol:

$$
\begin{array}{l}
R ::= S \mid T \\
S ::= x \mid S.R \\
T ::= a \mid T.R \mid T + T \mid T^{\circ} \| T^{\circ} \mid T^{\circ} \rhd T^{\circ} \mid \mu x.T \\
\text{where} \quad a \in (\boldsymbol{Vis} \cup \{\tau\}) \\
\qquad\quad T^{\circ} \text{ is } T \text{ without free occurrences of variables}
\end{array}
$$

In the following, the intuitive meaning of the different operators is described.

Sequential Composition of two processes P and Q, denoted $P.Q$, is a process that starts the execution of Q after the *successful completion* of P. Successful completion of a process P is defined as process P reaching a wanted [2] state from which no actions can be performed. This operator constitutes a mixture of the *prefix* and *sequential composition* operators, as they are used in CSP [9].

Choice. The deterministic choice of two processes P and Q, denoted $P + Q$, is the process that can do either P or Q, where the choice is made by the environment. The non-deterministic choice of two processes P and Q, on the other hand, denoted $P \sqcap Q$, is the process that can either do P or Q, where the choice is made by the process itself. In both cases, the decision of which alternative to take, should be made at the beginning of the choice. Notice that the notation $P \sqcap Q$ is a shorthand for the process $(\tau.P) + (\tau.Q)$. Indeed, as τ denotes a hidden or internal synchronization event, the outside world cannot interfere at the beginning of the choice; $(\tau.P) + (\tau.Q)$ then degenerates to a non-deterministic choice. This construction is borrowed from CCS [10].

Parallel Composition of two processes P and Q, denoted $P \| Q$, executes the processes P and Q simultaneously and independently, except for events that are common to both processes; if both processes contain an event a, this event can only occur if both parties are ready to engage in a. With the exception that we only allow binary synchronization, this operator is also used in CSP.

Interrupt Composition of process P by a process Q, denoted as $P \rhd Q$, is the process that behaves like P but which is *interrupted* on the occurrence of the first event of Q. When interrupted, process P arrives in a *halt* state, and $P \rhd Q$ behaves like Q. After Q has terminated successfully, P leaves its halt state and $P \rhd Q$ resumes P, until possibly being interrupted again. If P terminates unsuccessfully, it is still possible for $P \rhd Q$ to behave like Q. If P terminates successfully, on the other hand, Q cannot start execution any more. The semantics of the interrupt composition, as they are presented here, differ from the ones defined by CSP and LOTOS [11]. In the latter models, an interrupt is interpreted as an *abort*; when a process Q "interrupts" a process P, process Q is executed, but P is never resumed. Clearly, this kind of preemption cannot be reduced to the interrupt behavior of today's processors.

Recursion is introduced by considering solutions of equations of the form $x = F(x)$, denoted as $\mu x.F(x)$, with $F(x)$ an expression in (process) variable x, constructed solely in terms of the sequential composition, choice composition, parallel composition and interrupt composition operators. Notice that an expression beginning with a variable, can only be used in a composition if it is preceded by an event $a \in \boldsymbol{Vis} \cup \{\tau\}$.

Besides the interrupt operator, our process calculus has no larger expressiveness than CSP and CCS. The key idea of our approach is that we believe many of the existing process-based languages (e.g. [2]) used to specify heterogeneous systems, can be translated to "intermediate code" of this calculus which then serves as a starting point for performing a syntax-directed Petri net translation, as described in the next section.

Example. Consider the following program fragment, written in C, as it can be found in a process description within the CoWare$^{\mathrm{TM}}$ data model [2]:

```
while(!(x%y)) {
 send(a,x);
 x++;
 receive(b,y);
}
 send(c,(x+y));
```

The `send(a,x)` statement initiates a communication to send data along channel `a`. If the environment, at the other

[1] In the sequel we will indistinctively use the notations "event" and "action"

[2] Because we have to exclude the deadlock state.

side of the channel, is not ready to receive data along this channel, the program is halted at this point. As soon as the environment becomes ready, data is transfered, and both parties proceed independently. The occurrence of this communication is modeled by the atomic event a. For the `receive(b,y)` and `send(c,(x+y))` statements the same reasoning applies. This program fragment is then modeled as $X = (a.b.X) \sqcap c$. Every time we encounter the `while` statement, the decision of (re-)entering the loop is made internally (`!(x%y)`) - we abstract from all internal data -and cannot be influenced by the environment. As a result, it is modeled as a non-deterministic choice.

3. Syntax-directed Translation to Petri Nets

In this section the translation of process expressions into Petri nets [13] is described in more detail. In section 3.1 we review some of the basic definitions and properties of Petri nets. In section 3.2 the composition operators are defined at the Petri net level. In section 3.3 we discuss the translation process itself.

3.1. Basic Definitions and Properties

Definition 3.1 (Labeled Petri Net) *A labeled Petri net (LPN) is a tuple* $\Sigma = \langle P, A, F, m_0 \rangle$ *with* $P \cap A = \emptyset$, $F \subseteq 2^P \times A \times 2^P$ *and* $m_0 : P \to \mathbb{N}$.

In the above definition P denotes a set of *places*, A a set of *actions*, F the set of *transitions* and m_0 an initial marking. For a transition $t = (p, a, q)$, a denotes the *label* or the *action* of t, where as p and q are often referred to as the set of *input* and *output* places of t, respectively.

Besides the structure of a Petri nets, there is also an associated dynamics. A *state* or marking, is the mapping of the places to the natural numbers, indicating the number of tokens in the places. Transitions between states are dictated by the following firing rule. In the sequel M_P denotes the set of all states (markings) of a Petri net with $|P|$ places.

Definition 3.2 (Enabling Rule) *Let* (p, a, q) *be a transition,* $m \in M_P$. $Enabled((p, a, q), m) \equiv \forall p : m(p) \geq 1$

Definition 3.3 (Firing Rule) *Let* $t = (p, a, q)$ *be a transition,* $m \in M_P$, $Enabled(t, m)$ = *true.*

$$NextState(m(p'), t) = \begin{cases} m(p') - 1 & \text{if } p' \in p \setminus q \\ m(p') + 1 & \text{if } p' \in q \setminus p \\ m(p') & \text{otherwise} \end{cases}$$

Definition 3.2 states that a transition t can fire if all its input places contain at least one token. Definition 3.3 states that firing of t removes one token in all its input places and adds a new token in all its output places.

The set of all reachable states is represented in a *reachability graph*. In such a reachability graph all vertices correspond to a valid marking of the Petri net and all arcs correspond to a transition from one marking to another due to firing of some transition in the net. The reachability graph of a Petri net N, denoted as $RG(N)$, can then be interpreted as the reflexive transitive closure of the next-state relation defined in definition 3.3.

Two important properties of Petri nets are *liveness* and *safeness*. Liveness concerns the question whether a transition can ever be fired, and is clearly opposed to *deadlock*. Safeness means that a place *does not* contain more than one token at any time.

3.2. Process Operators on Petri Nets

In this section the process of translating process expressions into Petri nets is highlighted. The approach is similar to the macro-module mapping approach [16] for translating a concurrent program into an asynchronous circuit. For each syntactical construct a Petri net element or operator is defined. In this way a process expression can be translated into a Petri net using a *syntax-directed mapping* scheme, detailed below.

In literature a lot of research has been devoted to the development of operators for the composition of Petri nets. In [17] finite and safe nets are constructed for the *Algebra of Communicating Processes*(ACP) of Bergstra and Klop [1] without recursion. The notion of a non-deterministic choice is absent either. In [14] safe and finite Petri nets are generated from a so-called *anonymous* language that contains CCS and almost the whole CSP as special "cases". The alternatives of a choice may however not contain parallelism, and every body of a recursion starts with an invisible action τ. In [7] a *Communication Petri Net Model* is proposed. The focus is a Petri net algebra; the concepts of successful completion (cf. sequential composition) and recursive processes are therefore missing.

In this work, the translation of a process expression into Petri nets is defined by means of translation function $\mathcal{PT}$. The syntax-directed organization of $\mathcal{PT}$ makes it necessary to include all possible syntactic constructs of a process expression in the domain of $\mathcal{PT}$, i.e. the set of variables $\boldsymbol{Var}$, the atomic actions, as well as the process operators.

As discussed in section 2, the sequential composition operator requires the notion of successful completion. As a result, the definition of a Petri net has to be extended to represent successful termination. Therefore we have chosen to classify certain places of the Petri net as *end places*. When all end places contain a token, the involved Petri net has terminated successfully. This new Petri net model is denoted as a *PE-net* and is defined below.

Definition 3.4 *A PE-net is a tuple* $\Sigma = \langle P, T, A, m_0, E \rangle$, $P \cap T = \emptyset$, $F \subseteq 2^P \times A \times 2^P$, $m_0 \subseteq P$ *and* $E \subseteq P$.

In the above definition P, T, F and m_0 have the same meaning as in (classical) labeled Petri nets (see definition 3.1) and E denotes the end places. The definitions and properties of (classical) labeled Petri nets can be lifted to

PE-nets in a straightforward way. The initial marking of a PE-net, however, is represented by the set of *initial places*, i.e. the set of places that contain a token in the initial marking. This is possible due to the safe Petri net representations of the allowed process expressions. A safe Petri has in each place at most one token and its marking can therefore be represented by a set of places m, where $p_i \in m$ indicates that there is a token in p_i. A non-safe representation would imply that the corresponding process expression exhibits *auto-concurrency*, i.e. is of the form $x = P \| x; \ldots$, which is not allowed in our syntax.

Leaf Processes. The PE-net representations of an atomic action $a \in \boldsymbol{Vis} \cup \{\tau\}$ and a variable $x \in \boldsymbol{Var}$ are illustrated in Figure 1. The end places are depicted as circles with double perimeter.

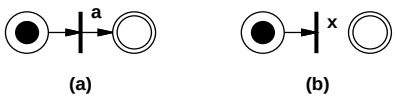


Figure 1: PE-net representations of (a) $a \in \boldsymbol{Vis} \cup \{\tau\}$ (b) $x \in \boldsymbol{Var}$

Definition 3.5 *Given an atomic action $a \in \boldsymbol{Vis} \cup \{\tau\}$, the corresponding PE-net, denoted as $\mathcal{PT}(a)$, is defined as* $\langle \{p_1, p_2\}, \{a\}, \{(\{p_1\}, a, \{p_2\})\}, \{p_1\}, \{p_2\} \rangle$

Definition 3.6 *Given a variable $x \in \boldsymbol{Var}$, the corresponding PE-net, denoted as $\mathcal{PT}(x)$, is defined as* $\langle \{p_1, p_2\}, \{x\}, \{(\{p_1\}, x, \emptyset)\}, \{p_1\}, \{p_2\} \rangle$

Sequential Composition. For the PE-net representation of a process expression $Q_1.Q_2$, the end places of $\mathcal{PT}(Q_1)$ are combined with the initial places of $\mathcal{PT}(Q_2)$, by means of a Cartesian product, effectively "abutting" the two PE-net representations together. This is shown in Figure 2.

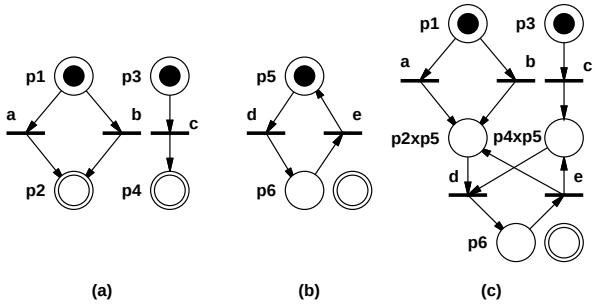


Figure 2: Petri net representations (a) $\mathcal{PT}(Q_1)$ (b) $\mathcal{PT}(Q_2)$ (c) $\mathcal{PT}(Q_1.Q_2)$

Definition 3.7 (Sequential Composition) *Let Q_1 and Q_2 be two process expressions, with $\mathcal{PT}(Q_1) = \langle P_1, A_1, F_1, M_{0_1}, E_1 \rangle$ and $\mathcal{PT}(Q_2) = \langle P_2, A_2, F_2, M_{0_2}, E_2 \rangle$. $\mathcal{PT}(Q_1.Q_2)$ is defined as:*

$\langle (P_1 \setminus E_1) \cup (P_2 \setminus M_{0_2}) \cup (E_1 \times M_{0_2}), A_1 \cup A_2, F', M_{0_1}, E_2 \rangle$

where

$$F' = \{(p, a, q \setminus E_1 \cup (q \cap E_1) \times M_{0_2}) \mid (p, a, q) \in F_1\} \cup \{(p \setminus M_{0_2} \cup E_1 \times (p \cap M_{0_2}), a, q \setminus M_{0_2} \cup E_1 \times (q \cap M_{0_2})) \mid (p, a, q) \in F_2\}$$

In contrast to [17] there is no need for one-step unfolding of $\mathcal{PT}(Q_2)$, as well as an expensive *complement* step. In [14] sequential composition is defined as a special case of parallel composition; by doing so, however, there is a need for an extra *concealment* or *hide* operator.

Choice Composition. For the PE-net representation of a process $Q_1 + Q_2$, conflicts are introduced between all pairs of initial transitions of $\mathcal{PT}(Q_1)$ and $\mathcal{PT}(Q_2)$, by means of a Cartesian product construction. The end places are combined similarly. For this to work properly, the initial places that are in cycles have to extracted by a one-step unfolding; in a choice, once the decision of what branch to take is made by the first execution of a transition, a loop iteration may then not cause the other branch to be taken. In our translation scheme, the PE-net representation of a recursive process definition already implements the desired unfolding (see Definition 3.9). Then, the above situation can only result from process expressions of the form $P + (R \rhd Q) \ldots$. In these situations the one-step unfolding is accomplished through a separate *root-unwinding* step. This preprocessing step is only effective for those initial transitions that are in cycles. The PE-net equivalent of the choice operator is shown in Figure 3.

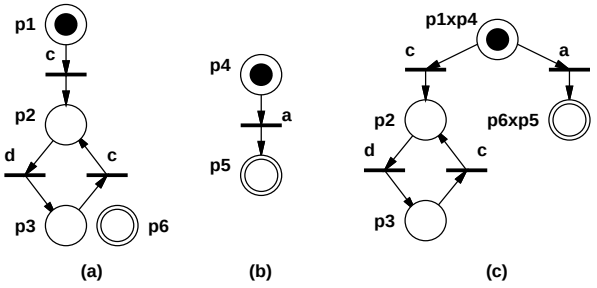


Figure 3: Petri net representations (a) $\mathcal{PT}(Q_1) = RootUnw(\mathcal{PT}(Q_1))$, with $Q_1 = \mu x.(c.d.x)$ (b) $\mathcal{PT}(Q_2) = RootUnw(\mathcal{PT}(Q_2))$ (c) $\mathcal{PT}(Q_1 + Q_2)$

Definition 3.8 (Root Unwinding) *Let $N = \langle A, P, F, M_0, E \rangle$ be a PE-net. Let P_{new} be new places $\notin P$, P_{cyc} the initial places of N, that are not in cycles, defined by $\{s \in P \mid M_0(s) \neq 0 \wedge \neg(\exists (p, a, q) \in F : s \in q)\}$, and η a bijection between P_{new} and P_{cyc}. The* root-unwinding *of N, denoted as $RootUnw(N)$ is defined as:*

$\langle A, P \cup P_{new}, F', (P \setminus P_{cyc}) \cup P_{new}, E \rangle$

where

$$F' = F \cup \{(p \cup p_n, a, q) \mid \ p_n \subseteq P_{new} \wedge (p \cup \mathrm{H}(p_n), a, q) \in F\}$$
$$\mathrm{H}(\{p_1, ..., p_n\}) = \{\eta(p_1), ..., \eta(p_n)\}$$

Definition 3.9 (Choice Composition) *Let Q_1 and Q_2 be two process expressions, with $RootUnw(\mathcal{PT}(Q_1)) = \langle P_1, A_1, F_1, M_{0_1}, E_1\rangle$ and $RootUnw(\mathcal{PT}(Q_2)) = \langle P_2, A_2, F_2, M_{0_2}, E_2\rangle$. $\mathcal{PT}(Q_1 + Q_2)$ is defined as:*

$\langle((P_1 \setminus M_{01}) \setminus E_1) \cup ((P_2 \setminus M_{02}) \setminus E_2) \cup (M_{01} \times M_{02}) \cup E_1 \times E_2, A_1 \cup A_2, F', M_{01} \times M_{02}, E_1 \times E_2\rangle$

where

$$F' = \{(p_1 \setminus M_{01} \cup (p_1 \cap M_{01}) \times M_{02}, a_1, q_1 \setminus E_1 \cup (q_1 \cap E_1) \times E_2) \mid (p_1, a_1, q_1) \in F_1\} \cup \{(p_2 \setminus M_{02} \cup M_{01} \times (p_2 \cap M_{02}), a_2, q_2 \setminus E_2 \cup E_1 \times (q_2 \cap E_2)) \mid (p_2, a_2, q_2) \in F_2\}$$

Besides the treatment of the end places, the above construction is similar to [7, 17]. In [17], however, the root-unwinding step introduces $2^{\#P_{cyc}}$ new transitions, with P_{cyc} the set of initial places that are in cycles. In [7], an extra restriction applies: all transitions that have initially marked input places must be enabled.

Parallel composition. In Petri nets a transition can be regarded as a synchronization mechanism since it can only fire if all input places contain at least one token. To model parallel composition with rendezvous synchronization, it is then sufficient to "join" the transitions that have the same action label, different from τ. Since more than one transition may be labeled with the same action, all combinations have to be considered. This is shown in Figure 4.

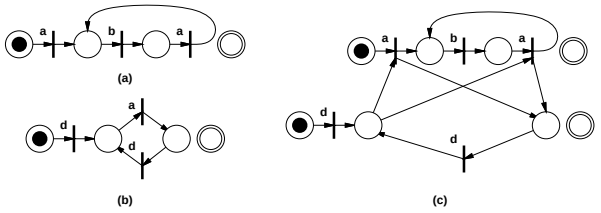


Figure 4: Petri net representations (a) $\mathcal{PT}(Q_1)$ (b) $PT(Q_2)$ (c) $PT(Q_1 \| Q_2)$

Definition 3.10 (Parallel Composition) *Let Q_1 and Q_2 be two processes, with $\mathcal{PT}(Q_1) = \langle P_1, A_1, F_1, M_{0_1}, E_1\rangle$ and $\mathcal{PT}(Q_2) = \langle P_2, A_2, F_2, M_{0_2}, E_2\rangle$. $\mathcal{PT}(Q_1 \| Q_2)$ is defined as:*

$\langle P_1 \cup P_2, A_1 \cup A_2, F', M_{01} \cup M_{02}, E_1 \cup E_2\rangle$

where

$$F' = \{(I, a, O) \in F_1 \cup F_2 | a \notin (A_1 \cap A_2) \setminus \{\tau\}\} \cup \{(I_1 \cup I_2, a, O_1 \cup O_2) | \ a \in (A_1 \cap A_2) \setminus \{\tau\} \wedge (I_i, a, O_i) \in F_i\}$$

Besides the treatment of τ events, the above construction is similar to [7].

Interrupt Composition. For the PE-net representation of $Q_1 \rhd Q_2$, we create a new place for every transition of $\mathcal{PT}(Q_1)$. Each new place is then connected via a self-loop with its corresponding transition. These new places are then combined with the initial places as well as the end places of $\mathcal{PT}(Q_2)$, by means of a Cartesian product construction. If Q_2 is a recursive process expression, the one-step unfolding within $\mathcal{PT}(Q_2)$ (see Definition 3.12) then prevents the transitions of $\mathcal{PT}(Q_1)$ from firing for every loop iteration within the former net. However, if Q_2 is of the form $(P \rhd R)\ldots$, within $\mathcal{PT}(Q_2)$ there can still be initial places that are in cycles (see Definition below). In this case an extra root-unwinding step is necessary. For the other cases, this preprocessing step has no effect. This construction is shown in Figure 5.

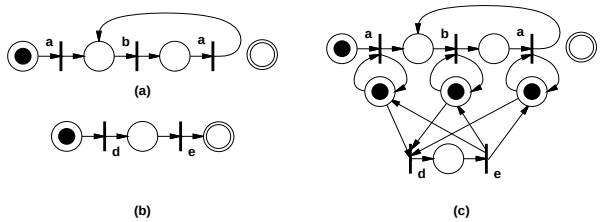


Figure 5: Petri net representations (a) $\mathcal{PT}(Q_1)$ (b) $\mathcal{PT}(Q_2) = RootUnw(\mathcal{PT}(Q_2))$ (c) $\mathcal{PT}(Q_1 \rhd Q_2)$

Definition 3.11 (Interrupt Composition) *Let Q_1 and Q_2 be two processes, with $\mathcal{PT}(Q_1) = \langle P_1, A_1, F_1, M_{0_1}, E_1\rangle$ and $RootUnw(\mathcal{PT}(Q_2)) = \langle P_2, A_2, F_2, M_{0_2}, E_2\rangle$. Let P_{new} be new places such that $P_1 \cap P_2 \cap P_{new} = \emptyset$, and H a bijection $\mathrm{H} : F_1 \to P_{new}$. $\mathcal{PT}(Q_1 \rhd Q_2)$ is defined as:*

$\langle P_1 \cup (P_2 \setminus (M_{02} \cup E_2)) \cup (P_{new} \times M_{02} \times E_2), A_1 \cup A_2, F', M_{0_1} \cup (P_{new} \times M_{02} \times E_2), E_1\rangle$

where

$$F' = \{(p_1 \cup p_n, a_1, q_1 \cup p_n) \mid (p_1, a_1, q_1) \in F_1 \wedge p_n = \mathrm{H}(p_1, a_1, q_1) \times M_{02} \times E_2\} \cup \{(p_2 \setminus M_{02} \cup (P_{new} \times (p_2 \cap M_{02}) \times E_2), a_2, q_2 \setminus E_2 \cup (P_{new} \times M_{02} \times (q_2 \cap E_2))) \mid (p_2, a_2, q_2) \in F_2\}$$

Recursion. For the PE-net representation of a process $\mu x.F(x)$, with $F(x)$ a process expression containing variable x, we first compute $\mathcal{PT}(F(x))$, using the translation techniques described above. The input places of each transition with label x are then combined with the initial places, by means of a Cartesian product, effectively creating the desired loop. The initial places as well as the initial marking are kept; this construction implements a one-step unfolding making the root-unwinding step obsolete for the PE-net translation of process expressions of the forms $P + P$ and $Q \rhd P$, with P a recursive process definition. This is shown in Figure 6.

Definition 3.12 (Recursion) *Let $F(x)$ be process expression with process variable x, constructed by means of sequential composition, (non-)deterministic choice composition, parallel composition and interrupt composition, and*

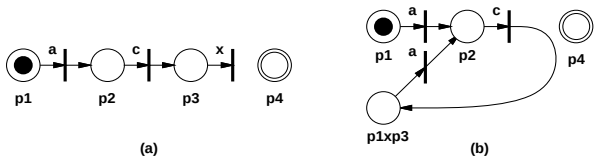


Figure 6: Petri net representations (a) $\mathcal{PT}\ (F(x))$ (b) $\mathcal{PT}\ (\mu x.F(x))$

$\mathcal{PT}\ (F(x)) = \langle P, A, F, M_0, E\rangle$. $\mathcal{PT}\ (\mu x.F(x))$ *is defined as:*

$\langle (P \setminus P_x) \cup (M_0 \times P_x), A \setminus \{x\}, F', M_0, E\rangle$

where

$$P_x = \bigcup_{(p,x,\emptyset)\in F} p$$

$$\begin{aligned} F' = \ & \{(p \setminus M_0 \cup (p \cap M_0) \times p_x, a, \\ & \quad q \setminus P_x \cup M_0 \times (q \cap P_x)) \mid a \neq x \\ & \quad \wedge (p, a, q) \in F \wedge (p_x, x, \emptyset) \in F\} \\ & \cup \{(p, a, q \setminus P_x \cup M_0 \times (q \cap P_x)) \mid a \neq x \\ & \quad \wedge (p, a, q) \in F\} \end{aligned}$$

3.3. Translation of Process Calculus expressions

The translation of a (complex) process expression into a Petri net can be defined recursively as follows. We start at the bottom of the syntax tree by translating the "leaf" atomic actions to PE-nets, according to definition 3.5. As we go up in the syntax tree, we gradually build up the PE-net representations of the intermediate subexpressions. This is illustrated in the example of Figure 7. Consider

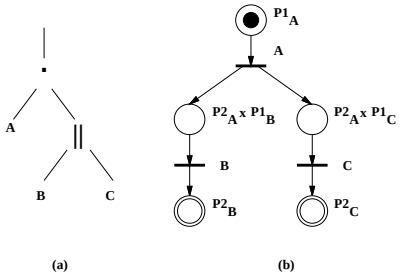


Figure 7: (a) Syntax-tree of $A.(B\|C)$ (b) $\mathcal{PT}\ (A.(B\|C))$

the process expression $A.(B\|C)$. This expression has the simple syntax tree shown in Figure 7(a). Starting at the bottom of the tree, the leaf actions are A, B and C with corresponding PE-nets $\mathcal{PT}\ (i)$ = $\{\{p1_i,p2_i\},\{i\}, \{(\{p1_i\},i,\{p2_i\})\},\{p1_i\},\{p2_i\}\}$ for $i \in \{A,B,C\}$. As we go up in the syntax tree, we first encounter the parallel composition operator, and we can compute $\mathcal{PT}\ (B.C)$ as an intermediate result. Next, as we go up, we arrive at the sequential composition operator, and use the intermediate result to compute $\mathcal{PT}\ (A.(B\|C)$, which is depicted in Figure 7(b).

4. Verification Framework

The techniques presented in this paper have been implemented in a tool, called JULIE, which consists of about 9000 lines of C code. The tool starts from the process calculus, applies a syntax-directed mapping scheme to translate between process expressions and Petri nets and performs an efficient analysis on the resulting Petri nets (e.g. checks for deadlocks, liveness properties, etc). Currently, we are developing an automated translation between heterogeneous C-VHDL specifications, as they are used in the CoWare^{TM}data model, and our process calculus. Because the overall mapping is semantics preserving, the analysis technique can be applied to validate the various intermediate specifications of the design process.

The analysis technique itself, called *generalized partial-order analysis* [18], tackles the two primary sources of combinatorial explosion that may occur in conventional reachability analysis. The first source is due to concurrently enabled actions for which standard analysis requires enumerating all possible orderings. This problem can be avoided by e.g. applying existing partial-order techniques where only one interleaved sequence needs to be analyzed for deadlock and liveness checks [15]. The second source is due to concurrently marked conflict places. This problem is solved by a generalized partial-order method which explores simultaneously concurrently enabled conflicting paths. The technique is based on a modified representation of markings to distinguish the different conflicting paths and can achieve an exponential reduction in algorithmic complexity.

5. Case study

To assess the viability of our approach we performed an extensive case study. More specifically, we experimented with the design of a Quadrature Amplitude Modulation (QAM) modem, integrating both a sender and a receiver section. The block diagram of the modem design is depicted in Figure 8. In the sender part (bottom square) the data to be sent is first formatted into a stream of alternating *I-data* and *Q-data*, each three bits wide. Then, this data is discretely levelized by the slicer and sent to the modulation block, which multiplies the I-data and the Q-data by orthogonal carriers. The middle square of Figure 8 represents the test bench including the channel model and the user interface. The upper square represents the receiver part. A tracking module is needed to derive "a local copy" of the carrier frequency, as well as an n-taps adaptive equalizer for correcting channel induced distortion. The de-slicer and symbol extraction block perform the inverse operations of their sender counterparts.

The QAM modem was modeled using the CoWare^{TM}environment [3]. The implementation target was a heterogeneous single-chip solution; because the (de)slicer and symbol creation (extraction) modules are dependent on

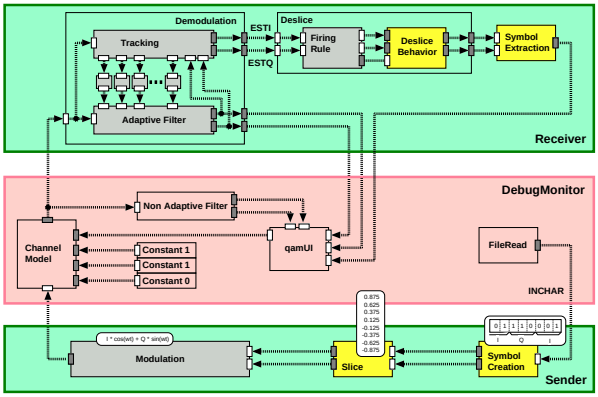


Figure 8: Block diagram of the QAM modem

the characteristics of the data, these modules were chosen to be implemented in software on an ARM-7 RISC processor. The other modules were specified in VHDL to be implemented in application-specific hardware. Simulations and formal verification using the techniques presented in the paper, revealed that the heterogeneous C-VHDL specification was indeed deadlock free.

To implement the communication between the hardware and the software, the ARM-7 processor boundary has to be crossed. This is shown in Figure 9 for the communication between the tracking module of the demodulator and the de-slicer. To realize the software drivers and the hardware in-

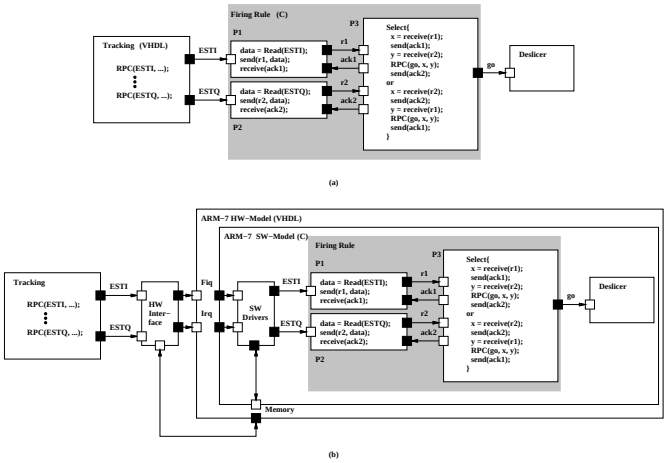


Figure 9: (a) Simplified view of communication between tracking module and de-slicer; (b) Implementation scheme of communication

terface, as depicted in Figure 9(b), the Symphony toolbox[3] has a number of I/O scenarios to select from. An I/O scenario (e.g. memory mapped I/O, interrupt driven I/O) consists of a software driver and a hardware counterpart for implementing a specific channel type on a particular processor. The Symphony toolbox assigns I/O scenarios on a per-channel basis. The I/O scenarios are designed to work correctly in a stand-alone operation mode. However, when assembled to a complete hardware/software interface, deadlocks may be introduced for a particular combination of I/O scenarios. Exhaustively verifying all these combinations by simulation is very time consuming as a new compilation step is required for every investigated I/O-configuration. Selecting the right I/O-configuration simply by inspection is quasi impossible, as the "real" problem instance is far more complex than what is presented here. Our verification approach formally diagnoses the cause of a possible deadlock, and provides the necessary feedback to select the right I/O scenario combination(s). To get an insight in what this means, an example is worked out below.

Consider the following I/O-configuration. The symbol creator of the sender part receives incoming samples from the INCHAR channel via software polling. Processes $P1$ and $P2$ of the de-slicer (see Figure 9(a)), on the contrary, are assigned to the FIQ and IRQ interrupt routines of the ARM-7, respectively. The corresponding hardware interface does not wait for the interrupt routines to complete; they are only "triggered".

Without loss of generality, we focus on the communication between the tracking module and the de-slicer. This can be modeled in our process calculus as follows:

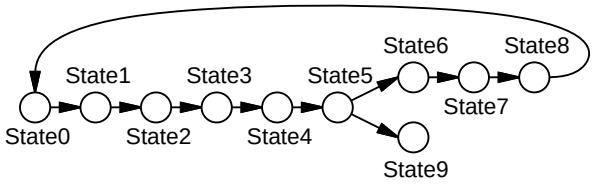
```
/* Software */
P1 = r1.ack1;
P2 = r2.ack2;
P3 = ((r1.ack1.r2.Deslice.ack2) +
      (r2.ack2.r1.Deslice.ack1)).P3;
SOFTWARE = P1 || P2 || P3;
/* Software drivers */
IRQ  = P1; /* P1 -> IRQ routine */
FIQ  = P2; /* P2 -> FIQ routine */
/* Main Body */
MAIN = ; /* Empty ! */
/* Processor Model of ARM-7 */
ARM = (MAIN > (IRQ_B.IRQ.IRQ_E))
                          > (FIQ_B.FIQ.FIQ_E);
/* Hardware Interface */
EST_I = FIQ_B; /* I/O-scenario fiq_trigger */
EST_Q = IRQ_B; /* I/O-scenario irq_trigger */
/* Tracking block of Demodulation */
TRACKING = EST_I.EST_Q.TRACKING;
SYSTEM = SOFTWARE || ARM || TRACKING;
```

The suffixes _B and _E denote the triggering and the completion of the interrupts involved. FIQ and IRQ represent the (calling of) the respective interrupt routines themselves. The (simplified) processor model of the ARM-7 models that the FIQ-interrupt has a higher priority than the IRQ-interrupt; in other words, the former interrupt can preempt the latter. If we run this example through JULIE we get the following reachability graph.

```
> Reach(SYSTEM);
Deadlock in State9  !
```

[3] Symphony is part of CoWare™ and is responsible for the hardware/-software interfacing and system architecture co-synthesis and integration problems.

```
STATE TRANSITIONS:
{
( State0        { FIQ_B }       State1 )
( State1        { r2 }          State2 )
( State2        { ack2 }        State3 )
( State3        { FIQ_E }       State4 )
( State4        { IRQ_B }       State5 )
( State5        { r1 }          State6 )
( State6        { Deslice }     State7 )
( State7        { ack1 }        State8 )
( State8        { IRQ_E }       State0 )
( State5        { FIQ_B }       State9 )
}
```



From state `State9` in the table above, no progress is observed which shows that this particular I/O configuration results in a deadlock. Indeed, after the first FIQ-routine has finished, the IRQ-routine is called. However, this latter routine, before being effective, can be preempted immediately by a new FIQ interrupt, which has a higher priority, and as result we arrive in a deadlock situation.

In Table 1, a number of possible I/O configurations are listed. As can be observed, verifying a particular configuration takes less than 1 second CPU time. From the table one can conclude that two configurations can result in a deadlock. For selecting the "optimal" I/O configuration, Symphony only pursues with the remaining alternatives. This result clearly shows that the process calculus and the subsequent Petri net translation, being able to model all communicating processes involved as well as the processor model of the ARM-7 itself, is indeed a powerful combination for analysis and synthesis of embedded systems.

Table 1: I/O configurations for implementing the hardware interface and the software drivers of Figure 9(b)

I/O scenarios			dead-	time
INCHAR	ESTI	ESTQ	lock	(sec)[4]
sw-polling	irq-trigger	fiq-trigger	NO	0.46
sw-polling	fiq-trigger	irq-trigger	YES	0.41
sw-polling	irq-wait	fiq-trigger	NO	0.36
sw-polling	fiq-trigger	irq-wait	NO	0.35
sw-polling	irq-trigger	fiq-wait	NO	0.44
sw-polling	fiq-wait	irq-trigger	YES	0.36
sw-polling	fiq-wait	irq-wait	NO	0.33
sw-polling	irq-wait	fiq-wait	NO	0.37

[4] Run-times measured on a HP 900/715 64 MB.

6. Conclusions

In this paper we proposed a formal model based on Petri nets for reasoning about the behavior of a concurrent system. Using this Petri net level, existing formal verification techniques can then be leveraged.

To provide a formal mapping between language-level constructs and Petri net operations, we proposed a process calculus as an intermediate model. We believe many of the existing process-based languages, used to specify heterogeneous systems can be first translated to "intermediate code" of our process calculus. We presented an elegant syntax-directed translation scheme for building Petri net representations starting from process expressions.

The viability of our approach was tested, on a "real-life" example, for which the results were very promising. We are currently investigating the inclusion of explicit timing constraints. In the future we plan to fully integrate the presented verification framework into the Symphony design flow.

References

[1] J.A. Bergstra and J.W. Klop. Algebra of Communication Process with Abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[2] I. Bolsens, H. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest. Hardware/software Co-Design of Digital Telecommunication Systems. *Proceedings of the IEEE*, 85(3):391–418, March 1997.

[3] CoWare, 2385 Santa Ana Street, Palo Alto, CA 94303, USA and Kapeldreef 60, B-3001 Heverlee

[4] L. Philips I. Bolsens and H. De Man. A Programmable CDMA IF Transceiver ASIC for Wireless Communications. In *IEEE Custom Integrated Circuits Conference*, May 1995.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification using Symbolic Model Checking. In *Proceeding of the 27th ACM/IEEE Design Automation Conference*, Orlando, June 1990.

[6] O. Coudert, J. C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machines without building their State Diagrams. In *Workshop on Computer-Aided Verification*, Rutgers, June 1990.

[7] G. G. de Jong and Bill Lin. A Communicating Petri Net Model for the Design of Concurrent Asynchronous Modules. In *Proceedings of the 31rd ACM/IEEE Design Automation Conference*, June 1994.

[8] B. Gyselinckx, L. Rijnders, M. Engels, and I. Bolsens. A 4*2.5Mchip/s Direct Sequence Spread Spectrum Receiver ASIC with Digital IF and Integrated ARM6 Core. In *IEEE Custom Integrated Circuits Conference*, May 1997.

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

[10] R. Milner. *Communication and Concurrency*. Englewood Cliffs, NJ, Prentice Hall, 1981.

[11] International Standards Organization. *LOTOS: a Formal Description Technique based on the Temporal Ordering of Observational Behavior*. ISO, IS 8807, 1989.

[12] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri Net Analysis using Boolean Manipulation. *Lecture Notes in Computer Science 815, Springer Verslag*, pages 416–435, June 1994.

[13] J.L. Peterson. *Petri Net Theory and Modelling of Systems*. Prentice Hall, 1981.

[14] D. Taubner. Finite Representations of CCS and TCSP Programs by Automata and Petri Nets. *Lecture Notes in Computer Science 369*, June 1989.

[15] A. Valmari. A Stubborn Attack on State Explosion. In *$2^{n\,d}$ Workshop on Computer Aided Verification*, pages 156–165, New Brunswick, NJ, June 18-21 1990.

[16] K. van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. Cambridge International Series on Parallel Computation, 1993.

[17] R. van Glabbeek and F. Vaandrager. Petri Net Models for Algebraic Theories of Concurrency. *Proc. of PARLE II, Lecture Notes in Computer Science 259*, pages 224–242, 1987.

[18] S. Vercauteren. Efficient Detection of Deadlocks Using Generalized Partial Order Analysis. *IMEC Internal Technical Report, Reference: IMEC/VSDM/97Mar/Deadlock/gpo.ps*, April 1997.

[19] S. Vercauteren and Bill Lin. Hardware/software Communication and System Integration for Embedded Architectures. *Design Automation of Embedded Systems, Kluwer Academic Publishers*, 2:1–24, 1997.