A Communicating Petri Net Model for the Design of Concurrent Asynchronous Modules

*Gjalt G. de Jong *, Bill Lin*

IMEC Kapeldreef 75, B 3001 Leuven, Belgium

Abstract *Current asynchronous tools are focussed mainly on the design of a single interface module. In many applications, one must design interacting interface modules that potentially communicate in complex and intricate ways. When designing communicating asynchronous modules, several difficult problems arise. First, even if each individual module can be synthesized correctly, according to the environmental assumptions specified for that module, the composition of the communicating modules may not work properly. Thus, one needs to have a way to model how the modules interact with each other, and to verify that their cooperation is consistent. In addition, means should be provided for communication and synchronization at a level higher than signal transitions, and for exploiting the communicating nature of these modules in optimization. This paper proposes a communicating Petri net model for describing communicating asynchronous modules. Each module is modeled by means of a labeled Petri net that extends the widely used Signal Transition Graph model by providing an abstract synchronization mechanism based on rendez-vous semantics. This enables the designer to specify high-level communication as well as low-level details such as signal transitions. Abstract synchronization events are expanded automatically to low-level handshake signals. We have developed a new algebra for communicating Petri nets that is applicable to general Petri nets, involves no unfolding, and defines hiding as generalized net contraction. We have developed methods based on this formal algebra that can be used to manipulate communicating interface modules, to verify their consistency, and to use them as a basis for optimizations.*

1. Introduction

Advances in the field of VLSI offer today's system designers a wide range of implementation technologies to choose from. Examples include custom ASICs, FPGAs, DSPs, micro-processors, memories, system bus modules, and other off-the-shelf components. These different hardware and software modules can be combined to create complex heterogeneous software/hardware systems, packaged possibly using various PC boards and MCM technologies. These modules often interact in very complex ways. Examples of such system-level designs can be found in virtually every digital microelectronics application domain: e.g. telecommunications, computers, multi-media, and automotive. With CAD tools maturing for component design, the design bottlenecks are rapidly shifting from the component level to the system level. A key problem to solve in system-level design is I/O interfacing, which must be addressed in order to integrate system components that often use incompatible I/O protocols to communicate with their environments.

For the interfacing problem, the existing literature describes a number of approaches to it [1, 2, 3, 5, 6, 9, 10, 11]. Among them, methods based on the Signal Transition Graph (STG) model [3], have captured wide attention. However, most work based on this model is focussed mainly on the specification and design of a *single* interface circuit; *concurrent communicating interfaces* are not dealt with. A model in which concurrent processes can be specified in a concise way is needed as system interfaces are naturally expressed as separate modules, and in which the individual systems themselves are also described separately. Also, the implementation may force a separate physical implementation. Real-life designs show that asynchronous system designs are inherently distributive and involve highly interactive controllers. While work based on Hoare's CSP model have addressed some problems with communicating modules [2, 6], these methods are mainly based on syntax-directed translation techniques, which have not shown the same degree of automation and optimization as their STG based counterparts.

When designing communicating asynchronous interface modules, a number of difficult problems arise. A major issue is correctness. Although current methods guarantee that each separate block is synthesized correctly, they *do not* guarantee the global correctness of the composed system. To reconcile such problems, one needs to have ways to model how the interface modules interact with each other and to verify that their cooperation is consistent. For this, we propose a high-level communicating Petri net model, called *Communicating Interface Processes* (CIP) for describing communicating system interfaces. Each individual interface component is modeled by means of a labeled Petri net that extends the widely used STG model by providing an abstract synchronization mechanism based on *rendez-vous* semantics. This enables the designer to model how the concurrent interface processes cooperate and to specify high-level communication as well as low-level details such as signal transitions. Abstract synchronization events are expanded automatically to low-level handshake signals at a later stage, which can help to avoid potential communication *mismatches* that often occur with signal-level interactions. Since these events are expanded to a synchronization mechanism, correctness is ensured.

Accordingly, we have developed a *new algebra* on this model of abstract communication graphs to describe and *formally reason* about manipulations from synthesis and analysis points of view. This algebra is used as a formal framework to develop methods to manipulate communicating interface modules, to verify their consistency, and to use them as a basis for optimizations offered by the communicating nature of the specification. These methods all operate at the Petri net level, which avoids potential state space explosion problems encountered by state based techniques.

The remainder of the paper is organized as follows. Section 2 reviews the basics of Petri nets and STGs. Section 3 describes the CIP model and indicates how abstraction synchronization events are expanded to low-level handshake signals. Section 4 presents a rigorous exposition of our new algebra for general Petri nets. This includes the definition of all the operations that can be performed on nets. Section 5 describes how this general algebra can be applied to synthesis of communicating interface modules. Section 6 describes the application of our approach on a protocol translation module design example.

2. Preliminaries

We first summarize the basic concepts of Petri nets and Signal Transition Graphs (STGs) for describing a *single* interface process. These concepts are necessary in order to build up our framework for specification and synthesis of *communicating interfaces*.

* sponsored by the EC under HCM contract ERBCHBGCT920056

2.1 Labeled Petri nets

A labeled Petri net is a Petri net [8] in which transitions are labeled by actions.

Definition 2.1 *A **labeled Petri net** N is a tuple $(A, P, \rightarrow, M_0)$ with A a set of action labels, P a set of places, $\rightarrow \subseteq 2^P \times A \times 2^P$ a transition relation, and $M_0 : P \rightarrow \mathbb{N}$ an initial marking. (where $\mathbb{N}$ is the set of natural numbers).*

Besides the structure of Petri nets, there is also an associated dynamics. A *state*, or *marking*, is a mapping of the places to the natural numbers $P \rightarrow \mathbb{N}$, indicating the number of tokens in a place.

Definition 2.2 *Each transition (p, a, q) can 'fire' in a state M iff $\forall p' \in p : M(p') > 0$. The **firing** of a transition leads to the next state M' defined as:*

$$M'(p') = \begin{cases} M(p') - 1 & \text{if } p' \in p \backslash q \\ M(p') + 1 & \text{if } p' \in q \backslash p \\ M(p') & \text{otherwise} \end{cases}$$

Such a state transition is denoted as (M, a, M'). Given a Petri net N, the *reachability graph* of N, denoted as $RG(N)$, is the (reflexive) transitive closure of the above next-state relation. The nodes of the reachability graph represent the reachable state space of the net, whereas the edges (M, M') are labeled with the action a of the transition (p, a, q) which must be executed in M to reach M'.

In this paper, only finite and bounded nets are considered. Bounded nets are nets in which for every state, all places have a bounded number of tokens. Bounded nets are characterized by having a finite state space. Safe nets are nets in which each place has at most 1 token.

2.2 Signal transition graphs

Signal transition graphs (STGs) have been proposed by Chu [3] for synthesis of asynchronous circuits, e.g. the design of interfaces or protocol converters. It is an *interpreted* labeled Petri net.

Definition 2.3 *Let $S = I \cup O$ be a set of signal names with disjoint sets I and O, which are the input respectively the output signals. A **classical STG** is a strongly-connected live and safe labeled Petri net $(A, P, \rightarrow, M_0)$ with labels $A = S \times \{+, -\} \cup \{\varepsilon\}$. A transition s^+ denotes a rising transition for signal s, whereas s^- denotes a falling transition. ε denotes a dummy transition.*

With this definition, an STG may be a general Petri net. The following types of extensions to this classical STG model have been proposed [9]:

- Boolean guards, i.e. predicates on signal levels, attached to outgoing arcs of places. Such a predicate must be true to execute the transition that the arc leads to.
- Other signal transitions, like *toggle*, *stable*, *unstable*, and *don't care*. These additional signal transitions are used as short-hand notations.
- Elimination of the live and safe requirements.

The *state graph* of an STG is defined similar to the reachability graph of the corresponding Petri net. Only the states are also labeled with an encoding, which is a bit-vector with a value 0 or 1 for each signal name. The encoding of a next state is identical to the encoding of the current state, except for the signal with which the transition is labeled. For a *consistent state assignment* for a transition (m, s^*, m'), $m(s)$ must be 0 and $m'(s) = 1$ for a rising transition s^+, and $m(s) = 1$ and $m'(s) = 0$ for a falling transition s^-. Similar requirements for consistent state assignment can be defined for the additional signal transitions defined in [9].

3. Communicating interface processes

To enable the designer to model how concurrent interface processes cooperate by specifying high-level communication as well as low-level details such as signal transitions, we extend the model of labeled Petri nets by introducing abstract synchronization events. This leads to the model of Communicating Interface Processes (CIP). Such an abstract event is for instance a *send* operation in a higher level specification, e.g. $a!$ in a CSP like language. However, it is not specified how such a high level event is implemented by a low-level signaling scheme, e.g. a four-phase handshaking protocol.

In a set of such communicating labeled Petri nets, we model the synchronization of abstract communication events by means of rendez-vous. This rendez-vous synchronization is ensured by the lower level signaling scheme to which this abstract communication event can be expanded, e.g. a 4-phase handshaking protocol. This expansion can be done automatically.

Definition 3.1 *A **CIP** is a graph (V, E) where each $v \in V$ is a labeled Petri net $(A, P, \rightarrow, M_0)$ connected with each other by edges $e \in E$. The edges are labeled by signal names $s \in S$ or by abstract communication channels $\sigma \in \Sigma$. The actions A of the labeled Petri nets are given by $A_S \cup A_\Sigma$ with $A_S = S \times \{+, -\} \cup \{\varepsilon\}$ and $A_\Sigma = \Sigma \times \{!, ?\}$.*

The actions A_S are the normal signal transitions, whereas the actions A_Σ model the abstract communication events. $a!v$ models the sending of a value v along channel, or edge, $a \in E$ of the CIP (V, E); $a?x$ represents the reception of a value along channel a.

An example of an expansion of an abstract communication event c is for example the handshaking sequence $r_c^+ \rightarrow a_c^+ \rightarrow r_c^- \rightarrow a_c^-$ for an output action $c! \in A_\Sigma$. For data values to be transmitted, different delay-insensitive encoding schemes can be devised. One example is the dual rail encoding. But instead of using $2n$ wires to model n-bit wide data-items, an encoding with m wires can also be used. An encoding c can be defined as the set of wires that must go high for this value. Such an encoding is correct when no encoding covers another. In that case, the abstract event $a!v$, i.e. sending a value v along a channel a in which the value v is represented by the code c, can be expanded to the sequence of low-level signal transitions given by (where ',' means concurrent execution) $(\cdots, r_j^+, \cdots) \rightarrow a^+ \rightarrow (\cdots, r_j^-, \cdots) \rightarrow a^-$ for all $r_j \in c$.

4. Petri net algebra

For the labeled Petri nets as defined in Section 2.1, different semantics can be defined, but they are all related to the reachability graph. A commonly used semantics is the trace semantics, which is defined by all the possible firing-sequences of the net. Thus all paths in the reachability graph are viewed as elements of the trace set.

Definition 4.1 *For a net N, the **set of traces** is defined as:*
$L(N) = \{a_1, a_2, \cdots | \exists M' : (M_0, < a_1, a_2, \cdots >, M') \in RG(N)\}$

The operators that are defined next are well-known process algebra operators, as for instance used in CCS and CSP. They are also defined for Petri nets, e.g. in [4]. Of these operators, the parallel composition and the hiding operator are the most interesting ones. For a circuit algebra with 'complex leaf' models (i.e. other than just single actions), these two operators and the rename operator are sufficient. But to be complete, also the 'do nothing' action is defined, just as the action prefix and the (possibly nondeterministic) choice operator.

The parallel composition operator models a rendez-vous as in CSP, and is described also in [7]. New in here is the way in which the hiding operator is defined. In all the currently known approaches, hiding means that all the transitions with the label to be hidden are renamed to a specially treated silent action, cf. the epsilon moves in automata theory. Here we propose a method in which the transitions with labels to be hidden are removed from the net, cf. the epsilon-closure operator for automata. This is a net contraction.

For *all* the operators as they are defined in this paper, it is *not* necessary to unfold the net(s), which is commonly done when these operations are defined in literature for nets. Also our approach is *not* restricted to safe nets, which also is a common restriction.

4.1 Action operators

Definition 4.2 *The **deadlock action** nil is represented by the single place net* $(\emptyset, \{p\}, \emptyset, \{(p,1)\})$.

Proposition 4.1 $L(nil) = \emptyset$.

Definition 4.3 *Given a net N with a safe initial marking and an action a, **action prefix** for nets is defined as:*
$a.\,N = (A \cup \{a\}, P \cup \{m_0\}, \rightarrow \cup \{(m_0, a, M)\}, \{(m_0, 1)\})$
with $m_0 \notin P$ *and* $M = \{p \in P\colon M_0(p) \neq 0\}$.

Proposition 4.2 $L(a.\,N) = \{\varepsilon, a\} \cup \{a\}.\,L(N)$.

The definition given here applies only for nets with a safe initial marking. However, it can also be defined for general Petri nets, by keeping the original initial places as initial places, and using new output places for the action transition a. These new places must then be connected via a self-loop with the original initial transitions.

Definition 4.4 *Given a net N and action labels b and c, the **renaming** operator for nets is defined as:*
$rename(N, \{b \rightarrow c\}) = ((A \backslash \{b\}) \cup \{c\}, P, \rightarrow', M_0)$ *with*
$\rightarrow' = \{(p,a,q) | (p,a,q) \in \rightarrow \wedge\, a \neq b\} \cup \{(p,c,q) | (p,b,q) \in \rightarrow\}$

This definition of renaming can be extended in the natural way to renaming of sets of action names.

Proposition 4.3
$L(rename(N, \{b \rightarrow c\})) = rename(L(N), \{b \rightarrow c\})$.

4.2 Choice

For the choice operator to be defined correctly for general Petri nets, a one-step unfolding is necessary by duplicating the initial transitions.

Definition 4.5 *Let N be a net with a safe initial marking. Let* P_0 *be new places* ($P_0 \cap P = \emptyset$) *and* η *a bijection between* P_0 *and the initial places of N which are defined by* $\{p \in P | M_0(p) \neq 0\}$. *The **root-unwinding** of a net is defined as:*

$(A, P \cup P_0, \rightarrow \cup \{(p,a,q) | p \subseteq P_0 \wedge (\mathrm{H}(p), a, q) \in \rightarrow\}, M_0')$,
where $\mathrm{H}\colon 2^{P_0} \rightarrow 2^P$ *is the component-wise extension of* η *to sets, i.e.* $\mathrm{H}(\{p_1, \cdots, p_n\}) = \{\eta(p_1), \cdots, \eta(p_n)\}$, *and*

$$M_0'(p) = \begin{cases} 0 & \text{if } p \in P \\ M_0(\eta(p)) & \text{if } p \in P_0 \end{cases}$$

Definition 4.6 *Let for* $i \in \{1,2\}$ $N_i = (A_i, P_i, \rightarrow, M_{0_i})$ *be two nets with* $P_1 \cap P_2 = \emptyset$. *Let* N_i' *be the root unwinding of* N_i *with* $P_{0_1} \cap P_{0_2} = \emptyset$. *The non-deterministic **choice** operator for nets is defined as:* $N_1 + N_2 = (A_1 \cup A_2,\ P_1 \cup P_2 \cup P_{0_1} \times P_{0_2},\ \rightarrow',\ M_{0_1}' \times M_{0_2}')$
where $\rightarrow' = \rightarrow_1 \cup \rightarrow_2 \cup \{(p \times P_{0_2}, a, q) | (p,a,q) \in \rightarrow'_1 \wedge p \subseteq P_{0_1}\}$
$\cup \{(P_{0_1} \times p, a, q) | (p,a,q) \in \rightarrow'_2 \wedge p \subseteq P_{0_2}\}$

Fundamentally, root unwinding is needed in case of cycles to the initial places and only for the initial places that are in cycles. 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. This is illustrated in Figure 1.

Although, the definition of the root unwinding of a net is given for nets with safe initial markings, it can also be stated slightly different, such that the following proposition also holds for general nets. This can be accomplished by keeping the initial places with their initial marking. The initial transitions must then be duplicated, as in the root-unwinding step, and added with an uninitialized sentinel place in a self-loop which is also an output place of the original initial transition.

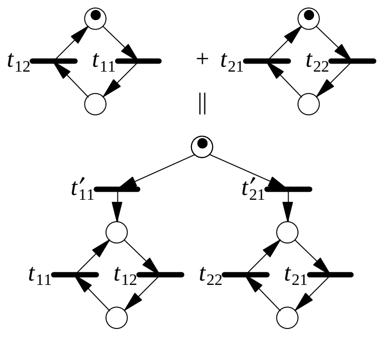

Figure 1. Example for choice with root-unwinding

Proposition 4.4 $L(N_1 + N_2) = L(N_1) \cup L(N_2)$

4.3 Parallel composition

All the proofs of the trace equivalence for all the operators defined so far are straightforward, since the reachability graph is changed accordingly. E.g. for the choice operator, this is implied by the combined reachability graph being the union of the two individual reachability graphs. For the parallel composition and the hiding operator, which will now be defined, this proof is less trivial.

In Petri nets, a transition is a kind of synchronization mechanism, since it can only fire if all input places have a token. In order to model parallel composition with a rendez-vous synchronization, it is sufficient to join the common transitions. Since more than one transition may be labeled with the same action, all combinations have to be considered.

Definition 4.7 *Let for* $i \in \{1,2\}$, $N_i = (A_i, P_i, \rightarrow, M_{0_i})$ *be two nets with* $P_1 \cap P_2 = \emptyset$. *The **parallel composition** of nets is defined as:*
$N_1 || N_2 = (A_1 \cup A_2, P_1 \cup P_2, \rightarrow', M_{0_1} \cup M_{0_2})$
where $\rightarrow' = \{(I,a,O) \in \rightarrow_1 \cup \rightarrow_2 | a \notin A_1 \cap A_2\}$
$\cup \{(I_1 \cup I_2, a, O_1 \cup O_2) | a \in A_1 \cap A_2 \wedge (I_i, a, O_i) \in \rightarrow_i\}$

To prove that this construction is equivalence preserving under trace semantics, we first need to define parallel composition of traces. Since we use a rendez-vous synchronization on common actions, also the composition of traces is defined in this way.

Definition 4.8 *Let* t_1 *and* t_2 *be traces defined on alphabets* A_1 *and* A_2 *resp.* $t_1 || t_2 = \{t \in (A_1 \cup A_2)^* \mid \forall i \in \{1,2\}\colon project(t, A_i) = t_i\}$.

Note that this set can be empty, e.g. for $a.\,b.\,c || c.\,a.\,b$. If this set is non-empty, the traces are said to be synchronizable and this set consists of all possible shuffles.

Definition 4.9 *For two trace languages* L_1 *and* L_2, *their **parallel composition** is defined as:* $L_1 || L_2 = \{t_1 || t_2 | t_1 \in L_1 \wedge t_2 \in L_2\}$.

Note that for prefix-closed (trace) languages, this set is also prefix-closed. Now we can state:

Theorem 4.5 $L(N_1 || N_2) = L(N_1) || L(N_2)$.

The proof of this theorem is straightforward from the definition of || on traces [7]. The reachability graph of $N_1 || N_2$ is the 'interleaved intersection' of the individual reachability graphs.

Figure 2 shows the parallel composition of $((a+b).\,c)* ||$ $(a.\,d.\,a.\,e)*$.

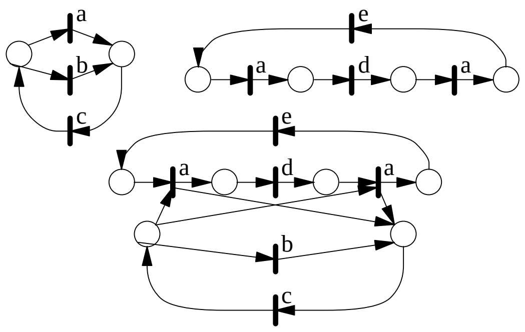

Figure 2. Parallel composition.

4.4 Hiding

Hiding is opposite to projection, in that all other signals than the specified ones are projected, i.e. for a language L on an alphabet Σ, $hide(L, a) = project(L, \Sigma \backslash \{a\})$. We now define a hide operator which satisfies $L(hide(N, a)) = project(L(N), A \backslash \{a\})$. We assume that the transition to hide has no self-loops. Otherwise, this would lead to a divergence, i.e. an unobservable self-loop known as livelock.

Definition 4.10 *Given a net N and a transition (p, a, q), the* ***contraction*** *of a transition is defined as:*

$$hide(N, (p, a, q)) = (A, (P \backslash p) \cup (p \times q), \rightarrow', M_0') \text{ where}$$

$$M_0'(p') = \begin{cases} M_0(p') & \text{if } p' \notin p \\ M_0(p) & \text{if } p' \in p \times q \end{cases}$$

$$\begin{aligned} \text{and } \rightarrow' = & \{(p', a, q') | (p', a, q') \in \rightarrow \wedge (p' \cup q') \cap (p \cup q) = \varnothing\} \\ & \cup \{(\mathrm{H}(p'), a, q') | (p', a, q') \in \rightarrow \wedge p' \cap p \neq \varnothing\} \\ & \cup \{(p', a, \mathrm{H}(q')) | (p', a, q') \in \rightarrow \wedge q' \cap p \neq \varnothing\} \\ & \cup \{(p', a, q') | (p', a, q') \in \rightarrow \wedge p' \cap q \neq \varnothing\} \\ & \cup \{(p \times q, a, q' \cup (q \backslash p')) | (p', a, q') \in \rightarrow \wedge p' \cap q \neq \varnothing\} \end{aligned}$$

where H *is the renaming function of the input places p of the transition to hide to the corresponding product places. I.e.* $\mathrm{H}: 2^P \rightarrow 2^{(P \backslash (p \cup q)) \cup (p \times q)}$ *is defined as:*

$$\mathrm{H}(\{p_1, \cdots, p_n\}) = \{p_i | p_i \notin p \cup q\} \cup \bigcup_{\{p_i | p_i \in p\}} \{p_i\} \times q \cup \bigcup_{\{p_i | p_i \in q\}} p \times \{p_i\}$$

Informally, this procedure can be described as (where $t = (p, a, q)$ the transition to hide):

1. add new places $p \times q$
2. duplicate each successor transition of t
3. connect the new places to all successor transitions
4. replace all occurrences of a $p' \in p$ in the transition relation $\rightarrow$ with the places (p', q') for all $q' \in q$
5. add the places $q' \in q$ to the postset of the successor transitions which did not have this place originally in their preset.
6. delete this transition t from the net.

The result of hiding the transition in Figure 3(a) is shown in Figure 3(b). Figure 3(c) shows the result of hiding the same transition but in a marked graph in which transition a, d, e, f, h, j, k and l are not present. In this figure, the new places $p \times q$ are combined into places $\{p'\} \cup q$ for all $p' \in p$. This is in general allowed, but complicates the proof somewhat. The first set of $\rightarrow'$ are the transitions which are not adjacent to the transition to hide; the second set are the transitions like e and f in Figure 3, whereas the third set are the transitions like a, b, c and d (these are given by rule 4); the fourth set are the 'duplicates' of the successors of the transition to hide (rule 2; e.g. the transitions g', h', i' and j' in Figure 3); the last set are the successors of the transition to hide but which have now all the new places as inputs and having also some output places of

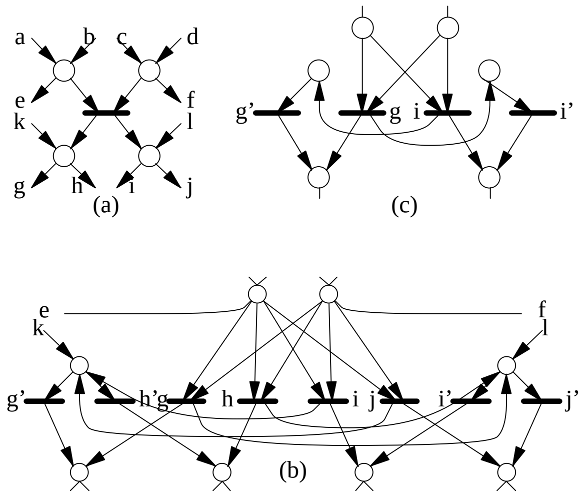

Figure 3. Hiding

this transition as outputs (rules 4 and 5; e.g. the transitions g, h, i and j).

The collapsing of the input and output places of the transition to hide is clear, since this in fact models the hiding operator: whenever there is a token in an input place, it might under certain conditions be considered as a token in an output place. The successor transitions are duplicated to preserve all the possible choices and conflicts in the original net. E.g. whenever a token is considered to be in one of the output places of the transition to be hidden, the other output places should also be considered in this way. In that case, a token must be removed from *all* the input places, such that no new conflicts or choices may occur later in the net after hiding, for instance by a new (partial) enabling of this transition. This is modeled by the curved incoming arcs of the conflictive places of the transition g', h',i' and j' in Figure 3.

The hiding of an action label a is defined as the successive application of the above definition of hiding for all transitions $(p, a, q) \in \rightarrow$, and then by deleting a from the action labels of N.

Proposition 4.6 *The final net $hide(N, a)$ is independent on the order in which the transitions (p, a, q) are hidden.*

Theorem 4.7 $L(hide(N, a)) = hide(L(N), a)$

The proof is a tedious case analysis by taking care of all possible choices and conflict situations and their corresponding partially enablings. It follows the lines of the motivation given above.

This operation can be simplified for special cases. For example, for transitions with a single and conflict-free input place and a single and choice-free output place, the operation is just a collapsing of these two places.

5. Synthesis of Communicating Interfaces

In this section, we will explain how the algebra of Section 4 can be used in the synthesis and verification of communicating interfaces.

5.1 Circuit algebra

A circuit algebra is an algebra on behavioral structures, like the Petri net algebra defined in the previous section, extended with the notion of inputs and outputs. It is meant as a specification formalism for the behavior of a composite, e.g. hierarchical, system. For a circuit algebra, usually only composition hiding and renaming are defined. When two systems are composed, they synchronize in their common signals. If two systems have input signal names in common, these signals are assumed to be inputs of both, but it is not allowed that there are common output signals. Internal signals are considered as outputs, which may be hidden. If there are no signals in common, the composition yields the concurrent behavior of both.

Communicating interface processes can be mapped to a communicating STG network by expanding all abstract communicating events to labeled transition structures. Signal transition graphs (STGs) are used in asynchronous system design [3], e.g. the design of interfaces or protocol converters. Usually an STG is a restricted subclass of Petri nets, e.g. the marked graphs or the free-choice nets. Therefore, the operations for general Petri nets can directly be applied to STGs, especially since by Definition 2.3 of classical signal transition graphs, an STG may be a general Petri net. However, important systems like arbiters cannot be modeled in these subclasses of marked graphs and free-choice nets. For this, general Petri nets should be allowed for an STG. Many properties can be checked structurally for marked graphs and free-choice nets in polynomial time, but which require exponential time for general Petri nets [8].

In Section 2.2 already some extensions to STGs are presented, as the introduction of boolean guards and the use of more signal transitions like *toggle*($\tilde{}$), *stable*(s), *unstable*($\#$) and *don't care*(x). As these additional signal transitions are a short-hand notation, they do not matter for the Petri net algebra defined here. To hide a signal s means to hide all signal transitions for this signal. The parallel composition synchronizes the common signals with respect to their signal transition type. To incorporate boolean guards, only the hiding operation requires a minor change. A boolean guard on an incoming arc of a transition to be hidden, must be propagated to the corresponding arcs in the resulting net; similarly for the outgoing arcs. For the parallel composition operator, boolean guards remain attached to the same arcs.

This leads to the following circuit algebra $C = (I, O, N)$ for communicating interface processes with $I(O)$ the set of input (output) signals, and N a labeled Petri net describing the behavior of the interface process. We then have (with $A \subseteq O$):

$C_1 \| C_2 = (I_1 \cup I_2 \backslash (O_1 \cup O_2), O_1 \cup O_2, N_1 \| N_2)$
$hide(C, A) = (I, O \backslash A, hide(N, A))$

5.2 Compositional synthesis

An application of this algebra is in synthesizing asynchronous systems, which can be modeled by STGs. When the environment of a (sub)system is known, its behavior may be reduced by using this knowledge. Suppose that it is known that a system M_1 is composed with a system M_2, of which both specifications are given. The individual STGs may contain behavior which will never be executed in the composition. Instead of synthesizing M_1 and M_2, it may be advantageous to synthesize $hide(M_1 \| M_2, A_2 \backslash A_1)$ and $hide(M_1 \| M_2, A_1 \backslash A_2)$ instead. The resulting STGs have a smaller behavior in terms of their traces, as the following theorem shows.

Theorem 5.1 *For $i \in \{1, 2\}$, $project(L(M_1 \| M_2), A_i) \subseteq L(M_i)$.*

This can be proved immediately from the definitions of the projection and inverse projection operators on traces. The following propositions for our Petri net algebra are valid.

Proposition 5.2 *The class of safe Petri nets is closed under all operations.*

Proposition 5.3 *The class of live Petri nets is closed under all operations except parallel composition.*

Thus even when the individual nets are live, the composed net need not to be live. This is due to the fact that one net restricts the behavior of the other net as Definition 4.8 shows.

Thus for compositional synthesis, only the common transitions can be non-live. The removal of these dead transitions, can be done in polynomial time and space for marked and free-choice nets [8]. By the following proposition, this means that this check is also polynomial on the composed net $N_1 \| N_2$.

Proposition 5.4 *Marked graphs are closed under action prefix, renaming and parallel composition.*

For free-choice nets, there is an analogous proposition, when the synchronization transitions are choice-free.

In the composition, the transitions that are not common to both, remain as concurrent as they are when the nets are viewed separately. Thus the net can be viewed as a net that is partitioned at the synchronizing transitions, and in which each partition consists fully of transitions of an individual net. Hiding of the non-common transition can then maximally lead to a duplication of the synchronization transitions. Thus although, by Theorem 5.1, the behavior of $hide(M_1 \| M_2, A_2 \backslash A_1)$ is smaller (thus yielding reduced implementations), the STG itself is not necessarily smaller. However, due to the cross-product and the duplication of the synchronizing transitions, many of them will be dead and can be eliminated as mentioned before.

5.3 Verification

An important issue in asynchronous system design is receptiveness. There exists a semantic distinction between the input and the output signal names. The inputs of a system are controlled by its environment, while the system determines its outputs autonomously. A system must therefore be receptive in its inputs, i.e. whenever the environment generates an input to the system, the system must be ready to accept it , i.e. synchronize with it. Note that synthesizing the composed net results in a correctly behaving net, since the synchronization is guaranteed. However, if the STGs are synthesized individually, just 'abutting' them may yield erroneous behavior, as one system is not receptive in its inputs.

Note that if N_1 never makes an output action when N_2 is not ready to accept it (and vice versa), then the above defined composition operator is receptive. In order to verify this conformance, we have to do an additional check after the composition.

Proposition 5.5 *Let N_1 and N_2 be strongly-connected live nets with the transitions $(p_i, a, q_i) \in \rightarrow_i$ $(i \in \{1, 2\})$ as the single common transition and where a is an output signal in N_1 and an input for N_2. A failure can occur iff there exists a marking in the composed net $N_1 \| N_2$ such that all the places in p_1 are marked, but not all the places in p_2 are marked.*

This means that a is enabled in N_1 but not in N_2. Since the individual nets are strongly-connected and live, there will exist a marking in which all input places of both transitions have tokens. But we are looking for the transition to be completely enabled in one net, while it is only partially enabled in the other net. From this, the following can be derived and which states the only condition when N_1 can make an output action when N_2 is not ready to accept.

Proposition 5.6 *If in the composed net $N_1 \| N_2$, the condition of Proposition 5.5 is not satisfied, then the compose operator is correct; otherwise, a failure is guaranteed to be possible.*

Note that this proposition does not state anything about finding all failures due to non-receptiveness, but only that at least one such a failure exists. This is due to the fact that a failure of one transition may mask the correctness in terms of receptiveness of other transitions. But there will always be a 'first' one (in terms of unfolding the net).

Thus besides the check of liveness of the composed graph, we have an additionally check for the live synchronization transitions. Let $(p_1 \cup p_2, a, q_1 \cup q_2)$ be such a transition, with $(p_i, a, q_i) \in N_i$ $(i \in \{1, 2\})$ and which is an output transition in N_1 and an input in N_2. To check for receptiveness, we have to verify for such a transition the live-safeness of a marking $M\backslash p'$ for some subset $p' \subsetneq p_2$ which is defined as:

$$M(p) = \begin{cases} 1 & \textit{if } p \in p_1 \cup q_2 \\ 0 & \textit{otherwise} \end{cases}$$

Recall that STGs are usually strongly-connected live-safe marked graphs, which are closed under parallel composition.

Theorem 5.7 *For strongly-connected live-safe marked graphs, the check for receptiveness for the parallel composition operator can be done structurally on the net in polynomial time and space.*

Note that for this receptiveness check, we may not do it on $hide(N_1, A_1\backslash A_2)||hide(N_2, A_2\backslash A_1)$ since then information is lost whether the synchronization transitions are reached via internal transitions or not. However, since the composed net $N_1||N_2$ can be viewed as being partitioned in partitions consisting fully of transitions of an individual net, the hiding operator can be refined in such a way to $hide'$, that not all transitions are contracted, but *one* dummy transition ε remains. In that case, the receptiveness check for general Petri nets may be restricted to the check on $hide'(N_1, A_1\backslash A_2)||hide'(N_2, A_2\backslash A_1)$ which, under the assumption that the reachability analysis is tractable for the individual nets, is tractable.

6. Implementation and an example

We have a prototype LISP implementation of the proposed concepts. In this section, we illustrate our approach and concepts on a protocol translation design example. It is a simplified variation of an I^2C protocol conversion module. The block diagram is shown in Figure 4 and consists of a *sender*, a *protocol translator*, and a *receiver* block. The behavior of the *sender* is shown in Figure 5. It is shown as a *hierarchical* model for conciseness. The sender is responsible for converting a transition signaling protocol from the sender side to a 4-phase protocol seen by the protocol translator. The sender side can issue four commands: *sec*, *reset*, *send0*, and *send1*. These four commands are indicated by a *toggle*(˜) on the corresponding wire with the same name. The environment will only issue one command at a time. Each toggle command gets translated into a 4-phase command by causing two wires to go high.

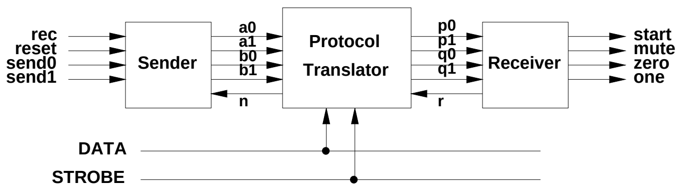

Figure 4. Block diagram of protocol translator design.

The conversion of the signals is shown in Table 1(a). The signals $a0$ and $b0$ will both make a $0 \rightarrow 1$ transition to indicate a *rec* command to the protocol translator. To make sure the protocol translator has read the command, a signal n is used. After $a0$ and $b0$ are set high, the protocol translator will make n a $0 \rightarrow 1$ transition to acknowledge the command. Then $a0$ and $b0$ can *return-to-zero* $(1 \rightarrow 0)$, followed by a *return-to-zero* of n. This behavior is shown in Figure 5(b). The other three commands are similarly specified, as shown in Figure 5(c). The top-level behavior of the *receiver* block is shown in Figure 6. The receiver block is responsible for converting 4-phase commands into transition signaling commands in a reverse analogous manner as the sender block. There are four commands for the receiver of which the signal conversion is shown in Table 1(b): *start*, *mute*, *zero*, and *one*. The signal r is used for the 4-phase protocol.

The protocol translator itself is specified in Figure 7. Initially, it sends a *start* command to the receiver. Then it waits for a command

Table 1. Translation table. (a) Sender. (b) Receiver.

(a)

rec˜	a0+	b0+
reset˜	a0+	b1+
send0˜	a1+	b0+
send1˜	a1+	b1+

(b)

p0+	q0+	start˜
p0+	q1+	mute˜
p1+	q0+	zero˜
p1+	q1+	one˜

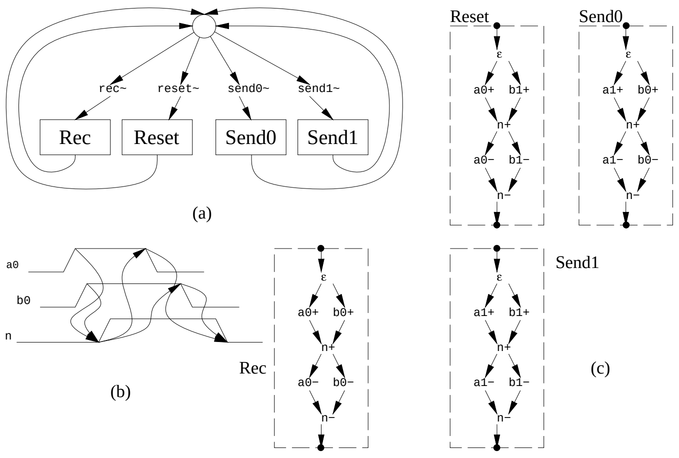

Figure 5. Sender protocol. (a) Top-level description. (b) Rec description. (c) Reset, Send0, and Send1 descriptions.

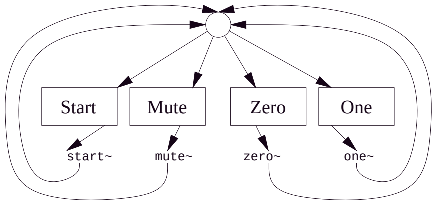

Figure 6. Receiver protocol. Top-level description.

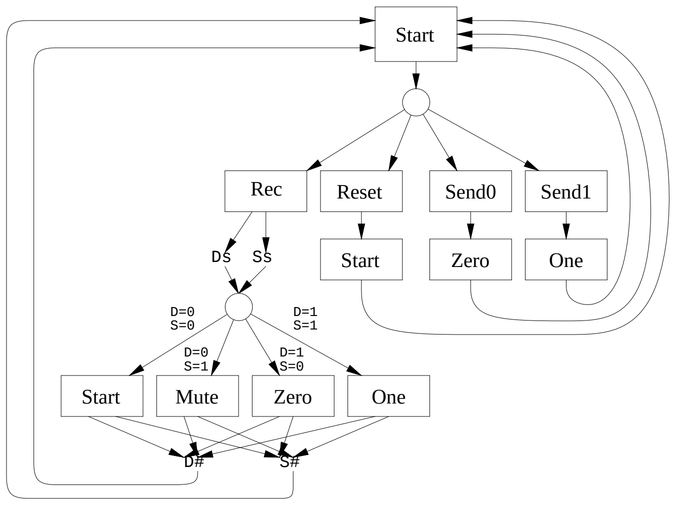

Figure 7. Protocol translator protocol.

from the sender. If the command is *reset*, *send0* or *send1*, then it simply sends the command *start*, *zero*, or *one*, respectively, to the receiver. Then it waits for the next command. If the input command is *rec*, then the *DATA* and the *STROBE* lines are expected to stabilize at either a 1 or a 0 value. Then depending on the values of these lines, either a *start*, *mute*, *zero*, or *one* command is sent. Then, the data and strobe lines can again become unstable, meaning that they can arbitrarily change values. So the behavior of the protocol translator depends on the values on the data and strobe lines.

If each of these STGs is synthesized correctly, then the global composition of them also works correctly in this case. This is because the behavioral assumptions on what the other modules can do are properly modeled in each individual STG. So in this case, the specification is consistent. However, one can provide an *inconsistent* specification that will not work correctly when composed together. In Figure 8, an *inconsistent* specification of the sender block is shown. The STGs for the cases *reset*, *send0* and *send1* are analogous to the *rec* STG. In this specification, the signals $a0$, $a1$, $b0$, and $b1$ can go high and low *without waiting* for the protocol translator block to *acknowledge*. Thus, it does not properly implement the 4-phase protocol. For example in the *rec* command, the sender is able to make both $a0-$ and $b0-$ transitions without waiting for the acknowledge $n+$ of the transitions $a0+$ and $b0+$. This can cause the protocol translator to malfunction because the circuit synthesized for the protocol translator had not accounted for this behavior.

This problem can be tackled in two ways using our approach. One is simply to avoid such problems by using abstract communication instead of signal-level communication, as described in Section 3. Alternatively, the verification and synthesis methods described in Section 5 can be used as follows to check that an inconsistency has occurred. Let N_{send} and N_{tr} be the STG for the sender and protocol translator block, respectively. Then compose them together to produce $\bar{N} = N_{send} \| N_{tr}$. Check on $\bar{N}$ for the

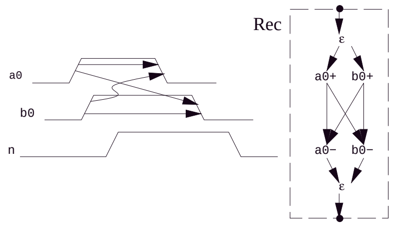

Figure 8. Inconsistent sender protocol.

source of inconsistency expressed in Propositions 5.5 and 5.6.

Now let's suppose that the sender block can only issue the commands *reset*, *send0*, and *send1*, but not *rec*, as shown in Figure 9(a). This means that the protocol block need not respond to the command *rec*; hence it can be greatly simplified. To build a *simplified* protocol translator block, we can use the Petri net algebra to *compose* the sender and the protocol translator STGs together, and then *hide* the signals not originally in the protocol translator block, i.e. $\bar{N}_{tr} = project(N_{send} \| N_{tr}, A_{tr})$. According to Theorem 5.1, this operation sequence will produce a new STG with more degrees of freedom. The simplified protocol translator block is shown in Figure 9(b). Using a similar approach, the simplified receiver block of Figure 9(c) is derived.

7. Concluding remarks

In this paper, we have discussed the problems that arise when designing communication interface modules. We have discussed correctness issues as well as optimization concerns. We have proposed a model of Communicating Interface Processes for modeling communicating system interfaces. It is a model based on communicating Petri nets, where each individual interface component is modeled by means of a labeled Petri net that extends the widely used Signal Transition Graph model with an abstract synchronization mechanism. This enables the designer to specify high-level communication as well as low-level details such as signal transitions. We have presented an algebra for communicating Petri nets that is applicable to general Petri nets, which involves no unfolding, and has hiding defined as generalized net contraction. We have also presented methods based on this formal algebra that can be used to manipulate communicating interface modules, to verify their consistency, and to exploit optimizations offered by the communicating nature of the modules.

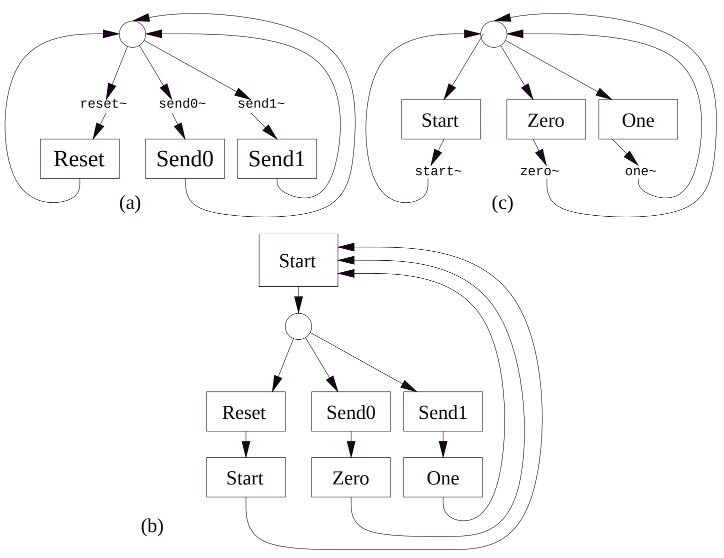

Figure 9. (a) Restricted sender block. (b) Simplified protocol translation block. (c) Simplified receiver block.

Acknowledgment — We are grateful to Charles Molnar for an interesting discussion on the topic.

References

[1] P.A. Beerel and T.H. Meng, "Automatic Gate-level Synthesis of Speed-Independent Circuits", *Proc. ICCAD*, 1992.

[2] K. van Berkel, J. Kessels, M. Roncken, R.W.J.J. Saeijs, and F. Schalij, "The VLSI-programming Language TANGRAM and its Translation into Handshake Circuits", *Proc. EDAC*, 1991.

[3] T.A. Chu, *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*, Ph. D. thesis, MIT, 1987.

[4] R. van Glabbeek and F. Vaandrager, "Petri Net Models for Algebraic Theories of Concurrency", *Proc. of PARLE*, LNCS 259, 1987.

[5] L. Lavagno, C.W. Moon, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Solving the State Assignment Problem for Signal Transition Graphs", *Proc. DAC*, 1992, pp. 568-572.

[6] A.J. Martin, "Programming in VLSI: From Communicating Processes to Self-Timed VLSI Circuit Synthesis", *Concurrent Programming*, ed. C.A.R. Hoare, Addison-Wesley, 1989, pp. 1-64.

[7] A. Mazurkiewicz, "Concurrency, Modularity, and Synchronization", *Proc. Math. Found. of Comp. Sci.*, LNCS 379, Springer Verlag, 1989, pp. 577-598.

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

[9] P. Vanbekbergen, C. Ykman-Couvreur, and B. Lin, "A Generalized Signal Transition Graph Model for Modeling Mixed Asynchronous/Synchronous and Arbitration Behavior", *Proc. Int. Workshop on Logic Synthesis*, 1993.

[10] V. Varshavsky, M. Kishinevsky, V. Marakhovsky, V. Peschansky, L. Rosenblum, A. Taubin, and B.Tzirlin, *Self-Timed Control of Concurrent Processes*, Kluwer Academic Publishers, 1990.

[11] K.Y. Yun and D. L. Dill, "Automatic Synthesis of 3D Asynchronous State Machines", *Proc. ICCAD*, 1992.