# Translations Between Textual Transition Systems and Petri Nets

Katerina Korenblat, Orna Grumberg, and Shmuel Katz
Computer Science Department
The Technion, Haifa, Israel
{pokozy,orna,katz}@cs.technion.ac.il
FAX: +972 4 822 1128 PHONE: +972 4 829 4322

**Abstract**

Translations between models expressed in textual transition systems and those expressed in structured Petri net notation are presented, in both directions. The translations are *structure-preserving*, meaning that the hierarchical structure of the systems is preserved. Furthermore, assuming non-finite state data manipulation has been abstracted out of the textual transition system, then translating one model to another and then back results in a model which is identical to the original one, up to renaming and the form of Boolean expressions. Due to inherent differences between the two notations, however, some additional information is required in order to obtain this identity. The information is collected during the translation in one direction and is used in the translation back.

Our translation is also *semantics-preserving*. That is, the original model and the translated model are bisimulation equivalent, assuming non-finite data abstraction. Thus, the translation preserves all temporal properties expressible in the logic CTL*.

The translations are both more generally applicable and more detailed than previously considered. They are shown both for individual modules, with a collection of transitions, and for a structured system, where modules are combined in different ways.
**keywords**: model translations, Petri nets, textual transition systems, structure and semantics preservation.

## 1 Introduction

In order to use different verification tools for different properties of a model, it has become common to translate among notations. Here we show how such translations can be done for the very different paradigms of textual transition systems as seen in [MP92] and a structured version of Petri nets, as seen in, e.g., [Rei98]. Each paradigm is translated both for a flat, simple version, and for a structured one. Rather than considering the translation as an internal prelude to activating a tool, we emphasize the possible optimizations that are possible in order to obtain a result natural in the target notation, and which has easy traceability back to the source. This is important, for example, in order to allow errors discovered in the target model to be traced back to the corresponding error in the source.

After briefly presenting the two notations for expressing models, we show our algorithms for translating from a textual transition system to a Petri net, and then for translations in the opposite direction. We do assume that the textual transition systems have abstracted away

from data manipulations of non-finite state variables (e.g., integer variables). It is important to note that the notion of 'transition' is fundamentally different for the two notations. A textual transition represents an entire family of connections between states to which the transition is applicable and the ones after the transition, expressed as a predicate or assignments relating the state before and the one after. A Petri net transition also applies to many configurations of tokens in the system (called a *marking*) that traditionally are associated with the states, namely all those for which the input places of the transition have tokens. These are transformed to a configuration where the output places have tokens, while the rest of the system configuration is unchanged. Thus both of these notions of 'transition' differ from a single edge connecting one full system state to another, in a state transition diagram or Kripke structure representation, even though those too are called 'transition systems'.

The translations must connect these concepts, and also treat the different concepts of modularity and synchronization supported by each. We also show that additional information from a translation in one direction can be used to help in traceability or to improve the quality of translations back in the other direction. This is especially useful when a system has been translated, the result has then been slightly modified, and it is then translated back to the original notation. Using the additional information can yeild a result which has a similar structure to the original. The additional information is shown to be "complete" relative to the abstraction of variables from the textual transition notation mentioned above. That is, if a model is translated to another and then immediately translated back, then based on the additional information the same model is obtained, up to renaming and changing the form of Boolean expressions.

Our translation is also *semantics-preserving* under data abstraction. That is, the original model and the translated model are bisimulation equivalent. Thus, the translation preserves all temporal properties expressible in the logic CTL*.

The translations seen in this paper are abstractions of the operation of actual compilers that are part of the VeriTech framework [GK99] for translating among specification and verification tools. The textual transition system notation is similar to the core notation of VeriTech, that is used as an intermediate notation in translating among diverse tools.

### 1.1 Related works

The problem of translating a specification given in terms of one formal model to others has been considered for many formalisms. The most general frameworks for this are the VeriTech project and the SAL project [BGL+00]. Within the STeP project [BBC+95] there is a translation from a C-like programming language notation to textual transition systems. Below we consider translations related specifically to Petri net models. Often translations to Petri nets arise from the task of transforming an input language of some task-oriented Petri net model [EZ99], [Kem00].

On the other hand, translations of Petri Nets to formalisms that provide additional analysis possibilities appear in [Wim], [GP98], [SV95].

Several papers considered the relationship between various classes of Petri nets and explicit state transition systems. In [NRT92] behaviour preserving transformations were shown between elementary transition systems and elementary net systems. In [Vog99] the existence of an ST-bisimilar Petri net is proved for an arbitrary asynchronous transition system. There are works (see, for example [WN95], [PK97]) characterizing classes of explicit state transition systems generated by different classes of Petri Nets. The idea of those translations is

to extract a set of events available in a state of the model and to simulate this set explicitly in terms of the other model. Those works focus on an extraction of classes of the models which somehow correspond. On the other hand, our task in this paper is to give a structured correspondence between wider classes of Petri nets and textual transition systems (that are closer to programs than are explicit state transiton systems).

In [GP98] a translator is presented from Petri nets to the language PROMELA of the verification tool SPIN. A place in a Petri net is translated to a variable (expressing a number of tokens in the corresponding place) in PROMELA, and a transition in a Petri net is translated to a rule for changing variable values, corresponding to the transition firing. A similar method is used in [Wim] for a translation of safe Petri nets to textual transition systems. Our translation establishes correspondences among transitions in Petri nets and transitions in textual transition systems in a similar way to theirs, although we attempt to optimize the result. Moreover, we also give a translation in the other direction.

Since the concept of modularity used in Petri nets is quite different from that accepted in other formalisms, most existing translations deal with nonmodular Petri nets. A translator that does refer to the compositional structure of the model is suggested in [SN96]. There a SA/RT specification model is translated to a class of Petri nets that does allow composing subnets through external places. Since the modularity in textual transition systems allows additional possibilities, we use a more powerful Object Oriented Petri net model, and show a correspondence between modularity of the models in both directions.

# 2 Preliminaries

## 2.1 Textual Transition Systems

The textual transition system notation (denoted TTS in the continuation) is similar to the one described in [MP92], but extended to treat various degrees of synchronization among modules. A *transition* is a state transformer with an enabling condition defining the set of states for which it is applicable, and assignments that relate the state before the transition to the state afterwards. We will refer to the former as the *enable*-part, and the latter as the *assign*-part. A *basic module* has a header with the module name followed by formal parameters within parentheses, locally declared variables, and a set of named transitions. A *composed module* has a header as above, and local variables, but contains instantiations of other modules, with actual parameters in place of formal ones, and with composition operators between them. Modules can be composed asynchronously (using ||| as in Lotos ), synchronously (using || ) and partially synchronously, by listing pairs of transitions that must synchronize, between | delimiters. Thus a typical partial synchronization would appear as $M1(a,b)|(t,s)|M2()$ where $t$ and $s$ are the names of transitions in $M1$ and $M2$, respectively (see Figure 6.2 for a richer example).

Two transitions that are synchronized are equivalent to a single 'product' transition defined by taking the conjunction of their enabling predicates, and with both sets of assignments. A module composed asynchronously from instantiations of two submodules is equivalent to a basic module with the union of the transitions in the components. One with synchronous composition is equivalent to taking the cross product of the transitions in the components, while a partial synchronization is equivalent to taking the cross product of those that synchronize and the union of the rest.

### 2.2 Object Oriented Petri Nets

In this paper we use as a Petri net model a combination of object-oriented concepts and standard place/transition Petri nets. The approach chosen for expressing the structuring primitives and composing mechanisms is from [EJN97], [Ess96], restricted to the case of safe place/transition Petri nets. Inheritance is out of our scope since there is no analogous concept in textual transition systems.

First, we define a *safe Petri net* as a structure $N = (P,\ T,\ F,\ m_0)$, where $P$ and $T$ are disjoint sets of *places* and *transitions*, respectively; $F \subseteq (P \times T) \cup (T \times P)$; and $m_0 \subseteq P$ is the *initial marking* of $N$. A *marking* of $N$ is the set of its places which contain tokens. A place $p$ is a *preplace* of a transition $t$ (written $p \in \bullet t$) if $(p,t) \in F$. It is a *postplace* of $t$ (written $p \in t\bullet$) if $(t,p) \in F$. A transition $t$ is *enabled* at a marking $m$ if $\bullet t \subseteq m$ and $t\bullet \cap (m \setminus \bullet t) = \emptyset$, i.e. all preplaces of $t$ contain a token and all postplaces of $t$ that are not preplaces do not contain a token. A marking $m'$ is obtained by *firing* of enabled transition $t$ from $m$ if $m' = (m \setminus \bullet t) \cup t\bullet$, i.e. in a firing of a transition tokens move from preplaces of the transition to its postplaces. A marking is *reachable* if it is obtained from the initial marking by firing a sequence of transitions.

A *PN-class* is characterized by a safe Petri net, a set of interfaces, a set of class instance holders referring to other classes, and a set of arcs connecting interfaces with places, transitions or other interfaces outside of the class. Directed and undirected interfaces are distinguished. For the first ones only interface-to-place and transition-to-interface arcs are allowed. We define a *basic PN-class* as a PN-class which does not refer to other classes. A place from a PN-class is an *input place* of this class if there is an arc from some interface of the PN-class to this place. We denote places, transitions and interfaces by *net elements*.

*Petri net models* are presented as communicating PN-class instances and net elements. Class instances communicate by sending and receiving tokens to and from one another. They are hierarchically structured, i.e. they may contain other class instances.

If a Petri net model is obtained from a translation of a transition system model using our TTS→PN-translation then we assume the following agreements on net element names.

- Two places constructed from one boolean variable are called by the same name superscripted by $'0'$ and $'1'$.
- Several places that correspond to one program counter are called by the same name, beginning with &, and extended with the program counter values.
- Several PN-transitions that correspond to one TTS-transition are called by the same name with different superscripts.
- A pair of input and output interfaces used for connection with the same outside place are called by the same name with prefixes $In\_$ and $Out\_$, respectively.

## 3 Translating Transition Systems to Petri Nets

### 3.1 Basic issues of the translation

Each transition system model can be translated into a Petri net model. There are two styles of translation. The first one constructs a flat Petri net in which we lose information about the modularity of the transition system model. The second style translates a transition system

model to a hierarchical Petri Net (for instance, the Petri net model described above) that keeps the original modular structure of the model. Note that for an arbitrary transition system model, a single module representation can be constructed using the semantics of transition systems. Then the translation of a transition system model to a flat Petri net is a special case of the modular translation which will be described below.

The main stages of the translation from transition systems to Petri Nets involve translating a basic TTS-module, and representing the modular structure of a transition system model. In the first stage, we associate internal variables of the TTS-module with places and external variables (including parameters) with interfaces; transitions from the TTS-module are associated with transitions in the corresponding PN-class (in a way to be described later).

In treating the modularity, an instantiation of a TTS-module is translated to an instance of the corresponding PN-class whose interfaces are connected with outside places produced for the external variables and actual parameters of the TTS-module. Asynchronous composition of two TTS-modules is translated to a PN-class consisting of the instances of the PN-subclasses corresponding to those TTS-modules. To represent partial synchronization we extend the set of interfaces of the corresponding synchronized PN-classes with new interfaces which correspond to preplaces/postplaces of the synchronized PN-transitions. Synchronous composition of two TTS-modules is translated as a partial synchronization between those TTS-modules in which all transitions are synchronized.

Explicit control constructions, allowed in other model notations, can be expressed in transition systems by program counters. To translate them to Petri Nets we produce a special place for each value of each of the program counters. Moving a token through these places gives the required control order.

Below we explain the translation outlined above in greater detail.

### 3.2 Representation of variables

To translate a variable of an enumerated type we can produce a place for each variable value and obtain a place/transition net. In this paper we consider in detail the translation of a boolean variable to a pair of places. We also give a brief description of an extension of this technique for a variable of an enumerated type by showing how to handle the program counters.

An unbounded (e.g., integer) variable is abstracted before the translation to a boolean one which is identically equal to $true$. Note that in transition system models an uninitialized variable is given some arbitrary initial value and from then on it is always defined. Therefore, an abstracted variable can be considered as a resource which is always present and whose value is not considered. We mark a fully abstracted variable with the label $'abs'$ to show that it is obtained by abstraction of a variable of a more powerful type.

Usually a boolean variable is translated to a pair of places corresponding to its values. At any state of the constructed Petri net we have exactly one token in this pair of places which represents a concrete value of the variable. However, there exists a class of boolean variables which we can represent by one place and interpret its $true$-value as the presence of a token in the place, and its $false$-value as the absence of a token. We can identify this class as the set of variables whose old and new values cannot be *false* simultaneously in any TTS-transition where the variable appears. In the continuation, such variables are marked with the label $'tkn'$. As will be shown in Section 4 when we translate a Petri net model to a transition systems representation, for each place we obtain a variable from this class if we do

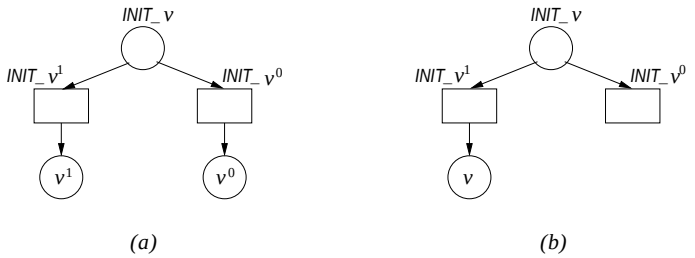



Figure 3.1: Modeling of nondeterministic choice of the initial value of $v$

not translate it in a special way due to some name agreements.

Thus we have a transition system model with boolean variables some of which are labeled with $'abs'$ or $'tkn'$. Consider a variable $v$. In the general case (there is no label for $v$), we represent $v$ as a pair of places $v^1$ and $v^0$ for $true$ and $false$ values of $v$, respectively. If $v$ is marked with $'abs'$ we translate it to the place $v^1$ which always contains a token. If $v$ is marked with $'tkn'$ we translate it to the place $v$ which contains a token iff $v = true$.

We say that an initial value of a variable is *necessary* if there is an execution of the transition system model in which the variable is used before its value is assigned by the noninitialized part of the model. For each variable which is not initialized but whose initial value is necessary, we model nondeterministic choice of its initial value in the following way. For a boolean variable $v$ without a label we add to the Petri net model an additional place $INIT\_v$ containing a token in the initial marking. A place $v^i$ obtains a token from $INIT\_v$ through the corresponding transition $INIT\_v^i$ (see Fig. 3.1a). For a boolean variable labeled with $'tkn'$ we produce a construction analogous to the previous case except that a place $v^0$ does not exist (see Fig. 3.1b). So there is an option of throwing away the initializing token. For an abstracted variable its concrete value is not considered and we can initialize it by putting a token in the corresponding place.

We refer to parameters and external variables of the TTS-module as *external identifiers*. An external identifier of the TTS-module is translated to interfaces of the corresponding PN-class just as an internal variable is translated to places. In a Petri net model, input, output, and undirected interfaces are distinguished. A variable with a label $'abs'$ is translated to an undirected interface. For a variable of other types we produce an input interface if the corresponding variable value is used in the module and an output interface if the corresponding variable value is produced in the module. Name agreements for an interface are the same as for a place, however if we need both input and output interfaces for an external identifier we add $'In\_'$ and $'Out\_'$ prefixes to their names to distinguish them.

### 3.3 Representation of TTS-transitions

Since a notion of a transition exists in both models, we will distinguish between TTS-transitions and PN-transitions.

When a variable is represented in Petri Nets by several places, any change of its value has to be expressed explicitly by firing of some PN-transition with its old value as an input and the new value as an output. So we need a separate PN-transition for each valuation of variables appearing in a TTS-transition (see Fig. 3.2).

For each TTS-transition we want a small number of PN-transitions. To decrease the number of PN-transition we transfer the *enable*-part of the TTS-transition to disjunctive

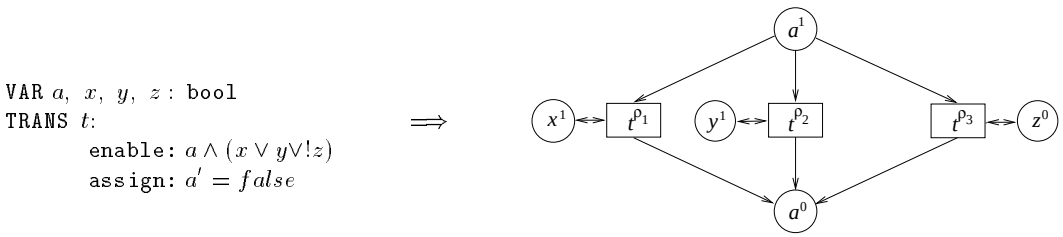



Figure 3.2: Translation of a TTS-transition $t$

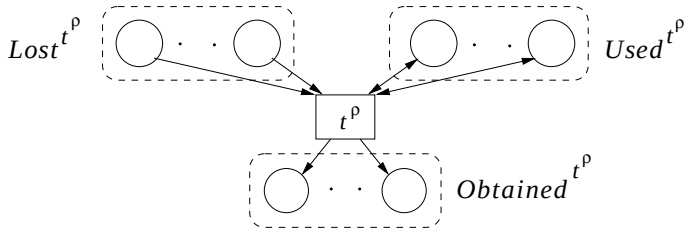



Figure 3.3: Construction of a PN-transition

normal form and translate a part of the transition corresponding to each disjunct as a separate PN-transition. For example, if we translate the TTS-transition $t$ from Figure 3.2 directly we will obtain a PN-transition for each of the seven valuations of variables $x$, $y$, $z$ and $a$ that satisfy the enabling condition. Using the disjunctive normal form $(a \wedge x) \vee (a \wedge y) \vee (a \wedge !z)$ we can consider only three PN-transitions for the valuations corresponding to each disjunct.

Fix a TTS-transition $t$. Below we consider the case in which all variables are local. The case of external variables of the translated TTS-module is obtained by using interfaces for the variables instead of places.

A set of *conditions* of a TTS-transition $t$ (written $Cond(t)$) is the set of disjuncts from the disjunctive normal form of the *enable*-part of $t$. For each condition $c$ from $Cond(t)$ and a valuation $\rho$ of variables of $c \cup assign(t)$ and their primed version satisfying $enable(t)$, we construct a PN-transition $t^\rho$ in the following way. Let us denote by $Obtained^{t^\rho}$ ($Lost^{t^\rho}$) a set of places corresponding to new (old) values of variables which are changed in $t$. A set of places corresponding to variables which are used in $t$ but do not change their values is denoted by $Used^{t^\rho}$. Let us characterize these sets for different types of variables in a formal way. Consider a variable $v$ such that $\rho(v) = val$ and $\rho(v') = val'$. If $val = val'$, we put the place corresponding to this value in $Used^{t^\rho}$. Consider the case where a value of $v$ is changed. If $v$ is not labeled, the place corresponding to $val$ belongs to $Lost^{t^\rho}$ and the place corresponding to $val'$ belongs to $Obtained^{t^\rho}$. If $v$ is labeled with $'tkn'$ then the corresponding place belongs to $Obtained^{t^\rho}$ if $val = false$, or to $Lost^{t^\rho}$ if $val = true$. Since after abstraction we obtain a variable which is always $true$, a place corresponding to an abstracted variable always belongs to $Used^{t^\rho}$.

Now we can construct $t^\rho$ as a PN-transition with a set of preplaces $Lost^{t^\rho} \cup Used^{t^\rho}$ and a set of postplaces $Obtained^{t^\rho} \cup Used^{t^\rho}$ (see Fig. 3.3).

As was shown before, the TTS-transition $t$ from Figure 3.2 is translated to three PN-transitions $t^{\rho_1}$, $t^{\rho_2}$, $t^{\rho_3}$, where $\rho_1$ : $a$=*true*, $a'$=*false*, $x$=*true*, $x'$=*true*; $\rho_2$ : $a$=*true*, $a'$=*false*,

VAR $a : int\ [a : boolean +' abs']$
$\&PC : int$ INIT 1
TRANS $t_1$:
enable: $\&PC = 1$
assign: $a' = a + 1\ [a' = true]$
$\&PC' = 2$
TRANS $t_2$:
enable: $\&PC = 2$
assign: $a' = 0\ [a' = true]$
TRANS $t_3$:
assign: $\&PC' = 3$

$\Longrightarrow$

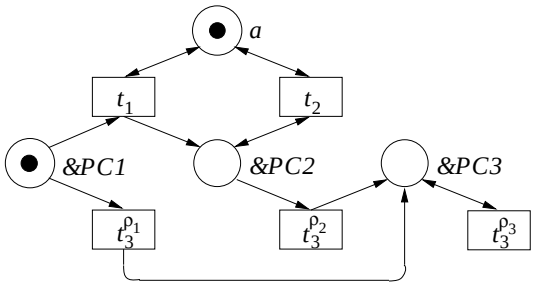



Figure 3.4: Translation of program counters ($\rho_1$: $\&PC = 1$; $\rho_2$: $\&PC = 2$; $\rho_3$: $\&PC = 3$)

$y$=*true*, $y'$=*true*; and $\rho_3$ : $a$=*true*, $a'$=*false*, $z$=*false*, $z'$=*false*. As an example we construct $t^{\rho_1}$ as follows. Since $\rho_1(a) \neq \rho_1(a')$, the place $a^1$ corresponding to $\rho_1(a)$ belongs to $Lost^{t^{\rho_1}}$ and the place $a^0$ corresponding to $\rho_1(a')$ belongs to $Obtained^{t^{\rho_1}}$. Since $\rho_1(x) = \rho_1(x')$, the place $x^1$ corresponding to $\rho_1(x)$ belongs to $Used^{t^{\rho_1}}$.

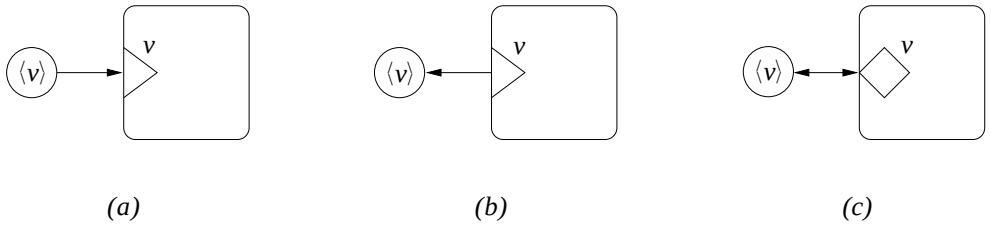
Let us consider now the translation of program counters. To express the order of execution among PN-transitions defined by a program counter we produce a special place for each possible value of the program counter. When we translate a transition with a program counter we cannot produce a token on the place for its current value without removing a token from the place for its previous value because a program counter can have no more than one value. We assume that program counters are always initialized. Obviously, a place corresponding to the initial value of the program counter contains a token in the initial state.

As was done for boolean variables, we construct for each value of a program counter in a TTS-transition a separate PN-transition. Given a TTS-transition and a program counter with known old and new values in it, to express the change of the program counter value in the constructed PN-transition we add the place corresponding to the old value of the program counter to its input, and the place corresponding to the new one to its output. Note that a TTS-transition in which the program counter appears only in the *assign*-part is equivalent to a version of that transition with a disjunction over all possible values of the program counter in the *enable*-part.

The translation of a program counter is illustrated in Fig. 3.4. We produce a place $a$ for a fully abstracted variable $a$ (a result of abstraction is shown in the brackets), and places $\&PC1$, $\&PC2$, $\&PC3$ for the corresponding values of $\&PC$. For the TTS-transition $t_3$ we have three possible values of $\&PC$ and three PN-transitions $t_3^{\rho_1}$, $t_3^{\rho_2}$ and $t_3^{\rho_3}$, respectively. In the TTS-transition $t_2$ $\&PC$ does not assigned, therefore a token is returned to the place $\&PC2$ after execution of the PN-transition $t_2$.

## 3.4 Translation of modular structure

In this section we translate an arbitrary TTS-module to a PN-class assuming that its variables are translated to places and interfaces as shown before, and for each submodule the corresponding PN-class is already produced. We need to show how an instantiation of a submodule induces arcs connecting the interfaces of the corresponding subclass with net elements outside of this subclass. To finish the translation we must also show how the different composition operations influence connections between the produced instances of subclasses.




*(a)* *(b)* *(c)*

Figure 3.5: Representation of the correspondence between interfaces and their actual values

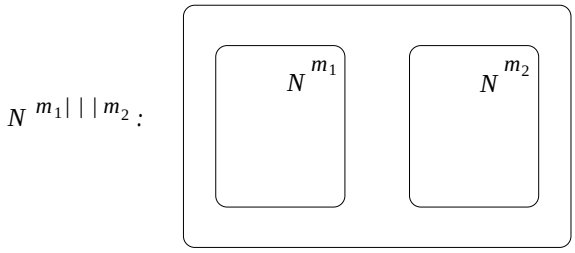



Figure 3.6: Translation of asynchronous composition

Given a TTS-module $m$ and an external identifier $v$ in it, an instantiation of $m$ uses a corresponding actual parameter of $m$, if $v$ is a parameter, or $v$ itself, otherwise, which will be denoted by $\langle v \rangle$. We first describe the construction of arcs connecting a PN-class with its outside in the case of variables labeled with *'tkn'*. As shown in Section 3.2, variables $v$ and $\langle v \rangle$ are translated to the interface $v$ and a place $\langle v \rangle$. If $v$ is an input interface we have an arc $(\langle v \rangle, v)$ (see Fig. 3.5a), and if $v$ is an output interface we have an arc $(v, \langle v \rangle)$ (see Fig. 3.5b).

In the case of variables labeled with *'abs'*, as shown in Section 3.2, variables $v$ and $\langle v \rangle$ are translated to the undirected interface $v^1$ and a place $\langle v \rangle^1$. Then we have a bidirectional arc $(v^1, \langle v \rangle^1)$ (see Fig. 3.5c).

An asynchronous composition of two TTS-modules is represented in a Petri net model as a class consisting of instances of classes corresponding to these modules (see Fig. 3.6).

Next we consider the translation of partial synchronization of two TTS-modules. Given a PN-transition $t$ from class $N^m$, a new class $N^m \setminus t$ is constructed as the class containing all PN-transitions of $N^m$ besides $t$ and additional input (output) interfaces for preplaces (postplaces) of $t$. A partial synchronization $m_1|(t_1, t_2)|m_2$ is represented as a class consisting of instances of classes $N^{m_1} \setminus t_1$ and $N^{m_2} \setminus t_2$ connected with the new PN-transition $t_1 t_2$ from $N^{m_1|(t_1,t_2)|m_2}$ in the following way: $t_1 t_2$ connects to interfaces corresponding to local variables used in the TTS-transition $t_i$ and to places corresponding to external identifiers used in the TTS-transition $t_i$. To illustrate partial synchronization consider Fig. 3.7. For the PN-class $N^{m_1}$ we construct the PN-class $N^{m_1} \setminus t_1$ which contains new interfaces $v_1$, $v_2$ and does not contain the PN-transition $t_1$. The produced synchronized transition $t_1 t_2$ is connected with its preplaces $v_1$, $u_1$ and postplaces $v_2$, $u_2$ in the subclasses through the corresponding interfaces and with the outside preplace $a^0$ and postplace $a^1$ directly.

Note that if some external identifier have different (old or new) values in the TTS-transitions $t_1$ and $t_2$, the corresponding synchronized transition does not exist. So we construct a synchronized transition $t_1 t_2$ only in cases where the original and obtained values in $t_1$ and $t_2$ coincide.

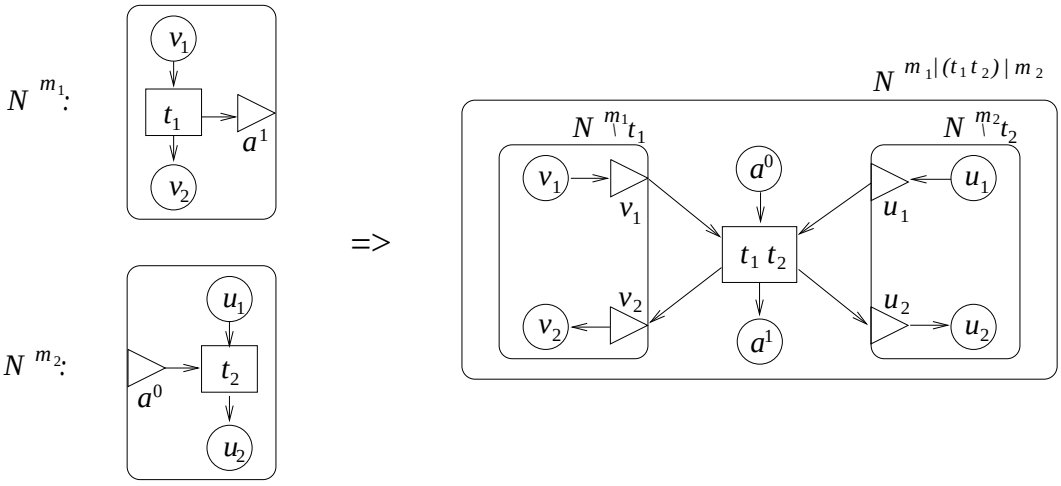



Figure 3.7: Translation of partial synchronization

A full synchronization can be considered as a partial synchronization of all combinations of transitions from the synchronized TTS-modules.

We now consider the translation of a global program counter appearing in more than one TTS-module. As was done in the flat case, we translate each possible value of the program counter to a place. For a value appearing in only one module we produce a place in the subnet corresponding to this module. A value appearing in several modules will be translated as a global variable and will be used in subnets through interfaces.

# 4 Translating Petri nets to Transition Systems

## 4.1 Representation of elements of a PN-class

In this section we show how a Petri net model can be translated to a transition system model. As for the other direction, there are two important aspects of the translation. The first is the construction of basic TTS-modules for basic PN-classes, and the second is the representation of structured PN-classes. To construct a basic TTS-module corresponding to a given basic PN-class we translate its places to variables, interfaces to either external variables or parameters and PN-transitions to TTS-transitions.

To design the translation of a basic PN-class $N^m$, we first consider the translation of its interfaces. We need a criterion to decide which interfaces will be represented by parameters of $m$ and which by external variables: if in all instantiations of $N^m$ the interface is connected with outside places that correspond to the same variable, then the interface is translated to an external variable. Otherwise, the interface is translated to a formal parameter of $m$ and the name of this parameter agrees with the name of the interface.

For places in $N^m$ we produce internal variables in $m$ except for input places. Places with the prefix & (corresponding to a program counter) will be translated in a special way, described later. For places $v^1$, $v^0$ from $N^m$ we produce a boolean variable $v$ in $m$ which is initialized with 1 if there exists an initial token in the place $v^1$ and with 0 – otherwise. In the same way we can translate a pair of places which contain exactly one token in any state of

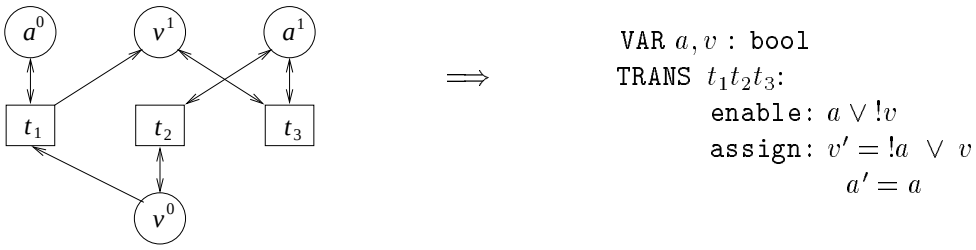



Figure 4.1: Translation of PN-transitions

the net (this pair of places forms a *place invariant*). However, here the choice of which place represents '1' and which '0' is arbitrary. The initialization of the corresponding variable is defined according to this choice.

Consider now a place $v$ in $N^m$ that has no superscript and is not a part of a place invariant. We produce a boolean variable $v$ in $m$ with the label $'tkn'$ which indicates that the value of the variable is to be interpreted as the presence of a token in the place. We initialize such a variable with *true* if the corresponding place contains a token in the initial marking, and with *false* – otherwise.

An input place is translated to an external variable if it can receive a token only through one arc. It is translated to a parameter, otherwise. In both cases it will be handled like any other place when the PN-class containing $N^m$ will be translated.

Next we consider the translation of PN-transitions. We refer to transitions that are connected with places only. Transitions connected with interfaces are translated in the same way.

A possible approach is to translate each PN-transition to a TTS-transition in which all variables are defined. Another approach is to identify a set of PN-transitions which can be represented by one TTS-transition under different valuations of its variables. Each such set will be translated to a single TTS-transition. The advantage of the latter approach is that the resulting system is more concise. Furthermore, if such a set of PN-transitions is the result of the TTS→PN-translation, then this approach results in a model which has a greater resemblance to the original transition system model.

We therefore follow the second approach. For this purpose we introduce the notion of *generalized transition* which is a set of PN-transitions with the same set of variables produced for their input and output places. A set of PN-transitions obtained from one TTS-transition in a preceding TTS→PN-translation is also considered as a generalized transition. Recall that all PN-transitions in this set have the same name with different superscripts and are therefore easy to identify.

As an example of a generalized transition, consider the PN-transitions $t_1$, $t_2$ and $t_3$ (see Fig. 4.1). Note that $\bullet t_1 = \{a^0, v^0\}$, $\bullet t_2 = \{a^1, v^0\}$, $\bullet t_3 = \{a^1, v^1\}$. These sets of places correspond to the set of variables $\{a, v\}$. Similarly, the outputs of $t_1, t_2$ and $t_3$ correspond to the set of variables $\{a, v\}$. Thus, $\{t_1, t_2, t_3\}$ is a generalized transition.

Given a generalized transition $T = \{t_1, \ \ldots \ , t_n\}$, we construct the TTS-transition $t^T$ as follows. First we construct an enabling predicate for each $t_j$ in $T$. The *enable*-part of $t^T$ is the disjunction of all these enabling predicates. Then we construct the *assign*-part of $t^T$. For each variable $v$, corresponding to a preplace or a postplace of some $t_j$ in $T$, the *assign*-part defines the value of $v$ after the execution of $t^T$.

Given a PN-transition $t_j$ from $T$, the enabling predicate for $t_j$ is the conjunction of the following conditions: For each place $p$ connected with $t_j$, that has a variable $v$ corresponding to it, if $p$ is an input place with a superscript 1 (0) the condition is $v$ ($!v$, respectively); if $p$ is an input place without superscript the condition is $v$; if $p$ is an output place without superscript which is not an input place the condition is $!v$. The reason for the latter case is that we deal with safe Petri nets in which a transition is not enabled if one of its output places contains a token.

We now show how to construct the *assign*-part. Given a PN-transition $t_j$ from $T$ and a place $v$ without superscript connected with $t_j$, the *assign*-part of $t^T$ contains the assignment $v' = true$ if $v$ is an output place and $v' = false$ if $v$ is not an output place. This definition is motivated by the fact that output places receive a token after the execution of the transition while input places lose their token.

Consider now the case in which $v$ is a variable with a superscript which corresponds to an output place of a PN-transition from $T$. Suppose that each set of input places determines a unique value of $v$ in $T$.

For each $t_j$ from $T$ we construct a formula $f^{t_j} = f_1^{t_j} \rightarrow f_2^{t_j}$, where $f_1^{t_j}$ is the conjunction of variables or their negations corresponding to the input places of $t_j$, and $f_2^{t_j}$ is the value of $v$ in $t_j$. The formula $f^T$ is defined as the conjunction of $f^{t_j}$ for each $t_j$ from $T$, and the *assign*-part of $t^T$ contains the assignment $v' = f^T$. To simplify the *assign*-part it is sometimes useful to conjunct it with the enabling condition of $t^T$. Note that this conjunction never changes the values calculated by the transition since the transition is executed only when it is enabled.

Next we handle a generalized transition $T$ which, for some input values can produce two different new values for some variable $v$. For each maximal subset $T'$ of $T$, which contains only one new value of $v$ we construct a formula $f^{T'}$ as shown above. Since in textual transition systems a nondeterministic choice is possible only for variables and constant values, if there is a set $T'$ for which $f^{T'}$ cannot be expressed by a variable or a constant value then we translate each $T'$ to a separate TTS-transition. Otherwise, the *assign*-part of $t^T$ defines the new value of $v$ by a nondeterministic choice among the formulas $f^{T'}$ for each subset $T'$ of $T$ containing only one new value of $v$.

The following example demonstrates the translation of the generalized transition $T = \{t_1, t_2, t_3\}$ (see Fig. 4.1). Since for each input of $T$ there is only one new value for $v$ and $a$, $T$ is translated to one TTS-transition. We construct the TTS-transition $t_1t_2t_3$ as follows: for $t_1$ we have the enabling predicate $!a \wedge !v$, for $t_2$ $a \wedge !v$, and for $t_3$ $a \wedge v$. Combining the three predicates by disjunction and applying some simplifications we obtain the *enable*-part of $t_1t_2t_3$: $a \vee !v$.

To construct an *assign*-part of $t_1t_2t_3$ we write the relationship between the input valuations and the output values of $v$: $((!a \wedge !v) \rightarrow true) \wedge ((a \wedge !v) \rightarrow false) \wedge ((a \wedge v) \rightarrow true) \equiv !a \vee v$ and of $a$: $((!a \wedge !v) \rightarrow false) \wedge ((a \wedge !v) \rightarrow true) \wedge ((a \wedge v) \rightarrow true) \equiv a \vee v$. Conjuncting this expression with the *enable*-part we obtain: $(a \vee v) \wedge (a \vee !v) \equiv a$. Therefore the *assign*-part of $t_1t_2t_3$ will be $v' = !a \vee v$ and $a' = a$.

We will now describe the translation of places corresponding to program counters. Note that in a Petri net any control structure is expressed by moving tokens and no program counters are used. However, a Petri net can contain places of a special form which have been introduced in the translation of program counters from a transition system model. In back-translation of a Petri net with such places, we restore the program counters in the following way. A set of places of the form $\&PCi$ corresponds to the program counter $\&PC$ in

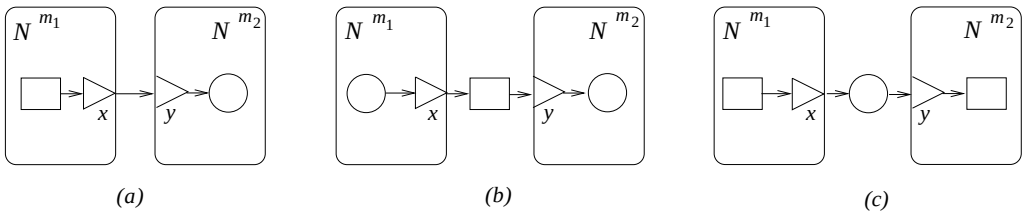



Figure 4.2: Types of connection among PN-classes

a transition system model. Fix a PN-transition $t^{PN}$ connected with one of the places $\&PCi$, and a TTS-transition $t^{TTS}$ obtained from $t^{PN}$. For any input place $\&PCi$ of $t^{PN}$ we add the conjunct $\&PC = i$ to the *enable*-part of $t^{TTS}$, and for any output place $\&PCi$ of $t^{PN}$ we add an assignment $\&PC' = i$ to the *assign*-part of $t^{TTS}$. In the case of generalized transition we obtain the *enable*- and *assign*-parts for separate transitions as shown above and compose them as in the case of usual places.

If the TTS→PN-translation handles enumerated-type variables similarly to program counters, then the back PN→TTS-translation can handle enumerated-type variables in the same way.

## 4.2 Composing structured classes

In the translation of a structured PN-class, first we find PN-transitions of a special form which are used for synchronization of different subnets. Such PN-transitions will be used for the construction of partial synchronization in the transition system model. Bellow we refer to a PN-transition in a PN-class as a *synchronized transition* if it is connected with interfaces of more than one subclass.

Since the structure of a TTS-module has to be homogeneous (i.e., it cannot mix submodules with simple transitions), in a PN-class $N^m$ which contains both subclasses and net elements we extract an additional subclass $N^{m\_add}$ and translate it to a separate submodule $m\_add$. $N^{m\_add}$ consists of all transitions appearing in $N^m$ except for the synchronized ones, and all places appearing in $N^m$ except for places connected with interfaces.

A PN-class can contain component holders that refer to other PN-classes. During class instantiation each component holder is replaced by an instantiation of the PN-class it refers to. Thus, to construct a TTS-module corresponding to a structured PN-class we need to translate instances of subclasses and relations among them.

Here we consider three types of connections among PN-classes in a Petri net model (see Fig. 4.2):

($a$) an output transition of one PN-class connects to an input place of another PN-class;

($b$) PN-classes are connected through an external synchronized transition;

($c$) PN-classes are connected through an external place.

Connections are translated together with the translation of instances of PN-classes. An instance of a PN-class is translated to an instantiation of the corresponding TTS-module, whose actual parameters are determined by the connections to the given PN-class. Below we describe the translation of the different types of connections.

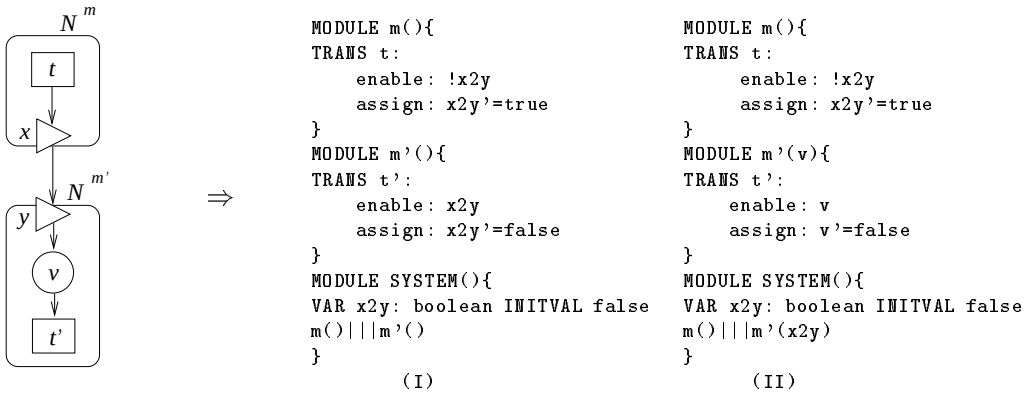



Figure 4.3: Translation of the connection of type $(a)$ using an external variable x2y (I), or a parameter v with the actual value x2y (II)

In case $(a)$ we produce an additional external variable for the arc connecting the interfaces of two submodules. We use it in the submodules instead of the variables produced for the interfaces and instead of the input places connected to these interfaces (see Fig. 4.3(I)). Another possible translation is to use the external variable as an actual parameter of the submodules connected by the given interfaces (see Fig. 4.3(II)).

In case $(b)$ we translate two subnets connected through a synchronized transition to a partial synchronization between two TTS-submodules corresponding to the given subnets. We now show how to produce a synchronized pair of TTS-transitions corresponding to a synchronized PN-transition. If we have no extra information we translate all places connected to a synchronized transition as external variables. We then construct a TTS-transition for the synchronized PN-transition using these external variables. This TTS-transition is added to each of the submodules and used in a synchronization pair. If there is extra information available, we can more meaningfully divide a synchronized transition to a pair of separate PN-transitions belonging to the synchronized subclasses. The extra information can be obtained from an analysis of the model or from a previous TTS→PN-translation, and is discussed further in the following section.

In case $(c)$, if an interface of either $N^{m_1}$ or $N^{m_2}$ is translated as a parameter, we translate the outside place connected with the interface to be an external variable. This variable is then used as the actual parameter of the TTS-module $m_1$ or $m_2$, respectively. Otherwise, in both $m_1$ and $m_2$ we use the external variable corresponding to the intermediate place instead of the variables produced for the interfaces connected with this place.

# 5 Additional information

When we apply a translation in one direction, slightly change the system, and then apply a translation in the other direction it would be desirable to obtain a system similar to the original. However some information is lost in the process of translation. This information can be saved as additional information outside of the constructed model and can later be used in a back translation. One way to keep the additional information is by adding labels to elements of the constructed model and by using a number of name agreements. It should be noted that if additional information is not available, some of it can be retrieved by an

analysis of the model.

We first describe the additional information collected during the TTS→PN-translation. Recall that the TTS→PN-translation is applied to an abstracted transition system model in which variables are restricted to be either fully abstracted or defined over finite domain. Below we assume that all finite domains are boolean, except those of program counters.

**(PLC)** When a boolean variable $v$ is translated to a pair of places, the places are names $v^1$ and $v^0$ to denote that they represent *true* and *false* values of $v$.

**(ABS)** For a fully abstracted variable, the corresponding place is labeled *'abs'*.

**(TR)** A TTS-transition is usually translated to several PN-transitions. These have names composed of the name of the TTS-transition with different superscripts.

**(PAR)** When an external identifier (a parameter or an external variable) is translated to an interface, the interface is labeled by *'par'* if the external identifier is a parameter.

**(SYN)** When a pair of synchronized transitions (each from a different TTS-module) is translated to one PN-transition, the arcs connected to the PN-transition are each labeled by the name of the TTS-module it came from.

**(PC)** A program counter $\&PC$ is translated to a set of places, one for each of its values. The place corresponding to the value $i$ will be named $\&PCi$.

We now describe the additional information collected during the PN→TTS-translation.

**(TKN)** When a place is translated to a variable, we label the variable by *'tkn'* to denote that its values can be interpreted as the presence of a token, i.e., its $false$-value is never used in an enabling condition of a TTS-transition without being changed.

**(ADD)** When a class $N^m$ consists of net elements as well as subclasses, the net elements are translated as a separate subclass. The corresponding TTS-module is labeled by *'add'* to indicate that it should not be translated back as a separate subclass.

**(VarMv)** When translating an input place of some PN-class as an external variable of the corresponding TTS-module, we label the external variable by *'input'* to indicate that it belongs to the PN-class.

As shown in Section 7, the suggested additional information is "complete" in the sense that applying two translations in a row results in the original model, up to renaming.

# 6 Example

As an example we consider an Alternating Bit Protocol. Let us translate its Petri net representation (Fig. 6.1) to a transition system model (Fig. 6.2). First we translate basic PN-classes. For the PN-class SENDER with interfaces *new, next, mes* and *ack* we produce a TTS-module `SENDER` with parameters `new, next`. For arcs *(mes, mes), (ack, ack)* joining PN-classes SENDER and RECEIVER we produce external variables `m` and `a` which are used in the translation of these PN-classes instead of the variables produced for the interface *mes* and the input place *sent_mes* (an interface *ack* and an input place *sent_ack*, respectively).

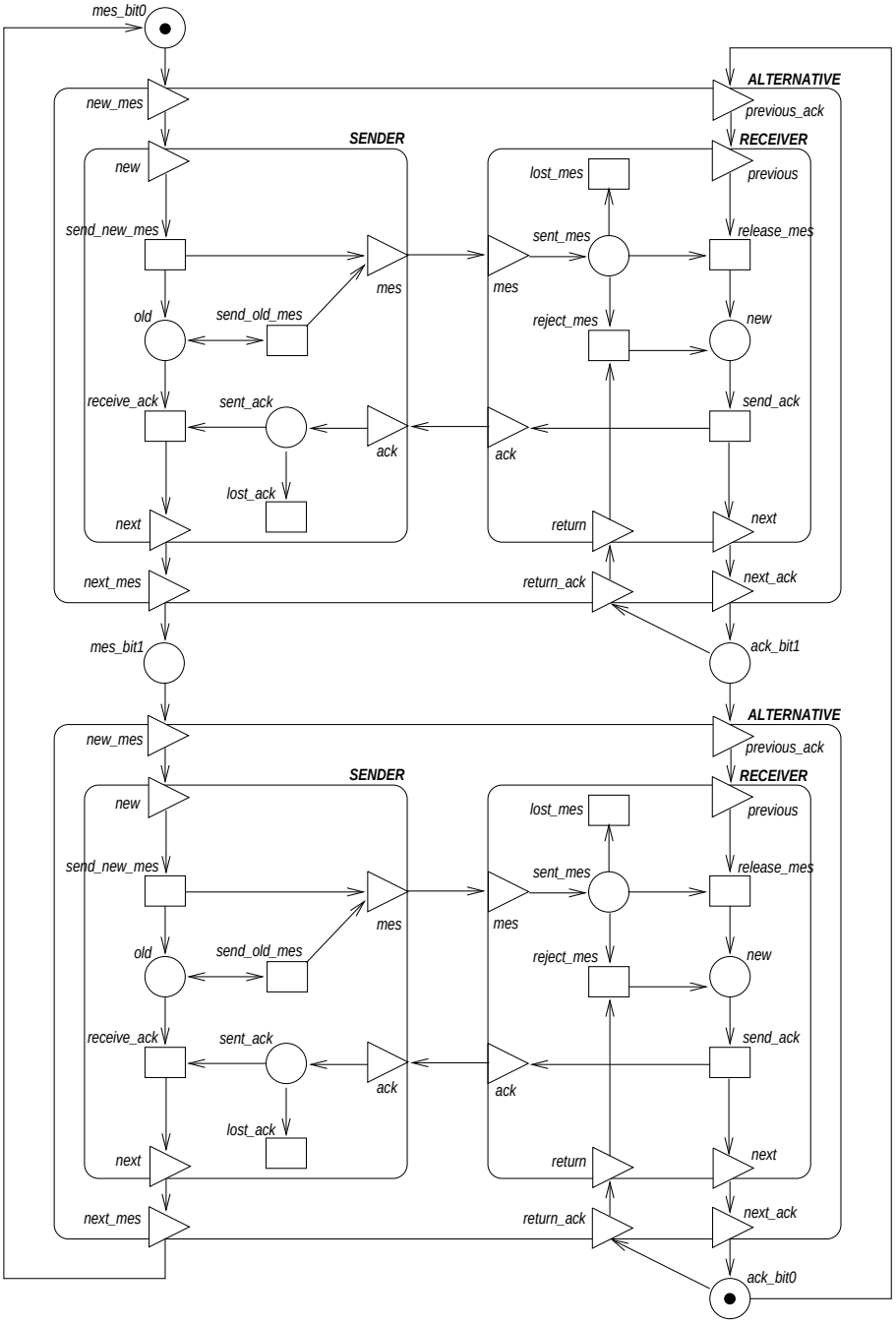



Figure 6.1: Petri Net representation of Alternating Bit Protocol

```
HOLD_PREVIOUS
VAR mes_bit0: boolean INITVAL true
    mes_bit1: boolean INITVAL false
    ack_bit0: boolean INITVAL true
    ack_bit1: boolean INITVAL false

MODULE SENDER(new, next){
VAR old: boolean INITVAL false
TRANS send_new_mes:
    enable: new /\ !old /\ !m
    assign: old'=true
            m'=true
            new'=false
TRANS send_old_mes:
    enable: old /\ !m
    assign: m'=true
            old'=true
TRANS receive_ack:
    enable: a /\ old /\ !next
    assign: next'=true
            a'=false
            old'=false
TRANS lost_ack:
    enable: a
    assign: a'=false
}
```

```
MODULE RECEIVER(previous, next, return){
       VAR new: boolean INITVAL false
       TRANS release_mes:
           enable: m /\ previous /\ !new
           assign: new'=true
                   m'=false
                   previous'=false
       TRANS reject_mes:
           enable: m /\ return /\ !new
           assign: new'=true
                   m'=false
                   return'=false
       TRANS send_ack:
           enable: new /\ !next /\ !a
           assign: a'=true
                   next'=true
                   new'=false
       TRANS lost_mes:
           enable: m
           assign: m'=false
       }
```

```
MODULE ALTERNATIVE(new_mes, next_mes, previous_ack, next_ack, return_ack){
VAR m: boolean INITVAL false
    a: boolean INITVAL false
(SENDER(new_mes, next_mes)|||RECEIVER(previous_ack, next_ack, return_ack))
}
MODULE SYSTEM(){
(ALTERNATIVE(mes_bit0, mes_bit1, ack_bit0, ack_bit1, ack_bit1)|||
 ALTERNATIVE(mes_bit1, mes_bit0, ack_bit1, ack_bit0, ack_bit0))
}
```

Figure 6.2: Transition systems representation of Alternating Bit Protocol

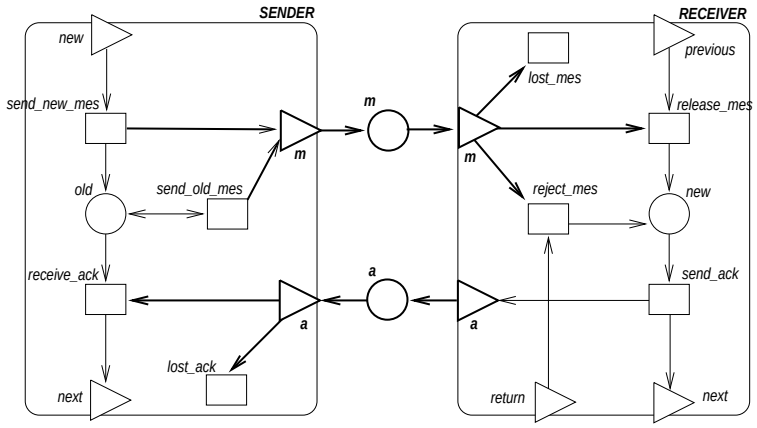



Figure 6.3: Changed part from Figure 6.1 after translation from TTS

For the internal place *old* we produce a boolean variable `old` initialized with *false*. As an example of the translation of a PN-transition let us consider the PN-transition *receive_ack*. The *enable*-part of the corresponding TTS-transition `receive_ack` is constructed as follows. For the preplaces *old, sent_ack* of *receive_ack* we obtain the condition `old` $\wedge$ `a` where the variable `a` is used instead of the variable corresponding to the input place *sent_ack* as was described above. For the postplace *next* of *receive_ack* we obtain the condition `!next`. In the *assign*-part we assign *true* to the variable `next` corresponding to the postplace of *receive_ack*, and *false* to the variables `old` and `a` corresponding to the preplaces of *receive_ack*.

The PN-class ALTERNATIVE consists of an instance of the PN-class SENDER and an instance of the PN-class RECEIVER. It is translated to the TTS-module `ALTERNATIVE` with parameters corresponding to the interfaces of ALTERNATIVE and the internal variables `m` and `a` representing arcs between those two subclasses. The instance of SENDER is translated to an instantiation of `SENDER` with the actual parameters `new_mes, next_mes`. These parameters correspond to the interfaces *new_mes* and *next_mes* of ALTERNATIVE that are connected with interfaces *new* and *next* of SENDER. The instance of RECEIVER is translated analogously, and the obtained two module instantiations are composed by asynchronous composition operation.

The Petri net model consists of two instances of the PN-class ALTERNATIVE connected through external places *ack_bit0, ack_bit1, mes_bit0*, and *mes_bit1*. These places are translated to the global variables with the same names. `ack_bit0` and `mes_bit0` are initialized with *true* because the corresponding places contain tokens. We translate the Petri net model to an asynchronous composition of two instantiations of the TTS-module `ALTERNATIVE` using the obtained global variables as actual parameters.

Next we translate the transition system representation of the Alternating Bit Protocol (Fig. 6.2) to a Petri net model, without using additional information. The changed part is shown in Figure 6.3 by bold lines. Note that all variables of the transition system model can be labeled with *'tkn'* because they originate from Petri net places. We now explain the translation of the basic TTS-module `SENDER`. Its internal variable `old` is translated to the place *old*. The parameters `new, next` and external variables `a, m` are translated to input interfaces *new, a* and output interfaces *next, m*. To exemplify the translation of a TTS-transition let us translate `send_old_mes`. This TTS-transition is translated to the unique PN-transition *send_old_mes* because we have only one condition `old` $\wedge$ `!m` in its *enable*-part, and only one possible new valuation $[old = true,\ m = false]$ for its variables. It is easy to see that $Lost^{send\_old\_mes} = \emptyset$, $Obtain^{send\_old\_mes} = \{m\}$, and $Used^{send\_old\_mes} = \{old\}$.

The TTS-module `ALTERNATIVE` is translated to the PN-class ALTERNATIVE consisting of an instance of the PN-class SENDER and an instance of the PN-class RECEIVER which are connected with places *mes* and *ack* and with the interfaces of ALTERNATIVE. The TTS-module SYSTEM is translated analogously to ALTERNATIVE.

The only difference between the original Petri net model and the one obtained after two translations is that the places *sent_mes* and *sent_ack* have been moved outside of the PN-classes *RECEIVER* and *SENDER*, respectively. In addition, some of the places and interfaces are named differently. Using the additional information (VarMv) will result in moving places *m* and *a* into the subclasses of the PN-class ALTERNATIVE. Thus, the new Petri net model will be identical to the original one (up to renaming of net elements).

# 7 Correctness

Given a Petri net model $N$ with a set of places $P^N$, the Kripke structure of $N$ is a four tuple $\mathcal{M}(N) = (M^N,\ m_0^N,\ R^N,\ L^N)$, where

- $M^N \subseteq 2^{P^N}$ is the set of reachable markings of $N$;
- $m_0^N$ is the initial marking of $N$;
- $R^N \subseteq M^N \times M^N$ is a transition relation such that $(m, m') \in R^N$ if $m'$ is obtained from $m$ by firing a transition of $N$;
- $L^N : M^N \to 2^{P^N}$ is the identity function.

Let $TS$ be a textual transition system with a set of variables $Var$ extended with labels from the set $\{'tkn',\ 'abs',\ 'PC'\}$. Let $P(Var)$ be the set of places corresponding to variables in $Var$. Such a correspondence is obtained by both TTS→PN- and PN→TTS-translations. The Kripke structure of $TS$ is a four tuple $\mathcal{M}(TS) = (S^{TS},\ S_0^{TS},\ R^{TS},\ L^{TS})$, where

- $S^{TS}$ is the set of reachable states of $TS$;
- $S_0^{TS}$ is the set of initial states of $TS$;
- $R^{TS} \subseteq S^{TS} \times S^{TS}$ is a transition relation such that $(s, s') \in R^{TS}$ if $s'$ is obtained from $s$ after the execution of some TTS-transition;
- $L^{TS} : S^{TS} \to 2^{P(Var)}$ is a function labeling a state of $TS$ with the set of places corresponding to variable values in the state. The table below defines for each value of a variable $v$ in a state $s$ the place produced for it by both translations.

| value of $v$ | label of $v$ | producing place |
|---|---|---|
| $true$ | – | $v^1$ |
| $false$ | – | $v^0$ |
| $true$ | $'tkn'$ | $v$ |
| $false$ | $'tkn'$ | – |
| $true$ | $'abs'$ | $v^1$ |
| $i$ | $'PC'$ | $vi$ |

Note that the labeling of the Kripke structure of $TS$ is not standard since it labels by places of Petri nets rather than by values of the variables. This is done in order for the Kripke structures of $TS$ and $N$ to be comparable by the bisimulation preorder.

**Theorem 1** *Given a textual transition system $TS$ in which all variables are initialized, and a Petri net model $N$ obtained from $TS$ by the TTS→PN-translation, the Kripke structures $\mathcal{M}(TS)$ and $\mathcal{M}(N)$ are bisimulation equivalent.*

**Proof Sketch.** Let us construct a relation $B \subseteq S^{TS} \times M^N$ as follows: $B = \{(s, L^{TS}(s)) \mid s \in S^{TS}\}$. Note that for every reachable state $s$ of $TS$ $L^{TS}(s)$ is a reachable marking of $N$. It can easily be shown that $B$ is a bisimulation relation. □

**Note.** A textual transition system in which not all variables are initialized has an initial state for each possible valuation of the uninitialized variables. Such a system is translated to the Petri net with PN-transitions which initialize the uninitialized variables (see Fig. 3.1).

The initializing PN-transitions and some of the markings they produce have no corresponding part in the textual transition system. Therefore, $\mathcal{M}(N)$ and $\mathcal{M}(TS)$ are not bisimilar. This can be fixed by defining an initializing stage in which only initializing transitions can execute. The initial states of $\mathcal{M}(N)$ are then defined as the states at the end of the initializing stage.

**Theorem 2** *Given a Petri net model $N$ and a textual transition system $TS$ obtained from $N$ by the PN$\rightarrow$TTS-translation, the Kripke structures $\mathcal{M}(N)$ and $\mathcal{M}(TS)$ are bisimulation equivalent.*

**Theorem 3** *If the TTS$\rightarrow$PN-translation is applied to a textual transition system $TS$, yielding a Petri net model $N$, and the PN$\rightarrow$TTS-translation is then applied to $N$, using the additional information (PAR), (PLC), (PC), (TR), and (SYN), then the resulting model is identical to $TS$ up to naming of variables and transitions and the form of boolean conditions.*

**Theorem 4** *If the PN$\rightarrow$TTS-translation is applied to a Petri net model $N$, yielding a textual transition system $TS$, and then the TTS$\rightarrow$PN-translation is applied to $TS$, using the additional information (TKN), (ADD), and (VarMv), then the resulting model is identical to $N$ up to naming of net elements.*

# References


[BBC+95] N. Bjorner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. Step: The stanford temporal prover - user's manual. Technical Report STAN-CS-TR-95-1562, Department of Computer Science, Stanford University, November 1995.

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, June 2000. Available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`.

[EJN97] R. Esser, J.W. Janneck, and M. Naedele. Using an object-oriented petri net tool for heterogeneous systems design: A case study. In *Proceedings of Algorithmen und Werkzeuge fur Petrinetze*, 1997.

[Ess96] R. Esser. *An Object Oriented Petri Net Approach to Embedded System Design.* Phd dissertation, ETH Zurich, 1996.

[EZ99] B. Eichenauer and M. Zelm. Transformation of cimosa-models into petrinet-models with pace. Technical report, 1999. http://www.ibepace.com.

[GK99] O. Grumberg and S. Katz. VeriTech: translating among specifications and verification tools–design principles. In *Proceedings of third Austria-Israel Symposium Software for Communication Technologies*, pages 104–109, April 1999. http://www.cs.technion.ac.il/Labs/veritech/.

[GP98] B. Grahlmann and C. Pohl. Profiting from spin in pep. In *Proceedings of the SPIN'98 Workshop*, 1998.

[Kem00] P. Kemper. Logistic process models go petri nets. In S. Philippi, editor, *Fachberichte Informatik, No. 7-2000: 7. Workshop Algorithmen und Werkzeuge fur Petrinetze, 2.-3*, pages 69–74. Universitatt Koblenz-Landau, Institut fur Informatik, 2000.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[NRT92] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementery transition systems. *Theoretical Computer Science*, 96:3–33, 1992.

[PK97] M. Pietkiewicz-Koutny. Transition systems of elementary net systems with inhibitor arcs. In P. Azma and G. Balbo, editors, *18th International Conference on Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 310–327. Springer-Verlag, 1997.

[Rei98] W. Reisig. *Elements of Distributed Algorithms– Modeling and Analysis with Petri Nets*. Springer-Verlag, 1998.

[SN96] L. Shi and P. Nixon. An improved translation of sa/rt specification model to high-level timed petri nets. In *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 518–537. Springer-Verlag, 1996.

[SV95] R. Sisto and A. Valenzano. Mapping petri nets with inhibitor arcs onto basic lotos behaviour expressions. *IEEE Transactions on Computers*, 44(12):1361–1370, 1995.

[Vog99] W. Vogler. Concurrent implementation of asynchronous transition systems. In *Application and Theory of Petri Nets 1999, 20th International Conference, ICATPN'99*, volume 1630 of *LNCS*, pages 284–303. Springer-Verlag, 1999.

[Wim] G. Wimmel. A bdd-based model checker for the pep tool. Technical report. Project Report 1997, http://theoretica.Informatik.Uni-Oldenburg.DE/ pep.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramskky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 1–148. 1995.