Twan Basten

Dept. of Electrical Engineering, Eindhoven University of Technology, PO Box 512, NL-5600 MB, The Netherlands
a.a.basten@tue.nl

17.5 Verifying Petri-Net Models Using Process Algebra

In this section, we describe an example of the combined application of two formal methods, namely Petri nets and process algebra [BW90], to design and verify a distributed system. First, we explain in a few words the engineering method that is the basis for combining the two methods. The key aspect is that it integrates system design with system verification. Second, we briefly explain the net model and the process algebra that are used. Third, we apply the integrated method to the development of a simple production unit. Finally, we explain what extensions of the method are possible and what still needs to be done to make the method useful in practice. An extensive treatment of the material presented in this section can be found in [Bas98, Chapter 3].

17.5.1 Method

Petri nets and process algebra are both formal methods that focus on the dynamic behaviour of systems. In distributed systems, the order in which communications between different parts of the system occur is often crucial. Does a indeed always happen before b? Can the system receive c while it is waiting for d? We are less interested in questions related to the data structures being used in the implementation of the system and the correctness of computations local to some specific part of the system. This does not mean that these issues are less important. However, other methods, such as, for example, assertional reasoning based on predicate logic, are more useful for those purposes. As a consequence, the method presented in this section is particularly useful for the design of distributed systems where the communication protocol between the various parts of the system plays an important role. It is not very well suited for data-oriented applications.

However, if Petri nets and process algebra both focus on dynamic system behaviour, what then is the use of combining them? The answer is simple: They complement each other very well. Petri nets have an easy-to-understand, graphical representation and are well suited to describe the dynamic behaviour of a system including the states in which the system can be. Hence, Petri nets are very useful for purposes of system validation and simulation. Process algebra, on the other hand, is a compositional, purely symbolic formalism, designed to compare the dynamic behaviour of different systems. It is most often used in verification. By applying term rewriting techniques and equational reasoning, it can be verified whether an implementation satisfies a given specification, where both specification and implementation are algebraic terms.

Based on the above observations, we arrive at the following engineering method consisting of four separate activities. Each activity is explained in more detail below.

1. Give a specification of the system in process-algebraic terms;
2. Construct a Petri-net model of the system;
3. Use simulation to validate the specification and to test whether the Petri net conforms to the specification;
4. Verify the correctness of the net with respect to its specification.

Algebraic specification. In the early stages of design, it is useful to specify in a concise way the order in which the actions in a system can occur. Process algebra is well suited for that purpose. Assume, for example that a system is initialised by executing either an a or a b, after which it performs a c and d in parallel. In process-algebraic terms, this is denoted as follows: $(a+b)\cdot(c \parallel d)$, where $+$ denotes choice, $\cdot$ denotes sequential composition, and $\parallel$ denotes parallel composition.

Petri-net model. Once one has a clear understanding of the dynamic behaviour of a system, it is usually not very difficult to construct a coloured Petri net whose behaviour conforms to the algebraic specification of the system. The reason for starting with an algebraic specification instead of a Petri-net model, is that the former is much more concise and easier to change than the latter. Also, in a coloured-Petri-net model, many details must be filled in which are not addressed in the specification step, such as data types, timing of events, etc.. At this stage, there is not yet a formal relation between the algebraic specification and the Petri-net model.

Simulation. The Petri-net model of a system can be simulated using tools as ExSpect [Bak97] or Design/CPN [JCHH91]. The specification and the net model can be validated and corrected, if necessary. Simulation is an excellent means to discover the more obvious mistakes in both the algebraic specification and the coloured-net model. However, by means of simulation only, it is usually impossible to guarantee that all possible errors are discovered.

Verification. In the verification phase, the Petri-net model is formally translated into an algebraic expression using the theory of [BV95a, BV95b]. By means of term rewriting techniques and equational reasoning, it is verified whether the behaviour of the net conforms to its algebraic specification. This step is useful to discover the more subtle mistakes in a specification and the corresponding Petri net.

The presentation of the four activities suggests an order. In an ideal situation, they are indeed applied in the order presented. However, needless to say that, in practice, they are seldom clearly separated. Systems engineering is a complex process where one often works on several of the four activities simultaneously. If one discovers an error during one step, it may be necessary to redo some of the work in other steps. Also, if one follows a top-down design strategy, it is possible to first design and verify a system on a high level of abstraction before adding more detail. In adding a detailed level of abstraction, it is of course recommended to repeat the design cycle.

17.5.2 Hierarchical Place/Transition Nets

To illustrate the method of the previous subsection, we use a hierarchical variant of ordinary P/T nets. Since this section focuses on the verification part of the method, which is done in process algebra, a formal definition of hierarchical P/T nets is omitted. Instead, they are explained using the example of Figure 17.1.

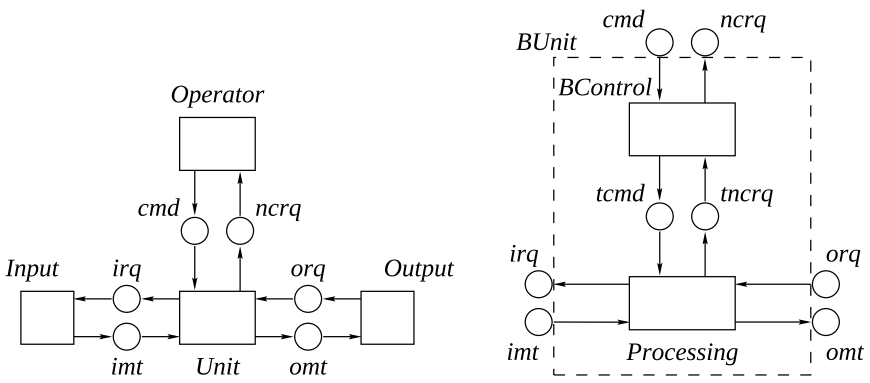

Fig. 17.1: A basic production unit and its environment.

The left part of Figure 17.1 shows a hierarchical P/T net modelling a production unit in its environment. A hierarchical P/T net may contain places, transitions, and *subnets*. The net in the example consists of four subnets connected via places. It does not have any transitions. Subnets may be instantiated with other hierarchical P/T nets. For example, subnet *Unit* is instantiated with the hierarchical net *BUnit*, which models a basic unit and is depicted in the right half of Figure 17.1. The net *BUnit* consists itself of two subnets. The dashed box divides *BUnit* in an internal and an external part, yielding a so-called *open* net. The external part or *interface* may only consist of places, which are called *pins*. The other places of the net are the *internal* places. When using an open net in a higher-level net, its pins must be mapped onto places. In the example of Figure 17.1, the mapping is simply the identity mapping. The absence of a dashed box means that a net is *closed*.

Since transitions are not always visible in a hierarchical P/T net, it is difficult to describe the behaviour of such a net in terms of transition firings. Instead, the behaviour of hierarchical P/T nets can be expressed nicely in terms of (simultaneous) consumptions and productions of tokens. For example, the *Operator* subnet in Figure 17.1 can send a command to the unit by putting a token in place *cmd*. The unit may receive the command by consuming the token, after which it can request input by putting a token in place *irq*.

Based on this observation, it is possible to give concrete form to the goal of the specification step of our method. In this step, we must specify the order in which consumptions and productions of tokens may occur. The possible behaviour given above for the interaction between a unit and its environment, for example, could act as a (partial) specification for a unit. This automatically yields the goal of the verification step. We must verify whether the observed behaviour of an implementation of a net satisfies its specification. For example, considering net *BUnit* in Figure 17.1, we might see the following behaviour. Commands received through *cmd* are transferred to *Processing* via *tcmd*. Subsequently, after receiving the command, *Processing* issues an input request by putting a token in place *irq*. If we abstract away from the command transfer between *BControl* and *Processing*, the behaviour of *BUnit* conforms to the behaviour specified for the interaction between the unit and its environment. Thus, we may conclude that *BUnit* satisfies (this part of) its specification.

Other details of the example in Figure 17.1 are not yet important. They are further explained in Section 17.5.4, where the method of the previous subsection is applied to the development of the production unit. In the next subsection, we introduce a process algebra which is designed to reason about the token game as described above.

17.5.3 A Brief Introduction to Process Algebra

This section briefly introduces a process algebra in the style of ACP [BW90] which can be used to reason about hierarchical P/T nets. The algebra is taken from [BV95b].

An ACP-like process algebra consists of a *signature* and a set of *axioms*. The signature defines the *sorts* of the algebra and its *functions* including the constants. The functions of an algebraic theory can be used to build *terms* representing *processes*. In our case, processes can be thought of as P/T nets. The axioms state which process terms are considered equal.

The algebraic theory for reasoning about P/T nets is parameterised with a universe of constants U, which are identifiers of places in a P/T net. Other sorts in the signature of the theory are the sort of atomic actions A, the sort of actions AC, and the sort of processes P; each atomic action is an action and each action is a process ($A \subseteq AC \subseteq P$).

An atomic action is either a consumption of a token from a place a in U, denoted as $a?$, or a production of a token in a, denoted $a!$. An action consists of an arbitrary number of simultaneous consumptions and/or productions. A function $\_ \mid \_ : AC \times AC \to AC$, called the synchronous merge, is used to construct actions. For example, $a? \mid a? \mid b!$ denotes the consumption of two tokens from place a and the production of one token in place b. Sort AC contains one special action τ which denotes the unobservable or silent action. The introduction of this action is useful for hiding internal behaviour of open P/T nets.

The most simple processes in our theory are the actions and a special process δ, called *deadlock* or *inaction*. Other processes can be constructed by means of the following operators, which are all functions of type $P \times P \to P$: $\_ + \_$, denoting choice, $\_ \cdot \_$, denoting sequential composition, $\_ ^* \_$, called the binary Kleene star, denoting iteration, $\_ \parallel \_$, called the merge, denoting parallel composition, and $\_ \mathbin{\|\!\!\_} \_$, called the left merge, also denoting parallel composition but with the restriction that the first action is executed by the left operand. Axioms for these operators, which may clarify their intuitive meaning, can be found in Table 17.1. The operators which have not yet been mentioned are explained below. The binding precedence of operators is as follows. Unary operators bind stronger than binary operators. Sequential composition and Kleene star bind stronger than all other binary operators. Choice binds weaker than all other operators. In Table 17.1, a is a place label in U, d is either an action in AC or δ, e, f, and g are actions in AC, and x, y, and z are processes in P.

Some of the axioms for the abovementioned operators may need an explanation. Axiom $A4$ states the right distributivity of sequential composition over choice. The fact that the left-distributivity axiom is absent implies that processes which have different moments of choice are considered to be different. That is, our theory is a so-called *branching-time* theory. Axioms $A6$ and $A7$ show that δ is indeed a natural representation of inaction. Axioms $M1$ through $M4$ define parallel composition using the left merge as an auxiliary operator. It follows from these axioms that the process algebra presented in this section is an *interleaving* theory. Axioms $ASC1$ and $ASC2$ are the so-called axioms of standard concurrency, defining some desirable properties of parallel composition. $BKS1$ through $BKS3$ axiomatise the binary Kleene star. Axiom $BKS1$ shows that process $x^* y$ means zero or more repetitions of process x followed by a single execution of y. Axiom AT says that only the visible part of an action is observed. Axioms $B1$ and $B2$ state that silent actions can be removed provided that the moments of choice are kept the same.

The operators introduced so far can be used in the specification step of a design process to specify the dynamic behaviour of the system under development. In general, such a specification takes the form of one or more terms containing only the abovementioned operators. The two operators that

$x + y = y + x$	$A1$	$e \mid f = f \mid e$	$S1$
$(x + y) + z = x + (y + z)$	$A2$	$(e \mid f) \mid g = e \mid (f \mid g)$	$S2$
$x + x = x$	$A3$		
$(x + y) \cdot z = x \cdot z + y \cdot z$	$A4$	$x \parallel y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x$	$M1$
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$A5$	$d \lfloor\!\lfloor x = d \cdot x$	$M2$
$x + \delta = x$	$A6$	$d \cdot x \lfloor\!\lfloor y = d \cdot (x \parallel y)$	$M3$
$\delta \cdot x = \delta$	$A7$	$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	$M4$
$\lambda^I_{\mathbf{s}}(\delta) = \delta$	$CSO1$	$(x \lfloor\!\lfloor y) \lfloor\!\lfloor z = x \lfloor\!\lfloor (y \parallel z)$	$ASC1$
$\mathbf{c}e \restriction I \subseteq \mathbf{s} \;\Rightarrow\; \lambda^I_{\mathbf{s}}(e) = e$	$CSO2$	$(x \parallel y) \parallel z = x \parallel (y \parallel z)$	$ASC2$
$\mathbf{c}e \restriction I \not\subseteq \mathbf{s} \;\Rightarrow\; \lambda^I_{\mathbf{s}}(e) = \delta$	$CSO3$		
$\lambda^I_{\mathbf{s}}(e \cdot x) =$		$x^* y = x \cdot (x^* y) + y$	$BKS1$
$\quad \lambda^I_{\mathbf{s}}(e) \cdot \lambda^I_{\mathbf{s}-\mathbf{c}e+\mathbf{p}e\restriction I}(x)$	$CSO4$	$x^* (y \cdot z) = (x^* y) \cdot z$	$BKS2$
$\lambda^I_{\mathbf{s}}(x + y) = \lambda^I_{\mathbf{s}}(x) + \lambda^I_{\mathbf{s}}(y)$	$CSO5$	$x^* (y \cdot ((x + y)^* z) + z) =$	
		$\quad (x + y)^* z$	$BKS3$
$e \mid \tau = e$	AT	$x \cdot \tau = x$	$B1$
		$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$	$B2$
$a \in I \;\Rightarrow\; \tau_I(a?) = \tau$	$TAC1$	$a \in I \;\Rightarrow\; \tau_I(a!) = \tau$	$TAP1$
$a \notin I \;\Rightarrow\; \tau_I(a?) = a?$	$TAC2$	$a \notin I \;\Rightarrow\; \tau_I(a!) = a!$	$TAP2$
$\tau_I(\delta) = \delta$	TAD	$\tau_I(e \mid f) = \tau_I(e) \mid \tau_I(f)$	$TA1$
$\tau_I(\tau) = \tau$	TAT	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	$TA2$
$\tau_I(x \parallel y) = \tau_I(x) \parallel \tau_I(y)$	$TAM1$	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	$TA3$
$\tau_I(x \lfloor\!\lfloor y) = \tau_I(x) \lfloor\!\lfloor \tau_I(y)$	$TAM2$	$\tau_I(x^* y) = \tau_I(x)^* \tau_I(y)$	$TA4$

Table 17.1: Basic axioms for reasoning about equality of processes.

have not been mentioned so far are used in verifying the behaviour of a P/T net against its specification. The most important one is the *causal state operator* $\lambda^{\_}_{\_}$.

The intuitive meaning of $\lambda^I_{\mathbf{s}}(x)$, where I is a set of place identifiers, $\mathbf{s}$ a bag of place identifiers, and x a process, is a P/T net x with internal places I and marking $\mathbf{s}$. The operator is axiomatised by five axioms, $CSO1$ through $CSO5$. The auxiliary functions $\mathbf{c}$ and $\mathbf{p}$, yielding for each action in AC the bag of its consumed and produced tokens, respectively, are defined as follows. For all $a \in U$ and $e \in AC$, $\mathbf{c}\tau = \emptyset$, $\mathbf{c}a? = a$, $\mathbf{c}a! = \emptyset$, $\mathbf{c}(a? \mid e) = a + \mathbf{c}e$, and $\mathbf{c}(a! \mid e) = \mathbf{c}e$; $\mathbf{p}\tau = \emptyset$, $\mathbf{p}a? = \emptyset$, $\mathbf{p}a! = a$, $\mathbf{p}(a? \mid e) = \mathbf{p}e$, and $\mathbf{p}(a! \mid e) = a + \mathbf{p}e$. The notation $\mathbf{x} \restriction D$ means that a bag $\mathbf{x}$ is restricted to some domain D. Axiom $CSO2$ can now be read as follows. It states that an action e may occur provided that all the tokens it consumes from *internal* places are available in the marking $\mathbf{s}$. Axiom $CSO3$ says that if not enough tokens are available, the action cannot be executed, resulting in a deadlock. The reason for not considering pins in the marking is that we want to determine the behaviour of a P/T net under the assumption that the environment is responsible for producing tokens into and consuming tokens from pins. Axiom $CSO4$ states that the result of executing an action e is that consumed tokens are removed from the marking, whereas tokens produced in internal places are added to it. Axioms $CSO1$ and $CSO5$ should be clear without further explanation.

The last operator in the theory is the operator τ_I. It renames consumptions from and productions into places from $I \subseteq U$ to the silent action τ. It is mainly used to hide internal behaviour of open P/T nets.

The basis of every verification are the axioms of Table 17.1. However, they are not always sufficient. To end this introduction to process algebra, one more axiom is given. The *Recursive Specification Principle* for the binary Kleene star (RSP^*) is a conditional axiom which gives for iterative processes a solution in terms of the binary Kleene star. The requirement "x guarded in $y \cdot x$" means that y cannot terminate successfully without executing at least one visible action. A formal definition of guardedness is omitted here and can be found in [Bas98, BV95a].

$x, y, z : P$	
$\dfrac{x = y \cdot x + z, \quad x \text{ guarded in } y \cdot x}{x = y \,{}^* z}$	RSP^*

The idea of verifying the behaviour of a P/T net against a specification is now straightforward. First, an algebraic term is constructed from the net, representing its observable behaviour. Second, the algebraic framework presented in this subsection is used to prove that the observable behaviour is equal to the specification. The production-unit example in the next section will clarify this approach. The formal definition of the algebraic semantics of hierarchical P/T nets based on the process algebra of this subsection can be found in [BV95a, BV95b]. In [Bas98, Chapter 3], an extensive treatment of algebraic semantics for P/T nets is given.

17.5.4 The Production Unit

In this subsection, we apply the method introduced in the previous sections to the development of a production unit. This case study is a simplified version of a problem that originated from Dutch industry.

Informal system requirements. The informal description of the production unit and its required behaviour is as follows. The most simple version of the unit can perform a single operation on a single piece of unprocessed material. First, it must receive a command. Then, it requests material, performs the operation specified in the command, and waits for an output request. Upon receiving an output request, it delivers the processed material. Typically, the processing of material takes the most time in the whole system. While the unit is processing material, it must be ready to receive the next command. In this way, the delay between two operations in a unit is minimised.

It must be possible to combine several basic units into a more complex unit by connecting them in series. For this purpose, a *generic* controller must be developed that controls *two* units. These two units are not necessarily basic ones. They can be more complex units built themselves from basic units and controllers. In this way, it must be possible to construct arbitrarily long series of basic units, using only two components, namely the basic unit and the generic controller. Therefore, it is essential that the interface behaviour of a series of basic units is similar to the behaviour of a single basic unit.

Especially the last requirement is rather vague, but, as often in practice, that is more or less the way the problem was phrased. The case study is a simplified version of the real case, because we assume that no errors can occur during the processing of the material. In reality, this is not the case. In the concluding remarks, we briefly return to this point.

Below, we discuss the design and verification of the basic unit. At the end of this subsection, we say a few words about the generic control component and the more complex units.

Algebraic specification. First, we start with the algebraic specification of a basic unit. From the informal description above, we derive the following actions that a unit can perform.

$cmd?$: receive a command	$ncrq!$: send a new-command request
$irq!$: request input material	$orq?$: receive an output request
$imt?$: receive input material	$omt!$: deliver output material

The behaviour of a basic unit can now be specified as follows:

$$BUnit = cmd? \cdot ((irq! \cdot imt? \cdot (ncrq! \cdot cmd? \parallel orq? \mid omt!))^* \delta) \qquad (17.1)$$

Note that it can be derived from the axioms in Table 17.1 that for any process x, $x^*\delta = x \cdot (x^*\delta)$. Hence, $x^*\delta$ denotes a non-terminating repetition of process x.

The above specification deserves a few words of explanation. It is not difficult to see that *BUnit* first receives a command after which it enters a non-terminating repetition. Each cycle of the repetition starts with an input request followed by the receipt of material. Upon receiving material, the unit starts its processing task, which does not result in a visible action. It continues by sending a new-command request followed by the receipt of the next command and *in parallel* it may deliver the output material, provided of course that the processing is completed. After it has delivered the output *and* it has received a new command, it starts with the next cycle of its non-terminating loop. In the above specification, the unit refuses to receive an output request if the output material is not available. Only if both the request and the material are available, the request is processed and the material delivered. A slightly different, but equally correct possibility is to let the unit first receive the output request and then deliver the material.

The next step is to construct a net model of the basic unit. Although in this section, we use P/T nets, the actual case study has been done with the tool ExSpect which is using coloured Petri nets. However, as mentioned earlier, we omit all the details about data structures and timing, and focus on the hierarchical construction and the communication structure instead.

Petri-net model. We have already seen the Petri-net model of the basic unit and its environment in Figure 17.1. The environment consists of an input system, an output system, and an operator. The environment is simply given for the sake of completeness. We do not go into detail about any of these three subnets. At the interface of the basic unit, we recognise the places we could expect after reading the list of actions a basic unit can perform. As mentioned, the implementation of the basic unit is divided into two separate subsystems, a control system, *BControl*, and a processing system, *Processing*. The control system transfers commands and new-command requests between the environment and the processing system.

The next two steps are simulation and verification. However, for that purpose, we need to have at least a specification of the systems *BControl* and *Processing*. Given such specifications, it is in principle possible to simulate the processing unit without detailing the net models of *BControl* and *Processing*. However, current tools as ExSpect and Design/CPN do not allow simulation of such incomplete net models. Therefore, we proceed as follows. First, we give an algebraic specification of the systems *BControl* and *Processing*. Then, we verify that the unit behaves correctly, provided the subnets for *BControl* and *Processing* satisfy their specification. Third, net models for *BControl* and *Processing* are given. Finally, the complete model of the basic unit is simulated and verified.

Algebraic specification. The specification for *BUnit* shows that the receipt of a command and the sending of a new-command request alternate. Given the informal requirement that *BControl* transfers commands and new-command requests between the environment and the processing system, the following specification for *BControl* makes sense:

$$\mathit{BControl} = (\mathit{cmd}? \mid \mathit{tcmd}! \cdot \mathit{tncrq}? \mid \mathit{ncrq}!)^* \, \delta \tag{17.2}$$

For the processing part, we arrive at the following specification. Actions are processed simultaneously as much as possible. Upon receiving a command, an input request is sent; upon receiving material, a new command is requested and upon receiving an output request, the output material is delivered.

$$\mathit{Processing} = (\mathit{tcmd}? \mid \mathit{irq}! \cdot \mathit{imt}? \mid \mathit{tncrq}! \cdot \mathit{orq}? \mid \mathit{omt}!)^* \, \delta \tag{17.3}$$

Note that in terms which are equal up to associativity brackets are often omitted. It may be surprising that neither *BControl* nor *Processing* contain explicit parallelism. However, if we have a close look at the specifications, we see that *Processing* produces a token in place *tncrq* at the same time it receives input material. Hence, the transfer of the new-command request to the environment by system *BControl* can occur in parallel with the delivery of output material by system *Processing*. This is exactly the desired result. The informal argument of this paragraph is confirmed by the verification in the next step of the design cycle.

Verification. Given the specifications for $BControl$ and $Processing$, we can now verify the behaviour of the basic unit. The formal algebraic semantics for the behaviour of the Petri net $BUnit$ is the following algebraic term (see [BV95a, BV95b]):

$$\tau_I \circ \lambda^I_\emptyset(BControl \parallel Processing) \ ,$$

where I is the set of internal places $\{tcmd, tncrq\}$. The above term can be interpreted as follows. The *unrestricted* behaviour of system $BUnit$ is simply the parallel composition of the behaviour of all its components. The causal state operator $\lambda^I_\emptyset$ restricts this behaviour to all possible firing sequences allowed by the initial marking which in the above case is the empty bag of tokens. The abstraction operator τ_I hides consumptions from and productions into internal places. By means of equational reasoning, it can be shown that the behaviour of the basic unit satisfies its specification. That is,

$$\tau_I \circ \lambda^I_\emptyset(BControl \parallel Processing) = BUnit \ .$$

The details of the verification are as follows. First, we introduce some abbreviations.

$tc \ = \ cmd? \mid tcmd!$	transfer command
$tn \ = \ tncrq? \mid ncrq!$	transfer new-command request
$pc \ = \ tcmd? \mid irq!$	process command
$ri \ = \ imt? \mid tncrq!$	receive input
$po \ = \ orq? \mid omt!$	produce output

The first part of the verification considers the complete behaviour of the basic unit without hiding internal details. Only after we have derived a simple expression for $\lambda^I_\emptyset(BControl \parallel Processing)$, its internal behaviour is hidden.

$$\begin{array}{ll}
& \lambda^I_\emptyset(BControl \parallel Processing) \\
= & \{ \text{ Specifications } BControl \text{ and } Processing;\ BKS1, A6 \text{ (Both } 2\times) \ \} \\
& \lambda^I_\emptyset(tc \cdot tn \cdot BControl \parallel pc \cdot ri \cdot po \cdot Processing) \\
= & \{ \ M1 \ \} \\
& \lambda^I_\emptyset(tc \cdot tn \cdot BControl \mathbin{⌊\!⌊} pc \cdot ri \cdot po \cdot Processing \\
& \qquad + pc \cdot ri \cdot po \cdot Processing \mathbin{⌊\!⌊} tc \cdot tn \cdot BControl) \\
= & \{ \ M3, A6, BKS1 \text{ (All } 2\times) \ \} \\
& \lambda^I_\emptyset(tc \cdot (tn \cdot BControl \parallel Processing) \\
& \qquad + pc \cdot (ri \cdot po \cdot Processing \parallel BControl)) \\
= & \{ \ CSO5, CSO4 \ (2\times), CSO2, CSO3 \ \} \\
& tc \cdot \lambda^I_{tcmd}(tn \cdot BControl \parallel Processing) \\
& \qquad + \ \delta \cdot \lambda^I_\emptyset(ri \cdot po \cdot Processing \parallel BControl) \\
= & \{ \ A7, A6 \ \} \\
& tc \cdot \lambda^I_{tcmd}(tn \cdot BControl \parallel Processing)
\end{array}$$

Summarising, the above derivation yields

$$\lambda^I_\emptyset(BControl \parallel Processing) = tc \cdot \lambda^I_{tcmd}(tn \cdot BControl \parallel Processing) \ .$$

In a similar way, it is possible to derive the following results.

$$\begin{array}{l}
\lambda^I_{tcmd}(tn \cdot BControl \parallel Processing) = \\
\qquad pc \cdot \lambda^I_\emptyset(tn \cdot BControl \parallel ri \cdot po \cdot Processing) \\
\lambda^I_\emptyset(tn \cdot BControl \parallel ri \cdot po \cdot Processing) = \\
\qquad ri \cdot \lambda^I_{tncrq}(tn \cdot BControl \parallel po \cdot Processing) \\
\lambda^I_{tncrq}(tn \cdot BControl \parallel po \cdot Processing) = \\
\qquad tn \cdot \lambda^I_\emptyset(BControl \parallel po \cdot Processing) \\
\qquad + po \cdot \lambda^I_{tncrq}(tn \cdot BControl \parallel Processing) \\
\lambda^I_\emptyset(BControl \parallel po \cdot Processing) = \\
\qquad tc \cdot \lambda^I_{tcmd}(tn \cdot BControl \parallel po \cdot Processing) \\
\qquad + po \cdot \lambda^I_\emptyset(BControl \parallel Processing) \\
\lambda^I_{tcmd}(tn \cdot BControl \parallel po \cdot Processing) = \\
\qquad po \cdot \lambda^I_{tcmd}(tn \cdot BControl \parallel Processing) \\
\lambda^I_{tncrq}(tn \cdot BControl \parallel Processing) = tn \cdot \lambda^I_\emptyset(BControl \parallel Processing)
\end{array}$$

The above equations can be combined to derive a result for the expression $\lambda^I_{tcmd}(tn \cdot BControl \parallel Processing)$. The reason to look at this expression and not any of the others, is that the specification of *BUnit* tells us that we may expect $\lambda^I_{tcmd}(tn \cdot BControl \parallel Processing)$ to be the beginning of an iteration.

$$\begin{array}{ll}
 & \lambda^I_{tcmd}(tn \cdot BControl \parallel Processing) \\
= & \{ \text{ Repeated substitution } \} \\
 & pc \cdot ri \cdot (\ tn \cdot (\ tc \cdot po \cdot \lambda^I_{tcmd}(tn \cdot BControl \parallel Processing) \\
 & \qquad\qquad + po \cdot tc \cdot \lambda^I_{tcmd}(tn \cdot BControl \parallel Processing) \\
 & \qquad\qquad) \\
 & \qquad + po \cdot tn \cdot tc \cdot \lambda^I_{tcmd}(tn \cdot BControl \parallel Processing) \\
 & \qquad) \\
= & \{ \ A4\ (2\times) \ \} \\
 & pc \cdot ri \cdot (tn \cdot (tc \cdot po + po \cdot tc) + po \cdot tn \cdot tc) \\
 & \qquad \cdot \lambda^I_{tcmd}(tn \cdot BControl \parallel Processing) \\
= & \{ \text{ Derivation below } \} \\
 & pc \cdot ri \cdot (tn \cdot tc \parallel po) \cdot \lambda^I_{tcmd}(tn \cdot BControl \parallel Processing)
\end{array}$$

$$\begin{array}{ll}
 & tn \cdot (tc \cdot po + po \cdot tc) + po \cdot tn \cdot tc \\
= & \{ \ M2\ (3\times) \ \} \\
 & tn \cdot (tc \mathbin{⌊\!⌊} po + po \mathbin{⌊\!⌊} tc) + po \mathbin{⌊\!⌊} tn \cdot tc \\
= & \{ \ M1 \ \} \\
 & tn \cdot (tc \parallel po) + po \mathbin{⌊\!⌊} tn \cdot tc \\
= & \{ \ M3 \ \} \\
 & tn \cdot tc \mathbin{⌊\!⌊} po + po \mathbin{⌊\!⌊} tn \cdot tc \\
= & \{ \ M1 \ \}
\end{array}$$

$tn \cdot tc \parallel po$

Axioms $A6$ and RSP^* yield

$$\lambda^I_{tcmd}(tn \cdot BControl \parallel Processing) = (pc \cdot ri \cdot (tn \cdot tc \parallel po))^* \delta \ .$$

Note that the guardedness requirement for RSP^* is fulfilled. Combining the above result with the result derived for $\lambda^I_\emptyset(BControl \parallel Processing)$ yields

$$\lambda^I_\emptyset(BControl \parallel Processing) = tc \cdot ((pc \cdot ri \cdot (tn \cdot tc \parallel po))^* \delta) \ ,$$

which already looks very similar to the specification of the basic unit. In the second part of the verification, the internal behaviour in the above result is hidden. Recall that I is the set of internal places $\{tcmd, tncrq\}$.

$$
\begin{array}{ll}
 & \tau_I \circ \lambda^I_\emptyset(BControl \parallel Processing) \\
= & \{ \text{ Previous result; substitution of abbreviations } \} \\
 & \tau_I(cmd? \mid tcmd! \cdot ((tcmd? \mid irq! \cdot imt? \mid tncrq! \cdot \\
 & \qquad (tncrq? \mid ncrq! \cdot cmd? \mid tcmd! \parallel orq? \mid omt!) \\
 & \quad)^* \delta)) \\
= & \{ \text{ Axioms for the abstraction operator } \tau_I \ \} \\
 & cmd? \mid \tau \cdot ((\tau \mid irq! \cdot imt? \mid \tau \cdot (\tau \mid ncrq! \cdot cmd? \mid \tau \parallel orq? \mid omt!))^* \delta) \\
= & \{ \ AT \ \} \\
 & cmd? \cdot ((irq! \cdot imt? \cdot (ncrq! \cdot cmd? \parallel orq? \mid omt!))^* \delta)
\end{array}
$$

Hence,

$$
\begin{array}{rl}
 & \tau_I \circ \lambda^I_\emptyset(BControl \parallel Processing) \\
= & cmd? \cdot ((irq! \cdot imt? \cdot (ncrq! \cdot cmd? \parallel orq? \mid omt!))^* \delta) \\
= & BUnit \ ,
\end{array}
$$

which is the desired result. We have formally *proven* that the behaviour of the P/T net *BUnit* is indeed as specified by its algebraic specification of Equation 17.1, provided of course that the subnets *BControl* and *Processing* are implemented according to their specifications of Equations 17.2 and 17.3. So, the next step is to give the net models for those two systems.

Petri-net model. Figure 17.2 shows the Petri-net models of *BControl* and *Processing*. System *BControl* keeps track of the state of the unit. A unit is either *ready* to receive a command or *busy* processing. System *Processing* can receive a command if it is *empty*. Upon receiving a command, it sends an input request and enters a state called *waiting for material*. Next, it receives *raw material*. Processing turns raw material into *processed material*. Finally, the unit delivers the processed material, returning to the empty state.

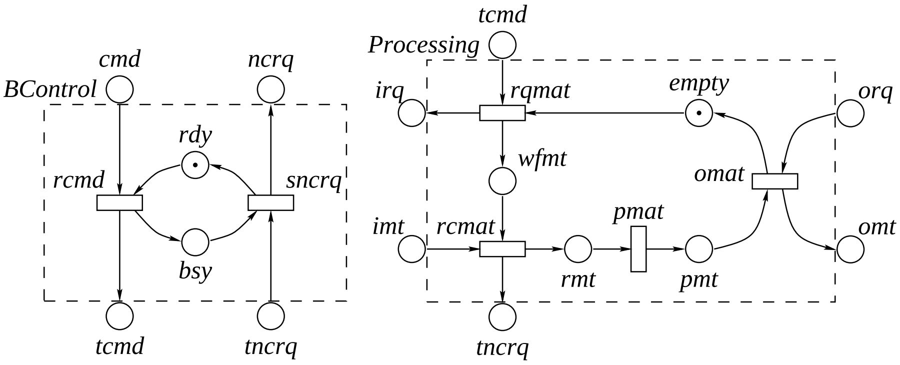

Fig. 17.2: The subsystems of the basic unit.

Simulation and verification. At this point, the P/T-net model for the basic unit is complete, which means that it can be simulated using tools as ExSpect and Design/CPN. Since the net of the production unit is very simple, the simulation results are not very surprising. Therefore, we do not go into details.

The final step in the design cycle of the production unit is the verification of the systems *BControl* and *Processing*. Details are left to the reader. We simply state the results. Transition names are used as abbreviations for the synchronous merge of their consumptions and productions. The algebraic semantics for *BControl* is as follows.

$$\tau_I \circ \lambda^I_{rdy}(rcmd \parallel sncrq) \ ,$$

where I is the set of internal places $\{rdy, bsy\}$. The semantics for *Processing* is the following term.

$$\tau_J \circ \lambda^J_{empty}(rqmat \parallel rcmat \parallel pmat \parallel omat) \ ,$$

where J is the set of places $\{empty, wfmt, rmt, pmt\}$. It is straightforward to verify that these two terms are equal to the specifications of *BControl* and *Processing* given in Equations 17.2 and 17.3. The verifications follow the line of the verification of *BUnit*. However, the verification of *Processing* is very tedious when only the basic axioms of Table 17.1 plus RSP^* can be used. The so-called *expansion theorem*, which is a generalisation of $M1$ to an arbitrary number of processes in parallel, greatly simplifies the calculations. The theorem can be found in [BV95a, BV95b].

As a consequence of the previous verification steps, we may conclude that the complete hierarchical net model of the basic production unit, given in Figures 17.1 and 17.2, satisfies its specification.

Properties of a basic unit. At this point, it is time to return to the requirement that the behaviour of a series of basic units must be similar to the behaviour of a single basic unit. We ask ourselves: What are the fundamental properties in the dynamic behaviour of a basic unit? The most obvious one is related to the interface between the unit and the operator. As we have seen already, the receipt of a command and the sending of a new-command request alternate. We can prove this property formally. Hiding all actions in specification $BUnit$ of Equation 17.1 except $cmd?$ and $ncrq!$ yields the following behaviour:

$$(cmd? \cdot ncrq!)^* \delta \ , \tag{17.4}$$

which shows that the basic unit of Figures 17.1 and 17.2 indeed alternates commands and new-command requests. The calculations are as follows. Let I be the set $\{irq, imt, orq, omt\}$.

$$\begin{array}{ll}
 & \tau_I(BUnit) \\
= & \{ \text{ Equation 17.1 } \} \\
 & \tau_I(cmd? \cdot ((irq! \cdot imt? \cdot (ncrq! \cdot cmd? \parallel orq? \mid omt!))^* \delta)) \\
= & \{ \text{ Axioms of the abstraction operator } \tau_I \ \} \\
 & cmd? \cdot ((\tau \cdot \tau \cdot (ncrq! \cdot cmd? \parallel \tau \mid \tau))^* \delta) \\
= & \{ B1, AT \} \\
 & cmd? \cdot ((\tau \cdot (ncrq! \cdot cmd? \parallel \tau))^* \delta) \\
= & \{ M1, M3, M2 \} \\
 & cmd? \cdot ((\tau \cdot (ncrq! \cdot (cmd? \parallel \tau) + \tau \cdot ncrq! \cdot cmd?))^* \delta) \\
= & \{ M1, M2\ (2\times), B1 \} \\
 & cmd? \cdot ((\tau \cdot (ncrq! \cdot (cmd? + \tau \cdot cmd?) + \tau \cdot ncrq! \cdot cmd?))^* \delta) \\
= & \{ A6, B2, A6 \} \\
 & cmd? \cdot ((\tau \cdot (ncrq! \cdot cmd? + \tau \cdot ncrq! \cdot cmd?))^* \delta) \\
= & \{ A6, B2, A6 \} \\
 & cmd? \cdot ((\tau \cdot ncrq! \cdot cmd?)^* \delta)
\end{array}$$

This result is almost the result we are looking for. The final step using RSP^* is as follows.

$$\begin{array}{ll}
 & cmd? \cdot ((\tau \cdot ncrq! \cdot cmd?)^* \delta) \\
= & \{ BKS1, A6 \} \\
 & cmd? \cdot ((\tau \cdot ncrq! \cdot cmd?) \cdot ((\tau \cdot ncrq! \cdot cmd?)^* \delta)) \\
= & \{ A5\ (3\times), B1 \} \\
 & (cmd? \cdot ncrq!) \cdot (cmd? \cdot ((\tau \cdot ncrq! \cdot cmd?)^* \delta))
\end{array}$$

Hence, $A6$ and RSP^* yield

$$cmd? \cdot ((\tau \cdot ncrq! \cdot cmd?)^* \delta) = (cmd? \cdot ncrq!)^* \delta \ ,$$

which is the desired result.

Another characterising property is that the unit behaves as a one-place buffer if we look at the stream of material. Hiding the appropriate actions yields:

$$\tau \cdot ((imt? \cdot omt!)^* \delta) \ , \tag{17.5}$$

which states that the unit after initialisation, expressed by the silent action τ, enters a cycle in which it alternates the input of material and the output of material. The details of the proof are left to the reader.

A final property is that the unit can accept two commands before it must produce output. That is, it behaves as a two-place buffer with respect to commands. Hiding the appropriate actions yields the following behaviour:

$$cmd? \cdot ((cmd? \parallel omt!)^* \, \delta) \ , \tag{17.6}$$

which is indeed the behaviour of a two-place buffer. This last property confirms that the unit satisfies the requirement that it must be able to receive a new command, while it is still busy processing.

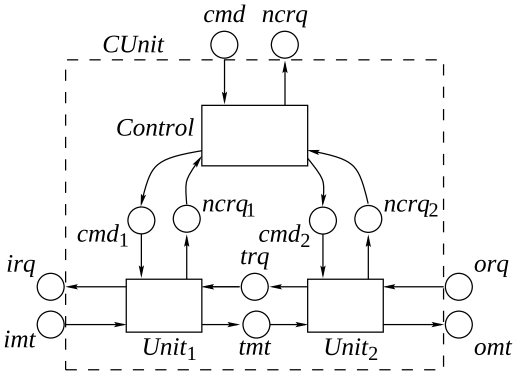

Fig. 17.3: Two units in series.

Complex Units. So, what happens if we combine basic units into more complex units? As mentioned, it must be possible to construct arbitrarily long series of basic units, using only two components, namely the basic unit and a generic controller for two units. We do not discuss the entire design trajectory. Figure 17.3 simply shows the high-level net of a complex unit consisting of two units in series. The basic idea is that the control component receives a command from the environment and forwards this to the two units. Commands may only be forwarded if the receiving unit is ready to accept it. In its most simple form, the complex unit consists of two basic units. However, by instantiating $Unit_1$ and $Unit_2$ with complex units themselves, it is possible to construct arbitrarily long series of basic units.

The question is what properties a complex unit must satisfy. Obviously, in order to be compatible with a single basic unit, it must alternate commands with new-command requests. That is, if we calculate the behaviour of system *CUnit* and hide the appropriate actions, it must satisfy requirement 17.4. Furthermore, looking at the stream of material, it is not difficult to see that a complex unit consisting of N basic units should behave as an N-place buffer. Each basic unit can contain one piece of material. Finally, in order to exploit the parallelism in the basic units, a unit consisting of N basic units should be able to receive $N+1$ commands before it must give its first output. These last two properties are generalisations of properties 17.5 and 17.6 derived in the previous paragraph for the basic unit.

We conclude this paragraph with the results we have obtained for complex units thus far. We have calculated the complete behaviour of a unit consisting of two basic units in series. We also have verified the above three properties for such a unit. The verifications are not very hard for someone experienced with the theory. However, they are very tedious and take quite some pages. Verifying the properties for a series of three basic units is not recommended without the support of tools. For more details on the production-unit case study, the reader is referred to [Bas98, Section 3.8].

17.5.5 Concluding Remarks

In this section, we have discussed an engineering method combining Petri nets and process algebra. The example of the production unit shows the possibilities of the approach.

To date, the algebraic verifications do not incorporate data; they focus on the order of consumptions and productions of tokens. An important extension of the method, is the inclusion of data in the style of μCRL [GP93]. Such an extension makes it possible to handle preconditions, or guards, in coloured nets, which is important since preconditions are used to steer the flow of tokens in a net. Furthermore, the induction principles of μCRL make it possible to derive properties about coloured nets with parameters. For example, assume that the production unit of Figure 17.3 is instantiated with one unit of N basic units in series and another unit of M basic units in series. Using the induction principles of μCRL, it is possible to prove the buffer properties of the unit for arbitrary N and M instead of specific instances of N and M. Although such a parameterised proof is more difficult than a proof for specific instances of the parameters, in the long run, it can save a lot of work.

Another useful extension of the method is the following. As mentioned, the case study discussed in this section is simplified in the sense that units always operate correctly. However, in reality, processing errors may occur. If such an error occurs, a unit must be reset, after which processing may resume. This means that error-handling facilities must be added to the specifications and the net models. It would be nice if some of the verifications discussed in this section could be reused in verifying the correctness of the extended production units. The theory presented in [AB97, BA99] can be used for this purpose. However, it needs to be adapted to the framework presented here.

Finally, the method discussed in this section is only useful in practice with proper tool support. It is simply impossible to do any real-world verifications completely by hand. First, it is necessary to develop tools supporting algebraic reasoning in general. Current tools are usually built for a specific process algebra and, hence, not useful for the process algebra used in our method. In a later phase, process-algebra tools and Petri-net tools must be combined to form one integrated design environment. An initial study on the possibility to provide support for ACP-style algebraic reasoning in the general proof environment PVS [SRI00] can be found in [BH99].

Bibliography

[AB97] W.M.P. van der Aalst and T. Basten. Life-Cycle Inheritance: A Petri-Net-Based Approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997, 18th. International Conference, ICATPN'97, Proceedings*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81, Toulouse, France, June 1997. Springer, Berlin, Germany, 1997.

[BA99] T. Basten and W.M.P. van der Aalst. Inheritance of Behavior. Computing Science Report 99/17, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, November 1999.

[Bak97] Bakkenist Management Consultants. *ExSpect 6 User Manual*, 1997. Information available at `http://www.exspect.com/`.

[Bas98] T. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, December 1998.

[BH99] T. Basten and J. Hooman. Process Algebra in PVS. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems, 5th. International Conference, TACAS'99, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 270–284, Amsterdam, The Netherlands, March 1999. Springer, Berlin, Germany, 1999.

[BV95a] T. Basten and M. Voorhoeve. An Algebraic Semantics for Hierarchical P/T Nets. Computing Science Report 95/35, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, December 1995. An extended abstract appeared as [BV95b].

[BV95b] T. Basten and M. Voorhoeve. An Algebraic Semantics for Hierarchical P/T Nets (extended abstract). In G. De Michelis and M. Diaz, editors, *Application and Theory of Petri Nets 1995, 16th. International Conference, Proceedings*, volume 935 of *Lecture Notes in Computer Science*, pages 45–65, Torino, Italy, June 1995. Springer, Berlin, Germany, 1995.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1990.

[GP93] J.F. Groote and A. Ponse. Proof Theory for μCRL: A Language for Processes with Data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Semantics of Specification Languages, Proceedings of the International Workshop SoSL*, Workshops in Computing, pages 232–251, Utrecht, The Netherlands, October 1993. Springer, Berlin, Germany, 1994.

[JCHH91] K. Jensen, S. Christensen, P. Huber, and M. Holla. *Design/CPN: A Reference Manual.* Meta Software Corporation, 1991. Information available at `http://www.daimi.au.dk/designCPN`.

[SRI00] SRI International. *PVS home page*, 2000. `http://www.csl.sri.com/sri-csl-pvs.html`.