1

INTRODUCTION

Many computing systems consist of a possibly large number of components that not only work independently or concurrently, but also interact or communicate with each other from time to time. Examples of such systems are operating systems, distributed systems and communication protocols, as well as systolic algorithms, computer architectures and integrated circuits.

Conceptually, it is convenient to treat these systems and their components uniformly as *concurrent processes*. A process is here an object that is designed for a possibly continuous interaction with its user, which can be another process. An interaction can be an input or output of a value, but we just think of it abstractly as a *communication*. In between two subsequent communications the process usually engages in some *internal actions*. These proceed autonomously at a certain speed and are not visible to the user. However, as a result of such internal actions the process behaviour may appear *nondeterministic* to the user. Concurrency arises because there can be more than one user and inside the process more than one active subprocess. The behaviour of a process is unsatisfactory for its user(s) if it does not communicate as desired. The reason can be that the process stops too early or that it engages in an infinite loop of internal actions. The first problem causes a *deadlock* with the user(s); the second one is known as *divergence*. Thus most processes are designed to communicate arbitrarily long without any danger of deadlock or divergence.

Since the behaviour of concurrent processes has so many facets, it is not surprising that their description has been approached from rather different angles. In particular, Petri nets [Pet62, Rei85], algebraic process terms [Mil80, BHR84, BK86, BW90b] and logical formulas of temporal or ordinary predicate logic [Pnu77, CH81, MC81] have been used. One may regret such a diversity, but we claim that these different

descriptions can be seen as expressing complementary views of concurrent processes, each one serving its own purpose.

To support this claim we present a theory where nets, terms and formulas represent processes at three levels of abstraction: Petri nets are used to describe processes as concurrent and interacting machines with all details of their operational machine behaviour; process terms are used as an abstract concurrent programming language that stresses compositionality, i.e. how complex processes are composed from simpler ones by a small set of process operators; and logical formulas are used to describe or *specify* the communication behaviour of processes as required by their users. The main emphasis and technical contribution of this theory are transformations for a top-down design of concurrent processes starting with formulas and ending in nets:

formulas
↓
terms
↓
nets

The top arrow refers to transformation rules for a systematic construction and verification of process terms from logical formulas and the bottom arrow refers to the transition rules of an operational net semantics of process terms. Apart from defining the transformations the theory will carefully develop its concepts. To show that they fit together well, we explore their relationship by establishing various properties.

It seems that our theory is the first approach which brings together nets, terms and formulas in one uniform framework, viz. that of process construction. We hope that it will contribute to a better understanding of these different description methods. On the more ambitious side, this theory is intended as one detailed case study for an essential topic in computer science: provably correct system construction through various levels of abstraction.

To achieve a coherent theory, we have concentrated on a simple setting, but one where a number of interesting process constructions are possible. The main decision was to leave communications as atomic or unstructured objects. This has freed us from several notational and conceptual problems (such as dealing with infinitely many values or assignable variables) and enabled us to concentrate more deeply on the rest. The remaining decisions concern various details such as the class and representation of nets, the operators allowed in process terms, the class of logical formulas and the notion of process correctness. When making such decisions our aim was definitions with pleasant consequences for all three views of concurrent processes.

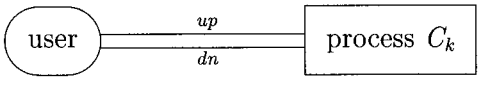
1.1 Three Views: an Example

As a first contact with this theory let us discuss an example inspired by Hoare [Hoa85b]. We wish to describe a process that behaves like a counter, i.e. that stores a natural number which can be incremented and decremented.

More precisely, we shall describe a k-bounded counter where $k \geq 1$ is some given constant. A k-bounded counter

- stores at each moment an integer value v within the range $0 \leq v \leq k$; initially this value is $v = 0$.
- allows two operations "up" and "dn" (shorthand for "down") to be performed as follows :
  - "up" is defined only if $v < k$ holds; it increments v by 1.
  - "dn" is defined only if $v > 0$ holds; it decrements v by 1.

The idea is to model the k-bounded counter as a process C_k that can engage in communications *up* and *dn* with its user. This information may be despicted as a connection diagram showing two communication lines between the user and the process named *up* and *dn* :



The effect of C_k engaging in a communication *up* or *dn* should be the same as performing the operations "up" or "dn" on the value v stored by the counter. To describe this dynamic aspect of C_k we now present three different views.

Logical Formulas. The most abstract view is a specification of what the process C_k is supposed to do, i.e. of its intended communication behaviour with the user, but not of its internal structure. We represent the communication behaviour of a process simply as a set of finite communication sequences, usually called *histories* of *traces*. This set is described by a formula of a predicate logic with variables ranging over traces. We therefore talk of *trace formulas* and *trace logic*.

For a fixed $k \geq 1$ the process C_k can be specified by the trace formula

$$S_k = 0 \leq up\#h - dn\#h \leq k$$

where we use $up\#h$ to denote the number of *up*'s in a trace h and $dn\#h$ to denote the number of *dn*'s. The difference $up\#h - dn\#h$ represents the value v stored in C_k after a trace h has occured. Formally, S_k describes the set of all traces h of communications *up* and *dn* such that for h and all its prefixes the value $v = up\#h - dn\#h$ is kept within the range $0 \leq v \leq k$.

For example, taking $k = 2$ we see that

$$\epsilon,\ up,\ up.dn \text{ and } up.up.dn$$

are all in the set of traces described by S_2, but

$$dn,\ dn.up,\ up.dn.dn$$

are not. Note that $h = dn.up$ itself satisfies the condition $0 \leq up\#h - dn\#h \leq 2$ but not its prefix *dn*. That is why h is not in the set of traces described by S_2.

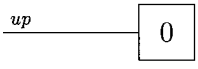
We say a process C_k satisfies the specification S_k if the following two conditions hold:

(1) the process C_k may only engage in traces that occur in the set described by S_k,
(2) if the user insists on, the process C_k must engage in every trace that occurs in the set described by S_k.

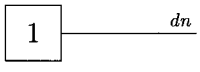
The first condition is a so-called *safety property* of C_k because it prevents C_k from doing something wrong, viz. engaging in a trace not allowed by S_k. The second condition is a so-called *liveness property* of C_k because it requires C_k to be responsive to certain communication requests from the user.

Process Terms. This view is more detailed; it describes how a process C_k satisfying S_k can be expressed as an abstract concurrent program. We call these programs *process terms* because they are generated by a small set of process operators.

Let us start with the case $k = 1$. According to S_1 the intended behaviour of C_1 is as follows. Initially, C_1 stores the value 0. Then only the communication up is possible. We picture the states of C_1 by drawing a box with the currently stored value inside and the currently enabled communication as lines outside:



After performing up the process C_1 stores the value 1. Then only the communication dn is possible:



Performing now dn brings C_1 back to its initial state:

A process C_1 exhibiting this behaviour can be expressed by the recursive equation

$$C_1 = up.dn.C_1. \tag{1.1}$$

For any process P the term $up.P$ denotes a process that first communicates up and then behaves like P. Similarly, for any process Q the term $dn.Q$ denotes a process that first communicates dn and then behaves like Q. Thus $up.dn.C_1$ denotes a process that first communicates up then dn and then behaves like C_1. By the recursive equation for C_1, this actually means that after the communication dn the process is ready for a communication up and then dn again etc. Thus C_1 denotes a process that can engage in the traces

$$\epsilon,\ up,\ up.dn,\ up.dn.up,\ up.dn.up.dn,\ \ldots$$

with its user. This is exactly the communication behaviour specified by the trace formula S_1.

In the syntax of process terms we shall use an explicit recursive construct instead of the recursive equation (1.1): viz.

$$C_1 = \mu X.up.dn.X. \tag{1.2}$$

Here X is a process identifier which stands for a recursive call of the term $up.dn.X$ prefixed by μX.

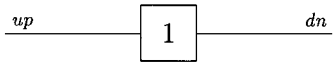
One of the objectives of this book is to show how a process term like C_1 can be constructed systematically from a trace formula like S_1. To this end, we shall develop *transformation rules* that transform trace formulas stepwise into process terms. In case of S_1 the construction consists of three main steps which we shall represent as a chain of three equalities:

$$\begin{array}{c} S_1 \\ \| \| \\ up.S_{1,up} \\ \| \| \\ up.dn.S_1 \\ \| \| \\ \mu X.up.dn.X \end{array} \tag{1.3}$$

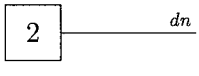
Each step corresponds to the application of one or more transformation rules. The initial step of the construction (1.3) starts with the trace formula S_1 and the last step ends with the process term C_1. In between we see two so-called *mixed terms*, i.e. syntactic constructs mixing process terms with trace formulas. Intuitively, the mixed term $up.S_{1,up}$ denotes a process that first communicates up and then behaves as specified by the trace formula $S_{1,up}$. Here $S_{1,up}$ is a variant of the formula S_1 which is obtained by performing a certain substitution on S_1. The details are given by a corresponding transformation rule.

Let us now turn to the case $k = 2$. According to S_2 the intended behaviour of C_2 is as follows. Initially, C_2 stores the value 0 and only allows the communication up to take place:

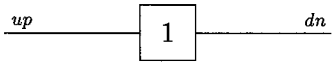
After performing up the process C_2 stores the value 1. Now both the communication up and dn are enabled:



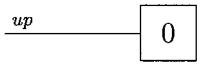
If the user chooses to engage in the communication up, the process C_2 changes its internal value to 2. Then only the communication dn is possible:



After performing dn the process C_1 is back in the state where 1 is the stored value and both communications up and dn are enabled:



If now the user chooses to engage in the communication dn, the process C_2 returns to its initial state



.

How can we represent this communication behaviour as a process term? Let us first explain a simple direct construction. We introduce three process identifiers named $ZERO$, ONE and TWO to represent the process C_2 being in a state where its internal value is 0,1 or 2. For each of these states we explore the next possible communication. This leads to the following system of recursive equations:

$$\begin{aligned} C_2 &= ZERO \\ ZERO &= up.ONE \\ ONE &= up.TWO + dn.ZERO \\ TWO &= dn.ONE \end{aligned} \tag{1.4}$$

For any processes P and Q the term $P+Q$ represents a choice between P and Q. More precisely, $P + Q$ denotes a process that behaves like P or Q depending on whether the first communication is one of P or one of Q. Thus the process ONE behaves like process $up.TWO$ if up is communicated and like $dn.ZERO$ if dn is communicated. Whether up or dn is chosen depends on the user of the process ONE.

By a simple substitution we can simplify (1.4) into the following system:

$$\begin{aligned} C_2 &= ZERO \\ ZERO &= up.ONE \\ ONE &= up.dn.ONE + dn.ZERO \end{aligned} \tag{1.5}$$

By introducing two nested recursive μ-constructs, we express (1.5) in the syntax of process terms:

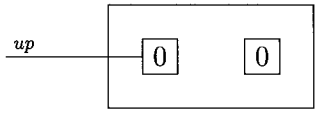
$$C_2 = \mu X.up.\mu Y.(up.dn.X + dn.Y) \tag{1.6}$$

Later we shall construct this process term C_2 systematically from the trace formula S_2 by applying transformation rules on mixed terms. Essentially, this construction will first discover the system (1.5) of recursive equations from S_2 and then express this as the process term (1.6).

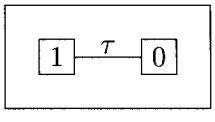
Let us now present a second, rather different construction of C_2. The idea is to combine two copies of the 1-bounded counter C_1 such that the communication line for dn of the first counter is linked to the communication line for up of the second counter. A communication on the link between dn and up is enabled if it is enabled on both

communication lines *dn* and *up* simultaneously. When enabled such a communication occurs as an internal action, i.e. without participation and even observation of the user. For the user only the external communications *up* and *dn* are visible.

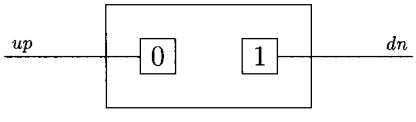
We picture this construction of C_2 as a big box with two smaller boxes inside, one for each copy of C_1. A state of C_2 is represented by putting the values that are currently stored in the copies of C_1 inside the smaller boxes and by drawing the (external and internal) communications as lines to or between the smaller boxes. For example, the initial state of C_2 is pictured as follows:



This picture shows that only the communication *up* is enabled and that it will effect the first copy of C_1. After performing *up* the first copy of C_1 stores the value 1 whereas the second stays at 0. At this state no external communication is possible because neither the first copy of C_1 can engage in *up* nor the second one in *dn*. This is undesirable from the user's point of view. However, now an internal communication on the link between the two copies of C_1 is possible because at the first copy the communication *dn* is enabled and at the second copy the communication *up*. Following Milner, we use the letter τ to indicate internal actions. Hence we picture the current state of C_2 as follows:

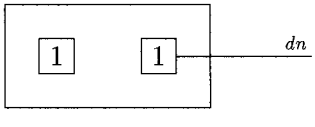


This is a so-called *instable* and *transient* state of C_2 because the internal communication τ is enabled and will proceed autonomously at a certain speed. This brings C_2 into the following stable state:

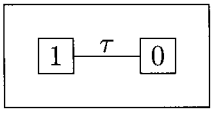


Now the communications *up* and *dn* are both enabled as the user would expect from a 2-bounded counter after having performed the initial communication *up*.

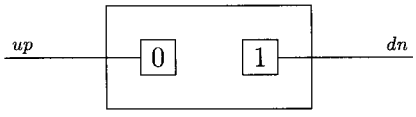
If the user chooses to engage in the comminication *up*, the process C_2 gets into a state where both copies of C_1 store the value 1. Then only the communication *dn* is enabled:



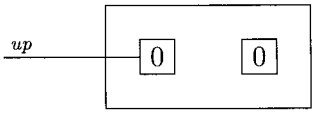
After performing dn the process C_2 enters the same instable state as above:



Again the internal communication τ proceeds automatically, which brings C_2 into the following stable state:



If now the user chooses to engage in the communication dn, the process C_2 returns to its initial state:



Here we have used pictures to describe the dynamic behaviour of this second construction of C_2. Now we explain how this construction can be expressed in the syntax of process terms.

We start from the process term

$$C_1 \;=\; \mu X.up.dn.X \tag{1.2}$$

constructed earlier and produce two copies of it by using a new communication lk (shorthand for *link*) and a renaming operator:

$$C_1[lk/dn] \text{ and } C_1[lk/up]. \tag{1.7}$$

For any process P and any two communications a,b the term $P[b/a]$ denotes a process that behaves like P but with all communications a renamed to b. Next, we apply parallel composition to these two copies yielding

$$C_1[lk/dn] \parallel C_1[lk/up]. \tag{1.8}$$

For any two processes P and Q the term $P\|Q$ denotes a process that behaves like P and Q working independently or concurrently except that all communications that occur in both P and Q have to be synchronised. For (1.8) this means that both copies of C_1 have to synchronise on the new communication lk. Note that (1.8) denotes a process that can engage in three communications, viz. up, dn and lk.

To transform the communication lk into an internal action that proceeds autonomously and invisibly for the user, we finally apply a hiding operator to lk. This brings us to a process term $C_2{}^*$ expressing the second construction of the 2-bounded counter:

$$C_2{}^* = (C_1[lk/dn] \| C_1[lk/up]) \backslash lk.$$

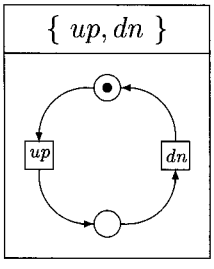
In general, for any process P and any communication b, the term $P\backslash b$ denotes a process that behaves like P but with all communications b transformed into internal actions.

Petri Nets. The most detailed view describes the operational machine behaviour of a process. To this end, we use Petri nets or, more precisely, *labelled place/transition nets*. Petri nets are easy to understand because they are a simple extension of the classical notion of an automaton. This extension deals with the explicit representation of concurrency.

Like automata, Petri nets have a graphical representation. We draw a box subdivided into an upper part displaying a set of communications, the *alphabet* of the net, and a lower part displaying the *flow graph* of the net. This is a directed graph with two types of nodes: *places*, some of them marked by one or more so-called tokens, and *transitions*, all of them labelled either by a communication of the alphabet of the net or by the symbol τ. Places are represented as circles and transitions as boxes.

The 1-bounded counter can be represented by the following Petri net $\mathcal{N}_1$:

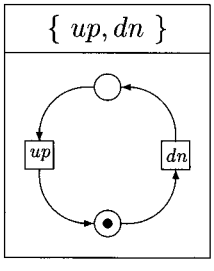
$\mathcal{N}_1 =$ (1.9)



In a Petri net a transition t is *enabled* if all places that are connected with t via an ingoing arc are marked by at least one token. In $\mathcal{N}_1$ the transition labelled with the communication *up* is enabled. *Executing* an enabled transition t results in removing one token from each place connected with t via an ingoing arc and adding one token to each place connected with t via an outgoing arc.

Thus executing the transition labelled with *up* in $\mathcal{N}_1$ yields the following new Petri net:

(1.10)



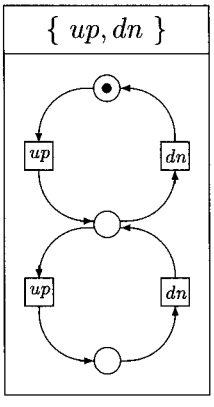
It has the same overall structure as $\mathcal{N}_1$ only that now the lower place is marked. Hence the transition labelled with *dn* is now enabled. Its execution brings us back to the Petri net $\mathcal{N}_1$.

The distribution of tokens in a Petri net is called a *marking*. We say that the Petri net (1.10) exhibits $\mathcal{N}_1$ at a different marking. Thus $\mathcal{N}_1$ has two markings: the initial marking shown in (1.9) represents the 1-bounded counter in a state where it stores the value 0 and where only the communication *up* is possible, and the second marking shown in (1.10) represents the counter in a state where it stores the value 1 and where only the communication *dn* is possible. Thus the Petri net $\mathcal{N}_1$ realises exactly the communication behaviour as described by the process term C_1.

One of the objectives of this book is to show how a Petri net like $\mathcal{N}_1$ can be constructed systematically from a process term like C_1. This will be done with the help of a so-called *Petri net semantics* for process terms. Combining the construction of $\mathcal{N}_1$ from C_1 with the construction of C_1 from the trace formula S_1 considered in (1.3), we obtain a construction of $\mathcal{N}_1$ from S_1 so that $\mathcal{N}_1$ realises exactly the communication behaviour specified by S_1.

Using the Petri net semantics, we can also construct Petri nets for the process terms C_2 and $C_2{}^*$. These nets describe two different realisations of a 2-bounded counter. For C_2 we obtain the following Petri net $\mathcal{N}_2$:

$\mathcal{N}_2 =$ (1.11)



In this initial marking the upper transition labelled with *up* is enabled. Its execution yields the following marking of $\mathcal{N}_2$:

(1.12)