Compositionality and modularity in process specification and design : a trace-state based approach

Citation for published version (APA):

Zwiers, J., & Roever, de, W. P. (1989). *Compositionality and modularity in process specification and design : a trace-state based approach*. (Computing science notes; Vol. 8904). Technische Universiteit Eindhoven.

Document status and date:

Published: 01/01/1989

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

**Compositionality and modularity in
process specification and design:
A trace-state based approach**

by

Job Zwiers and Willem-P. de Roever

89/4

April, 1989

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
All rights reserved
editors: prof.dr.M.Rem
 prof.dr.K.M.van Hee.

**Compositionality and Modularity
in Process Specification and Design:
A Trace-State Based Approach**

by

Job Zwiers

and

Willem-P. de Roever

CSN 88/

July 1988

Compositionality and Modularity in Process Specification and Design: A Trace-State Based Approach

Job Zwiers
Philips Research Laboratories
P.O. Box 80.000, 5600 JA Eindhoven
The Netherlands

Willem-P. de Roever
Department of Mathematics & Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands

Abstract

A top down development is presented of a distributed priority queue. The crucial characteristic that enables this development is the *compositionality* of the formalism used. Actually we argue that a stronger requirement is needed which combines compositionality with the ability to *adapt* given specifications, called *modularity*.

1 Introduction

A top down development is presented of a distributed priority queue within two formalisms that are based on states and histories of communications (called traces). The crucial characteristic that enables this development is the *compositionality* of the formalisms, i.e. the property that the specification of a composite program can be derived from the specifications of its constituent parts *without knowledge of their internal structure*, thus enabling top down development indeed.

During this development a communication buffer is derived. This leads up to the practical question how to integrate a priori given modules, such as the communication buffer in question, into the development process. Technically this imposes a stronger requirement upon our development formalisms which combines compositionality with the possibility to *adapt* given specifications. Systems allowing this combination are called *modular*. A small example of specification adaptation is given.

The top level specification for our example is of the form $P \text{ sat } \varphi$, meaning that all possible computations of program, P should satisfy assertion φ . At first the development uses specifications of this type. Then the need is felt to decompose *sequentially*, an operation which, since it does not fit nicely into *sat* based formalisms, is the starting point for Hoare's traditional pre/post style of specifications, but now in concurrent setting. After introducing this style in the second phase of our development process we immediately refine it to one in which the specification of the communication interface is separated from that for the other observables, i.e., the initial and final states and traces.

These concepts are illustrated for a restricted class of formalisms aimed at specifying safety properties. (Consult [PJ] for a similar setting but aimed at liveness).

2 Specification and Construction of Processes

The object of study of this report is the specification, construction and verification of parallel executing processes. In this section we would like to give an overview by showing the development of a parallel sorting algorithm. This gives us the opportunity to introduce the main programming language constructs, various specification methods, and some of the verification rules in an informal setting.

If one wants to reason in a secure way about processes then the first step is to introduce *formalized languages* to describe processes and specifications. Our main formal language to describe processes is called **TNP**.

The language **TNP**, for Theoretical Networks of Processes, evolved from an earlier language called **DNP**, for Dynamic Networks of Processes. The starting point for **DNP** was the concept of a dynamically changing networks of processes. A parallel network consists of a number of *processes* executing in parallel and *communicating* messages along interconnecting channels. A process is a sequential program that can *expand* temporarily into parallel subnetworks. This can happen recursively since the so formed subnetworks can contain new copies of the original (expanded) process.

By generalizing, and at the same time simplifying, **DNP** the language **TNP** was designed. Whereas **DNP** resembles a procedure based parallel programming language, **TNP** is much more in the style of languages such as TCSP. [OH].

We shall introduce the main language constructs of **TNP** as we need them in the development of a parallel sorting algorithm, also known as "priority queue" or, as we shall call it, the "sorted bag".

The parallel execution of two processes P_1 and P_2 is denoted by $P_1 \parallel P_2$. The CSP notation $c!e$ is used to denote the *sending* of the message denoted by e along the channel named c . Similarly, $c?x$ is a command that *requests* some message along c and stores it into variable x . Communication is *synchronous*, that is, the sender and receiver of a message must cooperate and, conceptually, a message is received at the same time as it was sent.

Example 1

We give a picture of the network $P \parallel Bag$, consisting of two processes named P and Bag . The two component processes are connected by means of channels named *insert* and *getmin*. Process P can send messages to Bag along channel *insert* by executing *insert!e* commands. For such communications to occur the Bag process must execute corresponding commands of the form *insert?x*. Similarly a communication along *getmin* occurs if P executes a *getmin?x* command in cooperation with a *getmin!e* command of Bag .

□

We want to *specify* the intended behavior of processes and to *verify* that the specification is met by some proposed implementation. For instance a description of the intended behavior of the Bag process could be the following:

Example 2

“ Bag behaves as a so called sorted bag of values. Bags are also called multisets since they are like sets except that multiple copies of the same value can be member of a bag. New values can be inserted by sending them to Bag via the *insert* channel. A value can be requested via the *getmin* channel and this results in sending back and removal from the bag of the smallest element of the bag.”

□

As is well known, a Bag process as described can be used to sort a given list of values. This is done by first inserting all elements of the list, followed by requesting an equal number of values.

We want to specify the Bag behavior in a rigorous way, rather than informally as in the English description above. Our formal description method is based upon the notions of *communication events* and *sequences* of such events called *traces* or *communication histories*.

A communication is an event that is described by a channel name and a communicated value. For instance, the event of communicating a value v via channel *insert* is described by the pair $(insert, v)$. A communication history h then, can be seen as a description of which values have been communicated along which channels in which order at some particular moment of time. For instance, the history $h \stackrel{\text{def}}{=} \langle (insert, v), (insert, w), (getmin, v) \rangle$ corresponds to the state where values v and w were sent, in that order, via *insert* followed by the communication of value v via *getmin*.

In essence a process specification can be seen as a formula of a (many sorted, first order) predicate logic that expresses the desired properties of the communication history h that should hold for all possible executions of the process, at any moment during the execution. Such a predicate is called an *assertion* about the behavior of the process.

We would like to give an impression of what a formal specification for the *Bag* process looks like. To this end we discuss a few aspects of the *language* of assertions.

Fundamental is the class of trace expressions. Examples are the *empty trace* ε , denoting the empty sequence of communications, and the *communication history* h , denoting the communications that have been performed by the process up to some moment during some execution of the process. The history h must be distinguished from ordinary trace valued *variables* t that denote some arbitrary sequence of communications, not especially related to those communications actually performed by the process. Operations on trace expressions te include the important *channel projection* operation $te|_{\{c_1, \dots, c_n\}}$. This denotes the subsequence of te consisting of all communications via the channels $\{c_1, \dots, c_n\}$. The special case $h|_{\{c\}}$ is often abbreviated as c , that is, a channel name, used as a trace expression, denotes the sequence of all communications via that particular channel. Operations that we shall use in the example below are $last(te)$, denoting the last communication of te , and $rest(te)$, denoting the sequence of all *but* the last communication of te .

The expression $last(te)$ is not a trace expression since it denotes a single communication rather than a sequence of communications. The channel name and the communicated value of a single communication α are referred to by the expressions $chan(\alpha)$ and $val(\alpha)$.

Assertions are many sorted first order predicate formulae. Here we only mention one such assertion, denoted by $te_1 \leq te_2$, which expresses that the sequence te_1 is an *initial prefix* of the sequence te_2 .

The fact that one requires an assertion χ to hold for *all possible* executions of a process P is not expressed within the assertion as such. Rather this is the interpretation of the satisfaction relation between processes and assertions. This is expressed by the following formula:

$$P \text{ sat } \chi.$$

We call such formulae SAT specifications, to distinguish them from other types of specifications that we shall later on.

Example 3

We use the following notation for bags:

- The empty bag is denoted by $\emptyset$, the bag containing elements $e_1, \dots, e_n$ by $[e_1, \dots, e_n]$.
- The union and difference of bags B_1 and B_2 are denoted by $B_1 \oplus B_2$ and $B_1 \ominus B_2$.
- The least element of a bag B of ordered values is denoted by $min(B)$.

Let " $bag(te, c)$ " denote the bag of all values of all communications via channel c that occur in the sequence denoted by trace expression te . Instead of $bag(h|\{c\}, c)$ we use the abbreviation $bag(c)$. Also we use $cont(c_1, c_2)$ as an abbreviation for $bag(c_1) \ominus bag(c_2)$, and $cont(te, c_1, c_2)$ as an abbreviation for $bag(te, c_1) \ominus bag(te, c_2)$, where te is some trace expression other than h .

Let us fix some arbitrary moment during the *Bag* execution. We want to express that for all *getmin* communications that occur in the sequence $h|\{insert, getmin\}$ reached thus far, the correct value, that is, the "current" least element at the moment in question, was sent back by the *Bag* process. Now assume that t is some prefix of $h|\{insert, getmin\}$ that ends with a *getmin* communication. We can express this assumption by means of the assertion

$$t \leq h|\{insert, getmin\} \wedge chan(last(t)) = getmin.$$

The "current contents" of the bag just before that last communication is given by the expression $cont(rest(t), insert, getmin)$. Therefore, the desired property is expressed by the assertion $\chi_{bag}(insert, getmin)$, defined as:

$$\begin{aligned} \chi_{bag}(insert, getmin) &\stackrel{\text{def}}{=} \\ \forall t \big((t \leq h|\{insert, getmin\} \wedge chan(last(t)) = getmin) \rightarrow \\ &val(last(t)) = min(cont(rest(t), insert, getmin)) \big). \end{aligned}$$

Finally, the *Bag* specification is the following SAT formula:

$$Bag \text{ sat } \chi_{bag}(insert, getmin).$$

□

Apart from operating as a sequential communicating program a process can be built up as a subnetwork. For instance we might implement the *Bag* process as a buffer process *Buf*, keeping a few *Bag* elements amongst which the current least element, executing in parallel with a process *Bag'* keeping the rest of the *Bag* elements. This is shown in the picture below.

As can be seen the channels *insert* and *getmin* are connected to the *Buf* component of the network. The two components themselves are connected by means of channels

down and *up*. The idea is that *Buf* on request will send its least element along *getmin*. The *Buf* process can also send elements "down" to, or request the least element contained in, the *Bag'* process. So the *Bag'* process must behave as a copy of the *Bag* process except that its channels have different names.

We would like to turn the intuitive descriptions of *Bag'* and *Buf* into formal specifications, and then *verify* that the parallel network $Buf \parallel Bag'$ does conform to the *Bag* specification. To this end we first discuss the verification principles for parallel processes.

A well known problem for the verification of parallel programs is that some specification that would be correct for a given process viewed in isolation, might be invalidated by the actions performed by other processes executing in parallel. In particular this is the case when a specification for a process *P* refers to a channel that can be modified by other processes without the cooperation of *P*. For instance, if a specification for the *Bag'* process would refer to the *insert* or *getmin* channel, then communications on these channels performed by the parallel executing *Buf* process, without cooperation with *Bag'*, could be in conflict with this specification. We shall avoid such specifications, for only then the soundness is implied of the following simple proof rule for parallelism:

Let $P_1 \text{ sat } \chi_1$ and $P_2 \text{ sat } \chi_2$ be specifications of the communication behavior of P_1 and P_2 that obey the restriction that the assertion χ_i only refers to communications via channels *connected to* process P_i , where $i = 1, 2$. Then from $P_1 \text{ sat } \chi_1$ and $P_2 \text{ sat } \chi_2$ one can infer the following specification for the parallel composition of the two processes:

$$P_1 \parallel P_2 \text{ sat } (\chi_1 \wedge \chi_2).$$

We took care that the *Bag* specification does obey this restriction, since the only reference to the communication history is by means of the *projection* of this history onto the channels *insert* and *getmin*. A communication performed via some other channel than those two by some other process *does* affect the value of *h*, but it does *not* affect the value of $h|_{\{\text{insert}, \text{getmin}\}}$.

To verify the correctness of the parallel composition $Buf \parallel Bag'$, one has essentially to prove that the *Buf* process preserves $\chi_{bag}(\text{insert}, \text{getmin})$. For, during the expansion, the *insert* and *getmin* channels are connected to the *Buf* process. Of course the fact that *Buf* conforms to this behavior depends on the *assumption* that the process *Bag'*, executing in parallel, behaves correctly as a bag with respect to the channels *down* and *up*, that is, satisfies the specification $\chi_{bag}(\text{down}, \text{up})$. Therefore we choose the following *Buf* specification:

$$Buf \text{ sat } (\chi_{bag}(\text{down}, \text{up}) \rightarrow \chi_{bag}(\text{insert}, \text{getmin})).$$

By the rule for parallel composition we then infer that:

$$Buf \parallel Bag' \text{ sat } ((\chi_{bag}(\text{down}, \text{up}) \rightarrow \chi_{bag}(\text{insert}, \text{getmin})) \wedge \chi_{bag}(\text{down}, \text{up})).$$

Since the assertion of this last formula clearly implies $\chi_{bag}(\text{insert}, \text{getmin})$, We have shown that $Buf \parallel Bag'$ does satisfy the required assertion.

The fact that the *Bag'* process shows the behavior of a *Bag* process, except that its channels have different names, suggests that the *Bag'* process can be implemented as a parallel network itself and so on, ad infinitum. However, instead of such an infinitely deep nested static network, we prefer a *dynamic* network in which *Bag* starts as a *sequential* process and *expands* into a subnetwork only after elements have been inserted into the bag. Moreover, we can construct the *Bag'* process from a *recursive copy* of the *Bag* process itself, and so we will obtain a *Bag* process that has a variable but finite degree of nesting, dependent on the number of elements contained in it. To denote that, within the program *S* that we shall use to implement *Bag*, recursive copies of *Bag* are allowed we use the *recursion construct* $\mu Bag.S$. Apart from recursion we need a few more language constructs. First of all, we must *rename* the channels of the recursive *Bag* copy into *up* and *down*. This can be denoted in our language TNP as $Bag\langle down/insert, up/getmin \rangle$. The effect is that messages sent along *getmin* by *Bag* appear from outside to be messages sent along the *up* channel. Similarly, messages sent to $Bag\langle down/insert, up/getmin \rangle$ via *down* are received by the *Bag* process via its *insert* channel. Our first approach to implement *Bag* can now be given:

$$\mu Bag \cdot insert?x ; (Buf \parallel Bag\langle down/insert, up/getmin \rangle)$$

The semicolon, as usual, denotes *sequential composition*. So first a value has to be received before the process expands into a subnetwork as indicated. The *Buf* process has access to the *x* variable in which the received value is stored. For instance it could send it back via the *getmin* channel to the outside world. But now a problem arises, for the recursive *Bag* copy *also* accesses the *x* variable. To avoid such "clashes" the so called *variable hiding* construct must be used to turn *x* into a *local* variable of the *Bag* process. This construct has the form $S \backslash x$ and it indicates that *x* is a local variable of process *S*. We have the same problem with channel names: the *up* and *down* channels *connected* to the recursive *Bag* copy have nothing to do with the channels of the same name *within* this copy. The problem is solved by using the *channel hiding* construct of the form $S \backslash c$ that denotes that *c* is an internal channel of *S*, not visible from outside.

We arrive at the following implementation for the *Bag* process:

$$\mu Bag \cdot (insert?x ; (Buf \parallel Bag\langle down/insert, up/getmin \rangle) \backslash up, down) \backslash x$$

Our solution still has one defect. Incarnations of the *Buf* and the *Bag* processes never terminate, and so although new incarnations are created when necessary these incarnations do not disappear when they are no longer needed. Ideally a parallel network $Buf \parallel Bag\langle down/insert, up/getmin \rangle$ should vanish as soon as neither the *Buf* process nor the *Bag* process stores anymore values. So we cannot simply design the processes such that they terminate as soon as they store no value. To this end we *synchronize* the *Buf* and the *Bag* processes as follows. We include a new local channel called *isempty* between the two processes. If at any moment a process stores no values, it offers to communicate via *isempty* whereafter it will terminate. That is, if the other process stores no values either, and so is also able to communicate via *isempty*, then the parallel combination can terminate as a whole. On the

other hand if some new value is received then the process stores it and is no longer willing to communicate via *isempty*. In our program the commands *isempty!* and *isempty?* denote communication commands where no values are sent or received. That is, only *synchronization* is required. (Alternatively one might see this as ordinary communication where some immaterial value is passed.) This suggests our final implementation for the *Bag* process:

```

μBag. (isempty!
      or
      insert?x ;
      (Buf || Bag(down/insert, up/getmin)) \ up, down, isempty ;
      Bag
      ) \ x.

```

The process has the *choice* between sending an *isempty* signal followed by termination and receiving a value via *insert* followed by an expansion. Note that if the subnetwork created by this expansion terminates itself then the whole process starts over again, since the execution of the subnetwork is followed by a recursive incarnation. In fact this second recursive call has the form of a "tail" recursion, and so could have been replaced by a loop construct.

A picture of this process after an expansion, where we have used *Bag'* to denote *Bag*(*down/insert, up/getmin*) is:

The vertical bar at the end of the outer *isempty* channel indicates that during the expansion this channel is not connected to any of the sub processes. Only *after* the subnetwork has terminated the *Bag* process is (again) able to send an *isempty* signal.

The picture below shows an execution of the *Bag* process from a certain (unfinished) stage for which $\text{cont}(\text{insert}, \text{getmin}) = [1, 5, 7]$. We have only shown the *Buf* processes and the most deeply nested *Bag* process. We show also how a new value (3) is inserted, and how the current minimum (1) is requested. Note that, as seen from outside the minimum is requested strictly after the value 3 has been inserted, but that internally the *Bag* process is still busy with the insertion process at that time. (the exclamation mark in the one but last picture indicates a synchronization action via an "*isempty*" channel.)

We adapt our *Bag* specification so as to express the fact that the *Bag* process only can terminate *after* it has sent an *isempty* signal, and that the contents of the bag is empty indeed, on termination.

In the assertion language, we use the symbol $\top$ to denote the characteristic predicate for computations that have *terminated*. Computations that do *not* satisfy this $\top$ predicate are called *unfinished*, and correspond to intermediate stages of the execution of a process. Unfinished computations are not the same as nonterminating computations; *every* execution passes through unfinished intermediate stages whether it eventually terminates or not.

The specification $\phi_{bag}(insert, getmin)$ now becomes:

$$\phi_{bag}(insert, getmin) \stackrel{\text{def}}{=} \chi_{bag}(insert, getmin) \wedge (\top \rightarrow (cont(insert, getmin) = \emptyset \wedge isempty \neq \epsilon)).$$

The new specification still requires $\chi_{bag}(insert, getmin)$ to hold at *all* stages, and for terminated computations it requires the *Bag* contents to be empty, and the sequence of communications sent via *isempty* to be nonempty. The $\phi_{bag}(insert, getmin)$ does *not*, and even *cannot*, express that the process *must* terminate after it has sent the *isempty* signal. The reason is that we study specification methods for so called *safety properties*, and the necessity of termination is not one of those properties.

3 Hoare specifications and Invariant specifications

Now that we have proposed a top level design for *Bag*, we would like to verify that it satisfies the bag specification. We sketch how this verification could proceed. Instead of introducing formal proofrules here we rely on the plausibility of certain verification principles. We shall encounter certain difficulties in connection with the *sequential* structure of the process. To resolve them we introduce new types of process specifications.

A well known verification principle, called Scott's induction rule, implies that, to verify that *Bag* satisfies the specification $\phi_{bag}(insert, getmin)$, it suffices to verify that the *body* of *Bag* does satisfy this specification, where we may assume "by induction" that the two occurrences of *Bag* processes within the body do satisfy $\phi_{bag}(insert, getmin)$. That is, under the hypothesis that *Bag* satisfies $\phi_{bag}(insert, getmin)$, we must verify the same assertion for the process:

```
( isempty!
or insert?x ;
  (Buf || Bag(down/insert, up/getmin)) \ {up, down, isempty};
  Bag
) \ {x}
```

To do so, we need verification principles for the channel and variable hiding constructs. Now these are fairly simple: if a process satisfies a certain assertion χ , and the assertion *does not refer to* some channel *c* or variable *x*, then the assertion remains valid after we hide this channel or variable.

It will be clear that in general such specifications can be obtained only by using appropriate *projections* of the communication history.

Since our specification $\phi_{bag}(insert, getmin)$ does not refer to the variable *x*, our verification task for the body of *Bag* boils down to the verification of $\phi_{bag}(insert, getmin)$ for both components of the choice construct. We concentrate on the verification for the second component, that is, of:

$(insert?x; (Buf \parallel Bag(down/insert, up/getmin)) \backslash up, down, isempty; Bag)$
 sat
 $\phi_{bag}(insert, getmin).$

The process of this last formula has the form of a sequential composition $P_1; P_2; Bag$, where P_1 is the process $insert?x$ and where P_2 is:

$(Buf \parallel Bag(down/insert, up/getmin)) \backslash up, down, isempty$

First we must establish specifications χ_1 and χ_2 , such that $P_1 sat \chi_1$ and that $P_2 sat \chi_2$.

This brings us to the *state transformer* aspect of our specifications. For instance, the P_1 process does not only *communicate* some value along the *insert* channel, but it also *modifies the process state* by storing the received value in x . As a consequence the specification χ_1 must describe this state transformation together with, and related to, the communication behavior. Actually this relationship is very simple; it is: $x = val(last(insert))$. This relationship between x and the value communicated via the *insert* channel exists only *after the P_1 process has terminated*. Therefore we must use again the $\top$ predicate, indicating a terminated computation. The specification for the P_1 process then becomes: $P_1 sat (\top \rightarrow x = val(last(insert)))$.

The last specification expresses a certain relationship between the communication behavior and the *final state* of computations. But in general one must also refer to the initial state in which the computation starts. For instance, for the process $down!x; up?x$ one must be able to specify that the value communicated via *down* is the *initial state value* of x , whereas the *final state value* of x is the value communicated via *up*. We use a "^o" superscript to indicate an initial state value of some variable. For example,

$(down!x; up?x) sat (val(last(down)) = x^o \wedge val(last(up)) = x)$

Our next problem is to invent a specification for the parallel construct. Because this parallel process starts executing only after a value has been sent via *insert*, the original bag specification is no longer applicable.

If our goal would be the verification of the *Bag* version that we considered in the previous section just before the introduction of the *isempty* channel, then we would know how to proceed. In this case, any communication history of the process $Buf \parallel Bag(down/insert, up/getmin)$, prefixed by a single communication $(insert, v)$ where v is arbitrary, should satisfy the *Bag* assertion $\phi_{bag}(insert, getmin)$. This means that the appropriate assertion for the parallel construct would be:

$\forall v (\phi_{bag}(insert, getmin)[(< (insert, v) > h) / h]),$

where $\dots [(< (insert, v) > h) / h]$ indicates that the concatenation of the one element trace $< (insert, v) >$ with the history h must be substituted for h .

This simple verification principle is well known from the literature, see for instance [ZH].

However, for the final version of *Bag*, we must not only account for the single communication via *insert* *before* the parallel construct starts, but also for the communications

by the recursive *Bag* call *after* the parallel construct has terminated. With respect to the communication history of the recursive *Bag* call we may assume, by Scott's induction rule, that it satisfies the bag specification. Unfortunately there is no elegant way to combine two assertions that describe the components of a sequential composition of processes. The reason why we succeeded in the situation above was that the communication histories of one of the two components were fairly simple.

In general one must construct a formula χ that describes the execution of our sequential process $P_1; P_2; Bag$. Informally speaking, this formula is:

"There are intermediate values x' and x'' for x , and subsequences h_1, h_2, h_3 of the complete communication history h , such that h is the concatenation of these three histories, χ_1 holds for the communications in h_1 and the transition of the x value from x° to x' , χ_2 holds for h_2 and the transition from x' to x'' , and $\phi_{bag}(insert, getmin)$ holds for h_3 and the transition from x'' to x ."

In the assertion language, it is possible to write down a predicate logic formula with the intended interpretation as above. This formula is abbreviated as:

$$\chi_1 \hat{\circ} \chi_2 \hat{\circ} \phi_{bag}(insert, getmin).$$

The second and final step in the verification of $P_1; P_2; Bag \text{ sat } \phi_{bag}(insert, getmin)$ is then to show that

$$(\chi_1 \hat{\circ} \chi_2 \hat{\circ} \phi_{bag}(insert, getmin)) \rightarrow \phi_{bag}(insert, getmin)$$

is a valid implication.

Let us compare this with the well known rule for sequential composition in Hoare's logic for sequential, i.e. non parallel, programs. The specifications for programs S are in the form of pre- and postconditions, denoted by $\{pre\} S \{post\}$. The pre- and postconditions are assertions on the initial and final state of the computation. To verify $\{pre\} S_1; S_2 \{post\}$ one must invent an *intermediate* assertion *int* and verify $\{pre\} S_1 \{int\}$ and $\{int\} S_2 \{post\}$.

Since there is no complicated verification condition to validate, like the implication $(\chi_1 \hat{\circ} \chi_2 \hat{\circ} \phi_{bag}(insert, getmin)) \rightarrow \phi_{bag}(insert, getmin)$, we prefer this type of reasoning over the approach sketched above.

The observation that formulae of the form $P \text{ sat } \chi$ are not very appropriate for the sequential composition construct formed the incentive for studying a new style of process specifications. These specifications are based upon the idea that if some process S_2 starts after termination of some other process S_1 , then at that moment of time there is already some communication history. We call this the *initial trace* for the execution of S_2 . By introducing this notion of an initial communication history, we can view a process as a transformer from *initial* histories and - process states to *final* histories and process states. We use the phrase "generalized state" for a combination of some history and some process state. The outer form of our specifications is the same as that of "classical" Hoare style formulae, with a pre- and postcondition. These pre- and postconditions are assertions on *generalized* states, however. To indicate this, we use the notation $(\varphi) P (\psi)$ for Hoare specifications,

where the φ and ψ are assertions on generalized states. With our Hoare formulae, one can essentially use the same type of reasoning for sequential programs as within Hoare's logic. In particular the proof rule for sequential composition of processes has the same form as the classical Hoare rule mentioned above.

Our Hoare style specifications have in common with SAT specifications the uniform treatment of terminated and unfinished computations. That is, the assertions φ and ψ of the Hoare formula $(\varphi) P (\psi)$ and the assertion χ of the SAT formula $P \text{ sat } \chi$ are interpreted for both terminated and unfinished computations. The characteristic predicate $\top$ is used within these assertions to distinguish between the two types of computations.

In practice it turns out to be easier to *separate* assertions into pre- and postconditions on generalized states that are interpreted for terminated computations only, and invariants on communication histories, that must hold continuously. This leads to a third type of formulae, called "Invariant specifications". They have the following form:

$$I : \{pre\} P \{post\},$$

where I is an assertion on histories only, and where pre and $post$ are assertions on generalized states, i.e. on histories and states together. The characteristic predicate $\top$ is not used within I, pre or $post$. The distinction between terminated and unfinished computations is made in the interpretation of the specification. Informally, the meaning of the specification is:

"If P is started in an initial state and with an initial history for which pre holds, then the invariant I will hold for the communication history of P at any moment during execution of P , and if and when P terminates, $post$ will hold for the final history and state of P ."

Remarks

- By "the communication history of P " we mean the complete history, that is, the initial history with communications performed by P at some moment during execution, *not* just the communications performed by P itself.
- It is understood that, if P terminates, the invariant holds up to *and including* the moment of termination.

□

We proceed with the *Bag* example, turning over to the Invariant specification style. First we transform the SAT specification $Bag \text{ sat } \phi_{bag}(insert, getmin)$ into an equivalent Invariant specification. A straightforward transformation is obtained by choosing a precondition that requires the initial trace to be *empty*, for then the role of the invariant is essentially the same as an assertion of a SAT formula. This would result in the following Invariant specification:

$\chi_{bag}(insert, getmin) :$

$\{insert = \epsilon \wedge getmin = \epsilon \wedge isempty = \epsilon\}$

Bag

$\{cont(insert, getmin) = \emptyset \wedge isempty \neq \epsilon\}.$

We have not included the assertion $\chi_{bag}(insert, getmin)$ in the postcondition, although it would have made no difference since the invariant requires it anyhow.

We are not satisfied with this specification for the following reason. Its precondition does not hold at those moments where the *inner recursive calls* of the *Bag* process start executing. So the given specification, although it correctly specifies the desired process behavior, will not fit into the correctness proof that we have in mind. As is not unusual with induction proofs, we can only prove a *stronger* specification. An Invariant specification can be made stronger by weakening its precondition, and by strengthening its postcondition and invariant. So the following formula is seen to be stronger than the specification above:

$\chi_{bag}(insert, getmin) :$

$\{cont(insert, getmin) = \emptyset \wedge \chi_{bag}(insert, getmin)\}$

Bag

$\{cont(insert, getmin) = \emptyset \wedge isempty \neq \epsilon \wedge \chi_{bag}(insert, getmin)\}.$

To prove this specification we give a so called *annotated program*. It consists of the program text of the *Bag* process with assertions on generalized states attached to control points of the process. Similar to the usual program annotations for Hoare's logic we enclose these assertions within set braces. The annotated text is preceded by a clause, called the invariant of the annotation, that has the form $\chi_{bag}(insert, getmin) : .$ Such an annotated text is to be understood as follows. If S is some piece of process text that occurs within an annotation that is prefixed by invariant I , and S is enclosed between the assertions p and q , then

$I : \{p \wedge I\} S \{q \wedge I\}$

is claimed to be a valid specification for S . (The reason that the invariant I is implicitly attached as a conjunct to all assertions within the annotations is just a notational convenience.)

The actual annotation for *Bag* is:

$$\begin{aligned}
& \chi_{bag}(insert, getmin) : \\
& \{cont(insert, getmin) = \emptyset\} \\
& (\text{ isempty!} \\
& \{cont(insert, getmin) = \emptyset \wedge isempty \neq \varepsilon\} \\
& \text{or} \\
& (\text{ insert?}x ; \\
& \quad \{cont(insert, getmin) = [x]\} \\
& \quad (Buf \parallel Bag(down/insert, up/getmin)) \setminus \{up, down, isempty\} ; \\
& \quad \{cont(insert, getmin) = \emptyset\} \\
& \quad Bag \\
&) \{cont(insert, getmin) = \emptyset \wedge isempty \neq \varepsilon\} \\
&) \setminus \{x\} \\
& \{cont(insert, getmin) = \emptyset \wedge isempty \neq \varepsilon\}
\end{aligned}$$

Note that the specification for the second inner call of the *Bag* process equals the specification for the whole. This can be formally justified by applying Scott's induction rule. The annotation does not indicate how to prove the specification that it claims to be valid for the parallel network within the *Bag* process. This specification is:

$$\begin{aligned}
& \chi_{bag}(insert, getmin) : \\
& \{cont(insert, getmin) = [x] \wedge \chi_{bag}(insert, getmin)\} \\
& (Buf \parallel Bag(down/insert, up/getmin)) \setminus \{up, down, isempty\} \\
& \{cont(insert, getmin) = \emptyset \wedge \chi_{bag}(insert, getmin)\}
\end{aligned}$$

We sketch how to prove this.

Again we may assume that the *Bag* specification holds for the recursive call within this network, and this implies the following for the *Bag* process with renamed channels:

$$\begin{aligned}
& \chi_{bag}(down, up) : \{cont(down, up) = \emptyset \wedge \chi_{bag}(down, up)\} \\
& \quad Bag(down/insert, up/getmin) \\
& \quad \{cont(down, up) = \emptyset \wedge isempty \neq \varepsilon \wedge \chi_{bag}(down, up)\}.
\end{aligned}$$

By strengthening the precondition and weakening the postcondition we obtain:

$$\begin{aligned}
& \chi_{bag}(down, up) : \{down = up = \varepsilon\} \\
& \quad Bag(down/insert, up/getmin) \\
& \quad \{cont(down, up) = \emptyset \wedge \chi_{bag}(down, up)\}.
\end{aligned}$$

The next task is to determine an appropriate specification for the *Buf* process. In our top down development, the most natural thing to do is to choose this specification such that it suits the verification of the parallel construct. Therefore we take the following one:

$$\chi_{bag}(down, up) \rightarrow \chi_{bag}(insert, getmin) :$$

$$\{cont(insert, getmin) = [x] \wedge \chi_{bag}(insert, getmin) \wedge up = down = \epsilon\}$$

Buf

$$\{cont(down, up) = \emptyset \rightarrow cont(insert, getmin) = \emptyset \wedge$$

$$\chi_{bag}(down, up) \rightarrow \wedge \chi_{bag}(insert, getmin)\}$$

We now have available specifications for the constituent components of the parallel construct. They satisfy the restriction that the channels and variables referred to within these specifications are the channels and variables of the corresponding process. Under these conditions the following rule is applicable:

Let $I_1 : \{pre_1\} P_1 \{post_1\}$ and $I_2 : \{pre_2\} P_2 \{post_2\}$ be Invariant specifications such that the channels and variables of I_i , pre_i and $post_i$ are contained within the channels and variables used by P_i , for $i = 1, 2$. Then the following formula can be inferred from these two specifications:

$$(I_1 \wedge I_2) : \{pre_1 \wedge pre_2\} P_1 \parallel P_2 \{post_1 \wedge post_2\}.$$

We can use this rule to conjoin the *Buf* and the *Bag* $\langle down/insert, up/getmin \rangle$ specifications. Then, by weakening the invariant and postcondition we obtain the following specification for the parallel construct:

$$\chi_{bag}(insert, getmin) :$$

$$\{cont(insert, getmin) = [x] \wedge \chi_{bag}(insert, getmin) \wedge up = down = \epsilon\}$$

$$(Buf \parallel Bag\langle down/insert, up/getmin \rangle)$$

$$\{cont(insert, getmin) = \emptyset \wedge \chi_{bag}(insert, getmin)\}$$

Let us omit the conjunct $up = down = \epsilon$ from the precondition. That is, we no longer assume that the local channels *up* and *down* are initially empty. Now, unlike the initial state of *variables*, the initial state of *channels* cannot be sensed in any way by a process, and so the possible communication histories are the same as far as the projection onto channels *other* than *up* and *down* is concerned. And because the invariant and postcondition *do not refer to the up and down channels*, they still remain valid for this larger set of possible communication histories. We conclude that the following formula is valid:

$$\chi_{bag}(insert, getmin) :$$

$$\{cont(insert, getmin) = [x] \wedge \chi_{bag}(insert, getmin)\}$$

$$(Buf \parallel Bag\langle down/insert, up/getmin \rangle)$$

$$\{cont(insert, getmin) = \emptyset \wedge \chi_{bag}(insert, getmin)\}.$$

Since this specification does not refer anymore to the local channels *up*, *down* or *isempty*, it remains valid if we hide these local channels. Therefore, we have shown the validity of:

$\chi_{bag}(insert, getmin) :$

$\{cont(insert, getmin) = [x] \wedge \chi_{bag}(insert, getmin)\}$

$(Buf \parallel Bag(down/insert, up/getmin)) \setminus \{up, down, isempty\}$

$\{cont(insert, getmin) = \emptyset \wedge \chi_{bag}(insert, getmin)\}$

This was to be shown.

We have succeeded in verifying the top level design for the *Bag* process. It remains to implement the *Buf* process, and to verify that the implementation does satisfy the specification that we chose for this component. This development does neither introduce new language constructs nor new verification principles. Therefore we simply list the resulting program text and the corresponding proof.

The proposed *Buf* implementation is:

$(insert?y ;$

$(x, y) := (min(x, y), max(x, y)) ; down!y ; Buf)$

or

$(getmin!x ;$

$((up?x ; Buf) \text{ or } isempty?))$

Similar to the verification of the *Bag* process it is necessary to prove a *stronger* specification for *Buf* than the one that we required above. Again, the reason is that the given specification is inappropriate as an induction hypothesis for Scott's induction rule.

The proof is in the form of an annotated program text:

Proof outline for the *Buf* process:

Invariant $\chi_{bag}(down, up) \rightarrow \chi_{bag}(insert, getmin) :$

$\{cont(insert, getmin) = cont(down, up) \oplus [x] \wedge$
 $\chi_{bag}(down, up) \rightarrow x = \min(cont(insert, getmin))\}$

(*insert*?*y* ;

$\{cont(insert, getmin) = cont(down, up) \oplus [x, y] \wedge$
 $\chi_{bag}(down, up) \rightarrow \min(x, y) = \min(cont(insert, getmin))\}$

(*x, y*) := (*min*(*x, y*), *max*(*x, y*)) ;

$\{cont(insert, getmin) = cont(down, up) \oplus [x, y] \wedge$
 $\chi_{bag}(down, up) \rightarrow x = \min(cont(insert, getmin))\}$

down!*y* ;

$\{cont(insert, getmin) = cont(down, up) \oplus [x] \wedge$
 $\chi_{bag}(down, up) \rightarrow x = \min(cont(insert, getmin))\}$

Buf

) {*cont*(*insert, getmin*) = *cont*(*down, up*)}

or

(*getmin*!*x* ;

{*cont*(*insert, getmin*) = *cont*(*down, up*)}

(*up*?*x* ;

$\{cont(insert, getmin) = cont(down, up) \oplus [x] \wedge$
 $\chi_{bag}(down, up) \rightarrow x = \min(cont(insert, getmin))\}$

Buf

) {*cont*(*insert, getmin*) = *cont*(*down, up*)}

or

isempty?

{*cont*(*insert, getmin*) = *cont*(*down, up*)}

)

{*cont*(*insert, getmin*) = *cont*(*down, up*)}

4 Compositionality and Modularity

For the task of program specification, construction and verification, programs are not to be considered as monolithic entities.

Rather a program is built up from parts that are specified, implemented and verified independently. The whole program is then verified on the basis of *specifications* for the parts, that is, without (using) knowledge of the inner structure of the parts.

There are several ways in which one can interpret this. One view is that "built up from parts" refers to the *syntactic phrase structure* of the program. According to this structure a program is either an *atomic action*, a *program variable*, ranging over program meanings, or else it is built up from smaller programs by means of some syntactic operator C . We assume here that each of these operators has a *fixed*, finite arity, that is, combines a fixed number of programs into one larger program.

Examples of atomic statements for the case of TNP are the assignment statement, and the communication commands. The set of syntactic operators includes for instance the sequential and parallel composition operators. Program variables coincide with process identifiers. Within TNP such variables can occur *free* or *bound*. Note that the recursion construct in TNP does *bind* program variables.

Proofs of program specifications should be *compositional*, that is, *syntax directed*. To explain this, assume that, according to this syntax, some program S is composed out of the programs $S_1, \dots, S_n$. That is, S is of the form $C(S_1, \dots, S_n)$ where C is some syntactic constructor. Then according to the statement made above a specification *spec* for S should be proven from specifications $spec_1, \dots, spec_n$ for the parts $S_1, \dots, S_n$. And, most importantly, the proof must *not* refer to the inner syntactic structure of these parts.

In a historical perspective, the first methods for program verification, invented by Floyd and Naur, were not compositional. The reason is that programs were represented as *flowcharts*, and the usual syntactic structure of a flowchart, that has essentially the form of a labeled graph, does not decompose a program into smaller *programs*, but rather into *atomic actions*. Moreover, the number of atomic actions that constitute a flowchart is not bounded from above. Therefore, we do not have a syntactic operator with fixed arity.

A major improvement was made by C.A.R. Hoare who reformulated the Floyd Naur method as a compositional proof system [Ho].

Hoare studied a very tiny language, often referred to as the class of "while programs". The method was extended by several people to encompass most of the usual sequential programming constructs. But when the first proof systems for *parallel* programs were created, these systems did *not* adhere to the syntax directed style.

For instance, a proof rule that violates this constraint is the one for parallel composition proposed by Owicki and Gries in [OG]. To verify a specification for a parallel program $S_1 \parallel S_2$ one must apply a so called *interference freedom test* to *proof outlines* that contain the *program text* of the parts S_1 and S_2 .

Another well known proof system, for the verification of CSP programs, is the system by Apt, Francez and de Roever [AFR]. Here a so called *cooperation test* must be applied, again to proof outlines containing the program texts of the parts. So this system is not compositional either.

One of the first publications containing a proof rule for parallel programs *without* reference to the inner structure of the constituent components was by Misra and Chandy in "Proofs of Hierarchical Networks of Processes" [MC]. It must be noted that Misra and Chandy did not study a language with a clear cut syntax. Rather they used a "picture language" for statically nested parallel processes communicating via channels. Therefore, strictly speaking their system is not compositional in the sense as defined above, because of the rather trivial reason that they have no syntactic operators.

Misra and Chandy's rule is based on an interesting new type of process specifications. Essentially these specifications consist of a certain *assumption* and *commitment* about the *communication history* of a process. It is especially this formulation in terms of communication histories that made it unnecessary to refer to the inner structure of processes.

The Misra and Chandy rule formed the basis for a compositional proof system for the language DNP ("Dynamic Networks of Processes") by Zwiers, de Bruin and de Roever ([ZBR]).

Before we go on discussing several aspects of compositionality in more detail, we would like to contrast this concept with a second interpretation of the statement made at the start of this section.

As one might have noticed already when we mentioned Misra and Chandy's rule, a context free grammar is *not* a presupposition for considering programs as "built up from parts". It suffices that there is some well defined notion of "black boxes" from which a program can be constructed, and which can later on be replaced by implementations in the form of concrete programs. Exactly *how* a program is built up from black boxes can be left open.

A clear advantage of this view is that, since it is more liberal with respect to program structure, many more interesting systems can be discussed in this setting. Typical examples are flowchart languages and transition systems. For instance, although a flowchart does not have a nice syntax described by a context free grammar, one *can* allow named black boxes in it. If only black boxes with a single entry and exit are allowed, then it is possible to *specify* these black boxes in the same way as complete flowcharts are specified, i.e. by means of an entry and an exit condition. Essentially this is the structure of the language FORTRAN, where subroutine bodies play the role of flowcharts and subroutine calls the role of black boxes.

Now our language TNP *does* have a neat context free grammar, so why are we so interested in this black box idea? The reason is that we want to make the distinction between what is called "programming in the small" versus "programming in the large". An often heard explanation of these terms is that programming in the large considers the program structure in terms of subroutines or procedures, whereas

programming in the small refers to the internal structure of these subroutines and procedures. The terms "small" and "large" suggest that the only difference is in the *scale* of the program, and that mathematically speaking there is no real difference at all. We present here a different point of view. Programming "in the small" is done by one person, designing a program and its correctness proof hand in hand. As a consequence, when a specification is designed for a certain program part it is already clear in which program context it must function, and how the specification must "fit" into the correctness proof of the whole. On the other hand, programming "in the large" is not an activity of one person at one time. Therefore specifications for program parts are designed, and proven correct, *without* knowing exactly the context in which the part is to be placed. An (extreme) example is the design of modules for a program library where there is literally no contact between the designer and users of a module. The price for this division of labor is partly paid by the *user* of modules. For he must treat those modules as black boxes for which only an *a priori given* specification is known, and these specifications might *not* suit the correctness proof of the whole. This implies that, in general, a priori given specifications must be *adapted* to a form that *does* fit into the proof. In our opinion, it is this extra step of specification adaptation that makes the difference between correctness proofs "in the small" and "in the large".

Let $spec(S)$ denote that some program S satisfies a certain specification $spec$. Specification adaptation means that a given specification $spec(\zeta)$ for a black box ζ has to be transformed into an alternative specification $spec'(\zeta)$. That is, it must be shown that $spec'(\zeta)$ is a valid formula on the basis of the given formula $spec(\zeta)$. Such adaptations do not exclusively occur within the context of programming "in the large". For instance, for proving the correctness of a *recursive* program, it is necessary to give a correctness proof for the "body" of a recursive construct on the basis of an induction hypothesis that has the form of a specification for recursive calls. In general this given hypothesis must be adapted for each occurrence of a recursive call within the body. Usually the adaptation is performed with the aid of the well known *consequence rule* and various *substitution rules*. An example is given in [Bak], where a (two page long!) proof is given, essentially showing that if ζ satisfies the Hoare style partial correctness formula

$$\{x = z\} \zeta \{x = z\},$$

and we also know, on the basis of syntactic considerations, that the variable z cannot be read or modified by ζ , then it is also true that ζ satisfies:

$$\{x = z - 1\} \zeta \{x = z - 1\}$$

Although specification is needed for programming "in the small", there is a major difference with programming "in the large". In the first case, when a specification is designed it is already known *whether* it has to be adapted, and *for which contexts*. By a proper choice of this specification one can ensure that all adaptations that are necessary can actually be proven correct within the given proof system at hand. We can contrast this with the situation for the second case, where *a priori given* specifications have to be adapted.

Example 4

We consider again the *Bag* process, introduced in the examples above.

On the one hand we have the syntactic phrase structure of the *Bag* process, showing for instance that it is a recursive process, and that the body of this recursive process consists of a choice construct, and so on. On the other hand one sees occurrences of black boxes named *Buf* and *Bag*. In the top down style verification that we gave for the *Bag* process, a certain specification was chosen for the *Buf* process that suited the verification for *Bag*. Thereafter we verified the *Buf* specification. Now consider the following different situation. Assume that actually the *Buf* process did already exist, due to some earlier design activity. That is, it was already constructed, specified and verified by someone else before, *not* with the intention to use it for our *Bag* design. It is unlikely that the specification would have been chosen the same as our *Buf* specification. For instance, it could have been as follows.

- $bcont \stackrel{\text{def}}{=} bag(insert) \oplus bag(up) \ominus bag(getmin) \ominus bag(down)$,
- $bcont[rest(down)/down]$ is $bcont$ with $rest(down)$ substituted for $down$,
- $lastchan \stackrel{\text{def}}{=} chan(last(h|\{insert, getmin, up, down\}))$,
- $lastval \stackrel{\text{def}}{=} val(last(h|\{insert, getmin, up, down\}))$,

Let the buffer invariant be the assertion:

$$\chi_{buf} \stackrel{\text{def}}{=} (lastchan = down \rightarrow lastval = \max(bcont[rest(down)/down])) \wedge \\ (lastchan = getmin \rightarrow lastval = \min(bcont[rest(getmin)/getmin]))$$

Then the alternative *Buf* specification is:

$$\chi_{buf} : \{bcont = [x]\} Buf \{bcont = \emptyset\}.$$

Since we want to treat *Buf* as a black box, *redoing* the verification task to see that *our* *Buf* specification is satisfied (too) is not possible. Rather the already existing specification must be *adapted*, that is, we must prove that the last specification *implies* the one that we used for the verification of the *Bag* process.

However, simply strengthening of the precondition and weakening of the postcondition and the invariant of the alternative specification will not suffice to prove this implication between specifications. For it is *not* the case that the invariant of the alternative specification implies that of the original *Buf* specification. The reason is that the alternative invariant guarantees something about the *last* communication of the history h , at least if it is a communication via *down* or *getmin*. But it does *not* guarantee anything about earlier communications in h . This is to be contrasted with the original *Buf* invariant, that has the form

$$\chi_{bag}(down, up) \rightarrow \chi_{bag}(insert, getmin).$$

The assertion $\chi_{bag}(c_1, c_2)$ has the form:

$$\forall t(t \leq h|\{c_1, c_2\} \cdots),$$

The result of this is that the original invariant makes a certain commitment about *all down* and *getmin* communications occurring in *h*. This indicates why the alternative invariant cannot imply the original one.

One might question whether the alternative specification actually *is* as strong as the original one, since its invariant is logically weaker than the other invariant. The intuitive reason why this is nevertheless the case is that the invariant of an Invariant specification is required to hold, not only for all possible executions, but also *at all moments during such executions*.

□

5 Conclusion

A specification method for the combination of state based and communication based programming can be developed without undue difficulties. There are several alternatives, one of which can be seen as an extension of “classical” Hoare style specification for sequential programs. A yardstick for such specification methods is the question whether *validation* and *verification* of actual programs is feasible. Another yardstick, especially important in the context of “programming in the large”, is whether the method allows specifications to be *adapted*.

Of course one needs formal proof systems to support these styles of development. Such proof systems are given in [Zw], where it is also shown

- that these proof systems are sound,
- that they are complete both from the standpoint of compositionality and modularity, and
- how they interrelate, i.e. how specifications and proofs in one system can be transformed into specifications and proofs in another system.

- [AFR] K.R. Apt, N. Francez, W.P. de Roever,
A proofsystem for Communicating Sequential Processes.
ACM TOPLAS 2 (3) (1980) (359-385).
- [Bak] J. de Bakker,
Mathematical theory of program correctness.
Prentice-Hall, 1980.
- [Ho] C.A.R. Hoare,
An axiomatic basis for computer programming.
CACM 12, 1969.
- [MC] J. Misra, and M. Chandy,
Proofs of Networks of Processes.
IEEE SE 7 (4) 1981.
- [OG] S. Owicki and D. Gries,
An axiomatic proof technique for parallel programs.
Acta Informatica 6 (1976) 319-340.
- [OH] E.-R. Olderog and C.A.R. Hoare,
Specification oriented semantics for communicating processes.
Proc. of the 10th ICALP conference 1983, LNCS 154.
Also: Techn. monograph PRG-37, Progr. Research Group, Oxford Univ. 1984.
- [PJ] P. Pandya, M. Joseph,
P-A Logic - A compositional proof system for distributed programs.
Techn. Report CS-88/1, Tata Institute of Fundamental Research,
Homi Bhabha Road, Bombay 400 005.
- [ZBR] J. Zwiers, A. de Bruin, W.P. de Roever,
A proof system for partial correctness of Dynamic Networks of Processes.
Proc. of the Conference on Logics of Programs 1983, LNCS 164, 1984.
- [ZH] Zhou Chao-Chen, C.A.R. Hoare,
Partial correctness of CSP.
IEEE Int. Conf. on Distributed Computer Systems 1981.
- [Zw] J. Zwiers,
Compositionality, Concurrency and Partial Correctness:
Proof theories for networks of processes, and their connection.
Dissertation, Eindhoven University of Technology.

Available Reports from the Theoretical Computing Science Group

	Author(s)	Title	Classification	
			EUT	DESCARTES
TIR82.1	R. Kuiper, W.P. de Roever	Fairness Assumptions for CSP in a Temporal Logic Framework		
TIR83.1	R. Koymans, J. Vytöpil, W.P. de Roever	Real-Time Programming and Synchronous Message passing (2nd ACM PODC)		
TIR83.2	H. Barringer, R. Kuiper	Towards the Hierarchical, Temporal Logic, Specification of Concurrent Systems		
TIR84.1	R. Gerth, W.P. de Roever	A Proof System for Concurrent Ada Programs (SCP4)		
TIR84.2	R. Gerth	Transition Logic - how to reason about temporal properties in a compositional way (16th ACM FOCS)		
TIR84.3	H. Barringer, R. Kuiper, A. Pnueli	Now you may compose Temporal Logic Specifications (Proc. STOC84)		
TIR84.4	H. Barringer, R. Kuiper	Hierarchical Development of Concurrent Systems in a Temporal Logic Framework		
TIR85.1	W.P. de Roever	The Quest for Compositionality - a survey of assertion-based proof systems for concurrent programs, Part I: Concurrency based on shared variables (IFIP85)		
TIR85.2	O. Grünberg, N. Francez, J. Makowsky, W.P. de Roever	A proof-rule for fair termination of guarded commands (Inf. & Control 1986)		

TIR85.3	F.A. Stomp, W.P. de Roever, R. Gerth	The μ -calculus as an assertion language for fairness arguments (Inf.& Control 1987)		
TIR85.4	R. Koymans, W.P. de Roever	Examples of a Real-Time Temporal Logic Specification (LNCS207)		
TIR85.5	H. Barringer, R. Kuiper, A. Pnueli	A Compositional Temporal Approach to a CSP-like Language		
TIR86.1	R. Koymans	Specifying Message Passing and Real-Time Systems (extended abstract)	CSN86/01	
TIR86.2	J. Hooman, W.P. de Roever	The Quest goes on: A Survey of Proof Systems for Partial Correctness of CSP (LNCS227)	EUT-Report 86-WSK-01	
TIR86.3	R. Gerth, L. Shira	On Proving Communication Closedness of Distributed Layers (LNCS236)	CSN86/07	
TIR86.4	R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, S. Arun Kumar	Compositional Semantics for Real-Time Distributed Computing (Inf.&Control 1987)	CSN86/08	
TIR86.5	C. Huizing, R. Gerth, W.P. de Roever	Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language	CSN86/09	PE.01
TIR86.6	J. Hooman	A Compositional Proof Theory for Real-Time Distributed Message Passing	CSN86/10	TR.4-1-1(1)
TIR86.7	W.P. de Roever	Questions to Robin Milner - A Responder's Commentary (IFIP86)	CSN86/11	
TIR86.8	R. Gerth, A. Boucher	A Timed Failures Model for Extended Communicating Processes; extended abstract (ICALP87)	CSN86/12	TR.4-4(1)
TIR86.9	R. Gerth, W.P. de Roever	Proving Monitors Revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)	CSN86/13	

TIR86.10	R. Koymans	Specifying Message Passing Systems Requires Extending Temporal Logic	CSN86/14	PE.02
TIR86.11	H. Barringer, R. Kuiper, A. Pnueli	A Really Abstract Temporal Logic Semantics for Concurrency (Proc. POPL86)		
TIR87.1	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept	CSN87/01	
TIR87.2	R. Kuiper	Enforcing Nondeterminism via Linear Time Temporal Logic Specifications	CSN87/05	
TIR87.3	R. Koymans	Temporele Logica Specificatie van Message Passing en Real-Time Systemen (in Dutch)	CSN87/06	
TIR87.4	R. Koymans	Specifying Message Passing and Real-Time Systems with Real-Time Temporal Logic	CSN87/07	PE.03
TIR87.5	F. A. Stomp, W.P. de Roever	A correctness proof of a distributed minimum-weight spanning tree algorithm		
TIR87.6	J. Hooman	A Compositional Proof System for an OCCAM-like Real-Time Language	CSN87/14	D4-1-2
TIR87.7	C.Huizing, R. Gerth, W.P. de Roever	A Compositional Semantics for Statecharts	CSN87/15	D4-2-1
TIR87.8	S. Ramesh	A new efficient implementation of CSP with output Guards		
TIR88.1	R. Gerth, M.Codish, Y.Lichtenstein, E. Shapiro	Fully Abstract Denotational Semantics for Concurrent PROLOG	CSN87/21	
TIR88.2	F.A. Stomp, W.P. de Roever, S.Ramesh	A New Decomposition Principle for Verifying Distributed Algorithms, Formalizing their Designer's Intuition		
TIR88.3	J. Zwiers	Compositionality, Concurrency, and Partial Correctness		

TIR88.4	C.Huizing, R. Gerth, W.P. de Roever	Modelling Statecharts Behaviour in a Fully Abstract Way	D4-2-2
TIR88.5	Ron Koymans	Extending Models and Temporal Logics with Inequality and Metric Operations	
TIR88.6	S. Ramesh	Algebraic Specification and Implementation of Infinite Processes	
TIR88.7	Ron Koymans, Ruurd Kuiper, Erik Zijlstra	Paradigms for Real-Time Systems	
TIR88.8	Rob Gerth, Amir Pnueli	Rooting Unity (Fifth Conf. Workshop on Software Specification and Design, Pitts- burgh)	
TIR88.9	Rob Gerth	On the Foundation of Program Refinement (Part I: Safety Properties)	
TIR8.10	C. Huizing, R. Gerth	On the Semantics of Reactive Systems	WP4-2-3
TIR88.11	Rob Gerth, Ruurd Kuiper	Action Refinement	WP4-4-1
TIR88.12	Jozef Hooman	Composition Specification and Verification of Distributed Real-Time Systems	
TIR89.1	Job Zwiers, W.P. de Roever	Predicates are Predicate Transformers: a unified compositional theory for concurrency	
TIR89.2	J.Hooman, S. Ramesh, W.P. de Roever	A Compositional Axiomatization of Sta- techarts: Soundness and Completeness	
TIR89.3	Jozef Hooman, Jennifer Widom	A Temporal-Logic Based Compositional Proof System for Real-Time Message Pass- ing	

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films
85/04	T. Verhoeff H.M.J.L. Schols	Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate
86/01	R. Koymans	Specifying message passing and real-time systems
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specifications of information systems
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several parallel systems
86/05	Jan L.G. Dietz Kees M. van Hee	A framework for the conceptual modeling of discrete dynamic systems
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers
86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987)
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86)
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes

86/13	R. Gerth
W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver
Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben
J.Paredaens	A formal approach to distributed information systems
87/04	T.Verhoeff	Delay-insensitive codes - An overview
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart
L.R.A. Kessener
W.J.M. Lemmens
M.L.P. van Lierop
F.J. Peters
H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeterministic automata.
87/11	P.Lemmens	Eldorado ins and outs.
Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and
A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns, searching for strings.
87/14	J. Hooman	A compositional proof system for an occam-like real-time language

87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas
87/17	K.M. van Hee G.-J. Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples
87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces
87/19	A.J. Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples
88/04	G.J. Houben J. Paredaens D. Tahon	The Nested Relational Algebra: A Tool to handle Structured Information
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling

88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits
88/11	G.J. Houben J.Paredaens	A graphical interface formalism: specifying nested relational databases
88/12	A.E. Eiben	Abstract theory of planning
88/13	A. Bijlsma	A unified approach to sequences, bags, and trees
88/14	H.M.M. ten Eikelder R.H. Mak	Language theory of a lambda-calculus with recursive types
88/15	R. Bos C. Hemerik	An introduction to the category theoretic solution of recursive domain equations
88/16	C.Hemerik J.P.Katoen	Bottom-up tree acceptors
88/17	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable specifications for discrete event systems
88/18	K.M. van Hee P.M.P. Rambags	Discrete event systems: concepts and basic results.
88/19	D.K. Hammer K.M. van Hee	Fasering en documentatie in software engineering.
88/20	K.M. van Hee L. Somers M.Voorhoeve	EXSPECT, the functional part.
89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.