EINDHOVEN UNIVERSITY OF TECHNOLOGY

Combining linear time temporal logic descriptions of concurrent computations

Citation for published version (APA):
Kuiper, R. (1989). *Combining linear time temporal logic descriptions of concurrent computations*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR311554

DOI:
10.6100/IR311554

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.

Download date: 16. Nov. 2023

OMBINING LINEAR TIME TEMPORAL LOGIC DESCRIPTIONS OF CONCURRENT COMPUTATIONS

Ruurd Kuiper

Combining linear time temporal logic descriptions of concurrent computations

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus, prof. ir. M. Tels,
voor een commissie aangewezen door het College van Dekanen
in het openbaar te verdedigen op
vrijdag 26 mei 1989 te 14.00 uur

door

RUURD KUIPER

geboren te Amsterdam

Dit proefschrift is goedgekeurd
door de promotoren

prof. dr. Willem-Paul de Roever
en
prof. dr. Howard Barringer.

ACKNOWLEDGEMENTS

I wish to express my gratitude to the University of Manchester and Eindhoven University of Technology for providing the opportunity and environment to write this thesis.

Most of all I thank Willem-Paul de Roever who was the continuingly inspiring, guiding and helpful force throughout the writing of this thesis and, furthermore, the one who introduced me to temporal logic.

I am also very grateful to my co-authors. I wish to express special gratitude to Howard Barringer and Amir Pnueli. They both generously shared their knowledge with me and were an ever stimulating factor in our joint research. Ron Koymans and Erik Zijlstra I thank for their perseverance with regard to Chapter 1. As my office mate, Ron deserves extra thanks for his help during the final stages of writing.

I thank Cliff Jones for the many suggestions he made and the help he provided during my stay in Manchester. I also wish to acknowledge the stimulus provided by the TEMPLE group in Manchester and Tobias Nipkow. Similar thanks for the period thereafter go to the members of the Sectie Theoretische Informatica at Eindhoven University of Technology.

Many thanks go to Carol Weintraub, Edmé van Thiel and Anita Klooster for typing the various parts of this thesis—often under time pressure.

I am most happy to thank Annette and my parents for their continuous support and patience during the writing of this thesis.

CONTENTS

Introduction 1

Specification Specified, 2
R. Koymans, R. Kuiper, E. Zijlstra.

Now You May Compose Temporal Logic Specifications, 53
H. Barringer, R. Kuiper, A. Pnueli,
Proc. of the 16th ACM Symposium on the Theory of Computing,
pp. 51-63, 1984.

A Compositional Temporal Approach to a CSP-like Language, 83
H. Barringer, R. Kuiper, A. Pnueli,
Formal Models in Programming, North-Holland, pp. 207-227, 1985.

A Really Abstract Concurrent Model and its Temporal Logic, 105
H. Barringer, R. Kuiper, A. Pnueli,
Proc. of the 13th ACM Symposium on Principles of Programming Languages,
pp. 173-183, 1986.

Enforcing Nondeterminism via Linear Time Temporal Logic
Specifications using Hiding, 129
R. Kuiper,
Proc. of the Colloquium on Temporal Logic and Specification,
Altrincham, 1987. To appear in LNCS, Springer, 1989.

Samenvatting 141

Curriculum vitae 145

INTRODUCTION

This thesis consists of five chapters in the form of papers, the first one still to be submitted. The last four consider different but related problems in the area of combining linear time temporal logic descriptions of concurrent computations. During the writing of this series of four papers and while studying other literature published during this period, the underlying questions became more clear to us—prompting the writing of a separate paper. Hence the first paper was written last; the others are presented in chronological order. Although later ones make use of insights obtained in previous ones, the papers are still relatively self contained.

Attempts at formulating the questions showed that some of them were only clear in an intuitive sense—if at all. We therefore made an attempt at clarifying and, in part, formalising the notions involved. Sometimes this proved to be trivial and sometimes surprisingly difficult.

The first paper records our investigations in that direction thus far. We feel that we have reached a point where most of the aims and questions that concern the other papers are reasonably clear. We are well aware though, that from a broader point of view a lot is still left to be done.

In the first paper we start by presenting an overview of notions about formalisms to describe computations. After having done so we introduce the four papers and discuss the relationships between them and also with respect to some other research in the same area.

The notions themselves are evaluated in a formal manner. We feel that thereafter their meaning should be sufficiently clear to discuss the papers in a more loose way. This seems to us more useful as an introduction to the papers than a formal treatment. Such a formal treatment might, in fact, only be appreciated after reading the papers. Also, the purpose of the formalisation was clarification rather than advocating formality for its own sake.

Thus, the real introduction to this thesis is the first paper.

SPECIFICATION SPECIFIED

Ron Koymans, Ruurd Kuiper,

Department of Computing Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven,
The Netherlands
Email: wsdcronkheitue5.bitnet or mcvax!eutrc3!wsinronk.uucp,
mcvax!eutrc3!wsinruur.uucp.

Erik Zijlstra,

Foxboro Nederland NV,
Koningsweg 30, 3762 EC Soest,
The Netherlands

ABSTRACT

Driven by the problem of how to motivate and compare several temporal logic based specification formalisms, [BKP84, 85, 86], [Ku87] and [La83a, 85], [AL88], we evaluate notions that describe properties of such formalisms in a systematic manner. It turns out, that some properties are difficult to formalise, like the intuitive feasibility of the representation of executions—in such cases we at least attempt to give some structure to their evaluation. Of the more readily formalisable ones in particular compositionality and modularity are discussed. Their domain of definition is extended to show the link between their use in different domains. Reactive systems and bias freedom are also addressed.

1. INTRODUCTION

Many different formalisms to describe systems and their behaviour are available nowadays, varying greatly in their expressive power as well as in the manner in which information is represented. There is a corresponding diversity in the ways in which such formalisms are evaluated. This is unfortunate, as it makes it difficult to compare the merits of different formalisms.

It even appears from the literature that particular evaluations are often given in terms specific to the formalism under consideration. An example is compositionality, a notion that after its introduction in the area of semantics went through a long history of more or less intuitive or limited definitions in the area of specifications (cf. [La83a,

---

The authors are currently working in and partially supported by ESPRIT project P937: "Debugging and Specification of Ada Real-Time Embedded Systems (DESCARTES)". The second author also acknowledges support under the Alvey/SERC grant GR/D/57492.

ZBR83, BKP84, La85, dR85, Sta85, HR86, NDGO86]) before regaining respectability in Zwiers's thesis [Zw88].

In this paper we take a step back from concrete formalisms and reconsider the aims and properties of such formalisms from a more abstract point of view. It appears, that properties of formalisms can be divided in two types, one more intuitive, the other more formal. We therefore address two questions, of rather different nature.

Firstly, how can it be assessed how well a formalism represents reality? We interpret this question in a broad sense. For example, not only the precision but also the ease by which a representation can be given matters. This is an intuitive, not readily formalisable judgement. However, the relationship to reality can, in our opinion, be assessed mote systematically and explicitly than is usually done.

The idea is to split the assessment of the representation of reality into the assessment of the subrepresentations of a collection of aspects we consider important. These aspects can be identified in a both more explicit as well as more abstract way than by choosing a particular model of representation. Examples are the representation of observation or of execution. The intuitive correctness of subrepresentations can usually be argued more clearly and, more importantly, enables better comparisons between different formalisms than when the representation as a whole is considered. An example is the assessment of the different ways stuttering is treated, like for instance in [La85, BKP86]. This is the content of Section 2.

Secondly, how can a collection of formal notions that describe properties of formalisms be obtained and defined in a general manner? The approach we take is to structure the generation of this collection. As will be explained in greater detail later, this roughly means identifying a few basic concepts and instantiating and refining these in different areas and at different levels of abstraction. This approach led to some new insight into expressiveness but, more importantly, to a better understanding of compositionality and modularity. This is carried out in Section 3.

It turns out that, as a bonus, the approach guides us in defining the concepts of reactive system, introduced in [HP85], and bias freedom, as used in [Jo86], in some more detail and generality than is usually done. As results from Section 2 as well as 3 are used, this is described in a separate section, 4.

The investigation of the two questions mentioned originated from attempts to understand better the motivations for the design of the temporal logic based formalisms presented in [BKP84, 85, 86] and [Ku87]. In Section 5 we therefore evaluate and compare these formalisms. The relation to work by Lamport ([La83a, 83b, 85, 89], [AL88]) is investigated. Some remarks are made about the approach by Nguyen et al [NDGO86] and a proof method by Stark ([Sta85]).

Although the paper is quite formal up to Section 5, we feel that our aim was clarification and not formalisation for its own sake. Section 5 is foremost intended as an introduction to [BKP84, 85, 86] and [Ku87]. Hence, we use the insights obtained by the formal approach in a much less formal manner there.

Each section has its own introduction where further division into subsections is provided.

2. JUSTIFICATION OF FORMALISATION WITH RESPECT TO THE INTUITION

Perhaps the most basic question concerning the justification of a formalism is how the description is related to reality. Hence the, rather elusive, aim in this section is to justify formal objects against intuitions.

The aim is not so much to find "the best" representation, but rather to better understand where and why formalisms differ.

By formalism we mean a symbolism that describes certain aspects of objects. In its full generality, this only requires some relation between objects and such aspects. It is tempting to choose a function as this relation. However, considering a specification formalism we see that such a choice is too narrow. One object can satisfy, i.e. be related to, several specifications and vice versa.

A concrete specification structure models certain aspects of reality in a specific manner. What these aspects are and how well specifications describe them depends on the intuitive structure of the objects, the formal structure of the specifications and the formal satisfaction relation between objects and specifications. The choice of these aspects in itself is a decision that belongs to a higher level than the actual modelling.

The idea now is simply to provide a list of intuitive aspects that are, at least in our opinion, relevant. To justify a particular modelling one then needs to justify the modelling of these aspects.

For example, one could model behaviour, especially concerning external choices, in different ways. One possibility is, to use a tree of states giving values to observable variables. Alternatively, one might choose the representation by sets of sequences of states together with failure sets ([Ho85]). As this will generally result in a different level of abstraction of modelling choices, the satisfaction relations between objects and specifications will also differ.

Such choices should be justified. In the example, it is already clear that useful aspects to consider might be observations and executions.

We structure this evaluation a little. As the most general formalism in this paper we use a specification structure. The word "structure" rather than "language" is chosen to emphasize that we take an abstract view, not, for example, focussing on features of the language in which specifications are written. Moreover, the justification process we describe is independent of the languages used to denote objects or specifications.

Definition 2.1.
An object structure $\boldsymbol{O}$ is a set of objects.
$O, O_1, O_2, \ldots$ are variables ranging over $\boldsymbol{O}$.

Definition 2.2.
A specification structure $< \boldsymbol{S}, \text{SAT} >$ for an object structure $\boldsymbol{O}$ consists of

(i) a set $\boldsymbol{S}$ of specifications together with
(ii) a relation $\text{SAT} \subseteq \boldsymbol{O} \times \boldsymbol{S}$.

$S, S_1, S_2, \ldots$ are variables ranging over $\boldsymbol{S}$.
$O \text{ SAT } S$ intuitively means that S is an aspect of object O.

We justify a formalism via splitting the problem of justification into several subproblems. This seemingly obvious approach will, in Section 5, be seen to allow some insights into the relative merits of different models that are hard to discuss directly. It will be carried out for program specification formalisms and hence also for program semantics—the latter being a special case of the former.

The split we suggest is the following.

Definition 2.3.
The intuitive justification of a specification structure $< \boldsymbol{S}, \text{SAT} >$ for an object structure $\boldsymbol{O}$ requires:

I.(a) A formalisation of (momentary) observation, $s \in \boldsymbol{s}$;
I.(b) a formalisation of execution, $\sigma \in \boldsymbol{\sigma}$;
I.(c) a formalisation of specification, $S \in \boldsymbol{S}$;

II. a formalisation of the relation SAT between object and specification;

formal operators

III.(a) $P_{\boldsymbol{\sigma}} : \boldsymbol{S} \rightarrow \mathcal{P}(\boldsymbol{\sigma})$ and
III.(b) $P_{\boldsymbol{S}} : \boldsymbol{\sigma} \rightarrow \mathcal{P}(\boldsymbol{s})$

that extract, respectively, a set of executions from a specification and a set of observations from an execution.

The justification consists of justifying the above six aspects of the formalisation.

Remarks.

1. The formalisation of specification is of course already given with the specification structure. We list it again, to emphasize that it is one of the choices to be justified, but not the only one. Its justification should now be limited to the aspects not addressed separately already.

   For example, in the comparison of a tree based model with a failure set model it should be clear that, abstractness considerations left aside, the former is better than the latter on all points listed—for instance, choice is modelled via structure rather than by extending observations to include a representation of all failed attempts at communication. The failure set model, however, is a lot easier to manipulate mathematically than the tree based one.

2. The justification, although comparing intuition and formalisation, is not limited in its arguments to this area. For instance, ease of mathematical manipulation of the specifications may be an important consideration.

   Also, although the justification is with respect to the intuition, this does not mean that in this justification no formal properties may be used. For example, the fact

that a formalisation is fully abstract, a notion to be discussed in Section 3, may be an argument in the justification of I.(c).

Later, in Subsection 3.3 and even more so in Section 4, we shall discuss object structures that have more structure, reflecting syntactic structure of programming languages. This leads to further requirements that one might want to impose on a specification structure that describes these objects, like compositionality or modularity. The extra structure can be used to give extra arguments in the justification. However, the link between the formalism and reality can, in our opinion, still be justified as before.

There are no a priori requirements about the actual modelling of the aspects listed in the definition as applied to specification structures. A semantics for programs can be viewed as a special case of a specification structure. The extra requirement—and this is an a priori requirement—is then that the SAT relation is functional. To keep the terminology consistent, a semantics is called a behaviour structure.

Definition 2.4.
A behaviour structure $< \boldsymbol{B}, \text{BEH} >$ for an object structure $\boldsymbol{O}$ consists of

(i) a set $\boldsymbol{B}$ of behaviours together with
(ii) a functional relation $\text{BEH} \subseteq \boldsymbol{O} \times \boldsymbol{B}$.

$B, B_1, B_2, ...$ are variables ranging over $\boldsymbol{B}$.

Proposition 2.1.
A behaviour structure is a specification structure for which the SAT relation is functional.

Remark.
As BEH is a functional relation, $O \,\text{BEH}\, B$ could be denoted as $\text{BEH}(O) = B$, or even using semantics brackets instead of BEH. However, the comparison with specification structures would then be less clear. As it is just this comparison that is one of our concerns, we use the more general notation while leaving it to the reader to perform the, straightforward, translation where necessary to help the understanding.

Proposition 2.2.
Viewed as a function, BEH induces an equivalence relation on $\boldsymbol{O}$ via $O_1 \sim_B O_2$ iff $\text{BEH}(O_1) = \text{BEH}(O_2)$.

It might be argued that one could limit justification with respect to the intuition to behaviour structures for programming languages. A specification structure should then be given in terms of the already justified behaviour structure rendering further justification superfluous. As we wish to allow specifications to describe behaviour directly rather than only as generated by programs, we considered the general case of specification structures. This is, for example, necessary to give a semantics to mixed specification and program terms.

Throughout the paper, standard names for variables, constructors, etc. are used to

indicate their domains; where no confusion is likely to arise, explicit mention of these domains is mostly omitted. Also, when $\boldsymbol{S}$ and $\boldsymbol{S}'$, or $\boldsymbol{B}$ and $\boldsymbol{B}'$, are used, their obvious connection with SAT and SAT$'$, respectively BEH and BEH$'$, is often left implicit. Similar holds for other information that is obvious from the context, for example the fact that all specification and behaviour structures apply to the object structure $\boldsymbol{O}$.

Example.

1. Consider a behaviour structure that takes classes of intuitively equivalent programs as its behaviours. This is a very general model; every semantics can be represented in this way. However, this is also a very non- intuitive way of representing behaviour: what would an observation or execution be in this representation? Indeed, points I.(a), (b), II and III can hardly be argued satisfactorily. The justification has to be given completely at point I.(c), where it has to be argued directly that the division in equivalence classes corresponds to the intuition.

2. Equally unsatisfactory as in Example 1 is, in the case of specification structures, taking programs themselves as specifications. The problem now is the justification of II, as SAT has to be defined in such a manner, that intuitively equivalent programs satisfy the same specifications.

3. Consider the readiness behaviour structure for, for instance, communicating sequential processes with synchronised communication, CSP-programs [Ho85]. The behaviour of a program is the set of all prefixes of sequences of successful communications with a readiness set attached. A readiness set is a set of communications all of which the program can engage in at that point of the sequence. Note that a prefix plus readiness set is included if it can occur rather than always does occur: if identical prefixes can lead to different readiness sets, both will be included in the behaviour.

   Difficulties arise when defining executions. If just complete communication sequences are taken as executions, the relation to behaviour, as mentioned in III.(a), is rather unsatisfactorily, as the readiness sets are not explained. Some may be content to regard complete sequences of communications plus all intermediate readiness sets as executions. However, the readiness sets are then still hard to reconcile with an intuitive notion of observation, the requirement in III.(b).

4. Similar problems arise when failure sets [Ho85] or saturated readiness sets [He88] are used: the extra abstractness obtained can be used to justify the extensions of the sets involved, but not their presence as such.

3. FORMAL PROPERTIES OF DESCRIPTIVE SYSTEMS

The aim in this section is to arrive at a reasonably complete collection of formal notions concerning expressiveness and the way component descriptions are combined as well as to provide definition for these in a general manner. We restrict ourselves to notions that admit straightforward formal definitions. An example of a, quite useful, property that falls outside our scope is:

"Similar systems have similar specifications."

In Subsection 3.1 the approach to arrive at the desired collection is outlined. In Subsections 3.2 and 3.3 the notions are introduced. In Subsection 3.4 this rather theoretical approach is, briefly, linked to existing formalisms.

It appears that all notions discussed in this section can be defined at a level of abstractness that makes the definitions independent of the choice of both basic behaviours and SAT.

Taking this systematic approach, we unavoidably run into quite some trivial and well-known notions and relationships between these. As we also encounter quite complicated ones eventually and as it is difficult to decide where to start other than at the beginning, we have chosen to include these. They should perhaps be viewed as an introduction to the approach rather than for their own merit. Also, we have chosen to gently introduce results via many evident propositions rather than proving a few more difficult ones.

3.1. The approach

To structure the inventarisation of the rather overwhelming variety of notions we employ two tactics.

The first tactic

This tactic derives from the observation that in the literature one often encounters notions that have the same or similar names while occurring in different areas. For example, compositionality is used for semantics, specifications and proof systems. This suggest the existence of more general concepts underlying such notions. In the sequel we distinguish between the use of the terms notion and concept. A notion is a formally defined instantiation in some area of a, usually more intuitive, concept.

This provides the first tactic, namely to consider two features of notions:

(i) the concept that the notion instantiates;
(ii) the area to which the notion applies.

As one would expect, not all concepts can be meaningfully instantiated in all areas. Furthermore, some concepts give rise to several different notions occurring in the same area.

An example is the concept of abstractness. This concept occurs in the area of behaviour structures as well as the area of specification structures but not, at least not directly,

in connection with proof systems. Also, in the area of specifications several different notions of abstractness will be seen to apply.

Evaluating the combinations of entries along axes (i) and (ii), mentioned above, and also comparing the further differentiations that are sometimes made for these combinations (as will be seen, again abstractness is an example) aids agains overlooking useful notions. Conversely, the obligation to consciously choose and motivate all entries along the axes somewhat helps to introduce only necessary ones.

The entries along the axes are:

Concepts

Abstractness, expressiveness, compositionality, modularity, soundness, completeness, bias freedom, reactivity.

Areas

Comparison of
- two specifications,
- a specification and a specification structure,
- two specification structures,

- similar for behaviours;

assessment of
- a specification structure,
- a behaviour structure,
- a proof system,
- a specification language.

It may seem surprising that the concepts expressiveness and abstractness are both included explicitly, as they occur sometimes as dual notions. However, it appears that this duality is not always direct and also that in some cases only one of the two is relevant. Therefore we always investigate both notions and wish to be able to choose either one as primary when introducing them.

The second tactic

This tactic is based on the observation that there exists a link between the level of abstraction at which a descriptive framework is given and the possibility to define notions pertaining to that formalism. For example, the level of abstraction at which the object structure is defined in Section 2 does not enable discussing compositionality.

Contrasting examples, treated in this paper, are abstractness and bias freedom. The former can be discussed in a quite unstructured setting, whereas the latter requires a formalism to possess notions of internal (state) configuration as well as observation.

We exploit this fact to structure the introduction of the notions and, more importantly, to clarify for each notion what features of formalisms are parameters of it. In order to maintain comparability of notions used in the evaluation of different formalisms such parameterisation should be minimalised and where unavoidable be made explicit.

For some structures in formalisms, for example specification structures, we give different definitions at different levels of abstraction. For example, structure may or may not be considered. This may be justified by the observation that the difference consist only in adding extra structure where required for the definition of more intricate notions. Furthermore, definitions are upward compatible in the sense that if a structure satisfies a detailed definition at a lower level, we shall take care that it also satisfies the more abstract version at a higher level.

In our opinion, it is better to simply omit structure at higher levels of abstraction rather than not make use of it. This is especially so when one is interested in the question whether notions that are defined for different formalisms are comparable. The definitions should then be given in terms that use as little as possible of the structure of the formalisms. Compositionality is a case in point of a notion that has been defined in many different ways in terms specific to particular formalisms, although a more abstract definition is perfectlfy well possible. Of course, a reader troubled by this approach may substitute the most detailed definition of a structure for all occurrences.

As a running example a very simple set theory based model is used to exemplify the various notions on. This model will be developed through different versions to show the relationship between definability of notions and the amount of structure present in the model. The purpose of the example is to clarify the meaning of the notions and to show how little structure is necessary to define them, not to give an intuitively justifiable representation of reality in the sense of Section 2.

3.2. Level 1—Unstructured objects

We start out by introducing the most abstract framework: Level 1. Then the notions that can be formulated in this framework are defined. Examples belonging to this level of abstractness are provided throughout the exposition.

3.2.1. The framework

In order to discuss formalisms, an indispensable prerequisite is to have something to describe. For this we use the object structure as defined in Section 2. This represents the most abstract level—no structure is assumed of either the objects or the domain they belong to.

Intuitively, objects can be thought of as black boxes, off-the-shelf components, programs, etc.—even behaviours would qualify as objects if so desired. This last possibility covers the case of a general specification structure where the intent is to directly describe behaviours.

Firstly, we take the specification view and start from the observation that objects have properties. There is, of course, no reason why some objects should not possess several or, alternatively, no properties. This leads to also adopting at this level the definition of specification structure as provided in Section 2.

Running example

Given as object structure some set of objects $\boldsymbol{O}$, a property can be represented as just

the subset S of $\boldsymbol{O}$ of all objects that possess that property. A specification structure then consists of

(i) a set $\boldsymbol{S}$ of subsets of $\boldsymbol{O}$ together with
(ii) the satisfaction relation "is element of", $\in$, $\subseteq \boldsymbol{O} \times \boldsymbol{S}$.

In the sequel, examples refer to the running example unless otherwise stated.

The running example can not only be viewed as a way to exemplify notions but also as a standard way to give a uniform representation of different structures, thus enabling comparison. Any specification can be represented by the subset of objects that satisfy it; the satisfaction relation is then represented by $\in$.

Secondly, rather than arbitrary properties, objects have behaviour. Again, at this level the definition of object structure from Section 2 is the appropriate one.

To avoid confusion, we mention the following rather subtle distinction. A specification structure relates, in a general way, properties to objects. These objects might be chosen to be behaviours, in which case properties of behaviours are specified. Alternatively, as for behaviour structures, one might choose behaviours as special properties. A behaviour structure thus relates behaviour to objects; it does not relate properties to behaviours.

Running example.
The special property "behaviour" of an object O can again be represented as just the subset B of $\boldsymbol{O}$ of all objects that have that behaviour. A behaviour structure then consists of

(i) a partitioning $\boldsymbol{B}$ of $\boldsymbol{O}$ into subsets together with
(ii) the relation $\in$ on $\boldsymbol{O} \times \boldsymbol{B}$.

The fact that $\boldsymbol{B}$ is a partitioning ensures that $\in$ is a functional relation on $\boldsymbol{O} \times \boldsymbol{B}$.

3.2.2. The notions

In the present, unstructured, framework the following entries along the axes can be discussed.

Concepts

Abstractness, expressiveness, soundness, completeness.

Areas

Comparison of
two specifications,
a specification and a specification structure,
two specification structures,

similar for behaviours;

assessment of
a proof system.

When trying to find intuitively appealing notions we have always had the set theoretic setting of the running example in mind. We urge the reader to use the examples to judge the validity of our hopes that the notions are sensibly defined.

3.2.2.1. *Abstractness and expressiveness*

In this subsection, we go from the general to the particular in that we first give notions concerning specification structures and then apply these notions to the progressively more restricted comparisons of a specification and a behaviour structure respectively two behaviour structures. Some of the notions will be seen to coincide in the last case.

Specification structures

We start the investigation of comparisons by comparing single specifications, possibly from different specification structures. Intuitively, one specification is more abstract than another if it is in some sense less precise. At the present level, where objects have no structure and specifications just characterise sets of objects via SAT, this can only mean that more objects satisfy that specification than satisfy the other one. Using the same intuition to define expressiveness renders a notion that is the direct dual of abstractness.

Definition 3.1.
For specification structures $< \boldsymbol{S}, \text{SAT} >$ and $< \boldsymbol{S}', \text{SAT}' >$ for $\boldsymbol{O}$,

a specification $S \in \boldsymbol{S}$ is abstract with respect to a specification $S' \in \boldsymbol{S}'$ if
for all objects $O \in \boldsymbol{O}$, if $O\,\text{SAT}'\,S'$ then $O\,\text{SAT}\,S$.

$S \in \boldsymbol{S}$ is expressive with respect to $S' \in \boldsymbol{S}'$ if
for all $O \in \boldsymbol{O}$, if $O\,\text{SAT}\,S$ then $O\,\text{SAT}'\,S'$.

Running example.
$S \in \boldsymbol{S}$ is abstract with respect to $S' \in \boldsymbol{S}'$ iff $S' \subseteq S$.

$S \in \boldsymbol{S}$ is expressive with respect to $S' \in \boldsymbol{S}'$ iff $S \subseteq S'$.

Proposition 3.1.
S is expressive with respect to S' iff
S' is abstract with respect to S.

Definition 3.2.
S from $< \boldsymbol{S}, \text{SAT} >$ is equivalent to S' from $< \boldsymbol{S}', \text{SAT}' >$ with respect to $\boldsymbol{O}$, $S \sim_{\boldsymbol{O}} S'$, if
for all $O \in \boldsymbol{O}$, $O\,\text{SAT}\,S$ iff $O\,\text{SAT}'\,S'$.

This can be viewed as stating that S and S' do not differ on $\boldsymbol{O}$.

Note, that because of the dependency of $\sim_{\boldsymbol{O}}$ on the SAT and SAT$'$ relations, if $S = S'$ it does not necessarily hold that $S \sim_{\boldsymbol{O}} S'$ (or the other way around).

Another, similar, notion, although perhaps not quite belonging to the area of comparisons of single specifications, is that two objects are equivalent if they are not different as far as a specification structure can distinguish.

Definition 3.3.
O_1 is equivalent to O_2 with respect to $< \boldsymbol{S}, \text{SAT} >$, $O_1 \sim_{\boldsymbol{S}} O_2$, if
for all $S \in \boldsymbol{S}$, (O_1 SATS iff O_2 SAT S).

Running example.
$S_1 \sim_{\boldsymbol{O}} S_2$ iff $S_1 = S_2$.
$O_1 \sim_{\boldsymbol{S}} O_2$ iff for all $S \in \boldsymbol{S}$, $\{O_1, O_2\} \subseteq S$ or $\{O_1, O_2\} \cap S = \emptyset$.

Proposition 3.2.
$S \sim_{\boldsymbol{O}} S'$ iff S is both expressive and abstract with respect to S'.

Remarks.

1. The reduction of equivalence to equality as in the case of the canonical example does not hold in general. The reason is that the SAT relation may blurr the difference between some specifications.

2. In itself, using "$\in$" as satisfaction relation does not guarantee either that equivalence implies equality. The reason it does in the case of the running example is that the sets used as specifications only have objects as elements.

   Let $< \boldsymbol{S}, \text{SAT} >$ again be as in the canonical example. Now let the relation SAT remain the same, i.e. $\in$, but extend the set of specifications $\boldsymbol{S}$ to $\boldsymbol{S}'$ as follows. Let $S \in \boldsymbol{S}$. Let E be a new element, not an object, i.e., $E \notin \boldsymbol{O}$. Extend the set $\boldsymbol{S}$ of specifications with $S_E = S \cup \{E\}$ to $\boldsymbol{S}'$. Then for $< \boldsymbol{S}', \text{SAT} >$, $S \sim S_E$ but not $S = S_E$ any more.

   Note, that strictly speaking it is still the satisfaction relation that blurrs the difference between S and S_E—but not so much because of the nature of the relation but because of the lack of objects to distinguish between specifications, i.e., because of a property of the domain of the relation. The next proposition records this observation.

Proposition 3.3.
($S \sim_{\boldsymbol{O}} S'$ implies $S = S'$) iff
(if $S \neq S'$ then there is an $O \in \boldsymbol{O}$ such that
either O SAT S but not O SAT$'$ S' or O SAT$'$ S' but not O SAT S).

The next step is to compare different specification structures. It appears that expressiveness and abstractness are far less clear concepts in this area; consequently, there exists quite a number of different notions. The differences stem mainly from different choices as to what should be compared: individual specifications or specification structures as a whole.

Rather than attempting to collect the many different notions that appear in the literature, we try to clarify the situation by reiterating the purpose of a specification

structure: characterising properties, i.e. groups of objects. It is therefore the ability with which this can be done that determines the expressive power of a specification structure.

To emphasize the care needed to avoid choosing undesirable notions we consider two choices that, in our opinion, seem reasonable but are not.

1. The most straightforward approach would seem to use the comparisons between single specifications that have just been given.

   $< \boldsymbol{S}, \mathrm{SAT} >$ is abstract with respect to $< \boldsymbol{S}', \mathrm{SAT}' >$ if
   for all $O \in \boldsymbol{O}$, $S' \in \boldsymbol{S}'$ there is $S \in \boldsymbol{S}$ such that
   $O \,\mathrm{SAT}\, S$ and S is abstract with respect to S'.

   This requirement, however, would be fulfilled as soon as $\boldsymbol{S}$ contains a specification that every object satisfies and therefore hardly imposes any restriction. Moreover, even if that possibility were avoided, the running example shows that any set of supersets would suffice.

2. Another useful option, most easily formulated for expressiveness, would seem to be to require that a formalism is expressive with respect to another one if it enables separating at least as many objects.

   $< \boldsymbol{S}, \mathrm{SAT} >$ is expressive with respect to $< \boldsymbol{S}', \mathrm{SAT}' >$ if
   for all $O_1, O_2 \in \boldsymbol{O}$,
   if $O_1 \sim_{\boldsymbol{S}} O_2$
   then $O_1 \sim_{\boldsymbol{S}'} O_2$.

   This requirement, however, is fulfilled in the setting of the running example if $\boldsymbol{S}'$ is taken to consist of all subsets of $\boldsymbol{O}$ and $\boldsymbol{S}$ to consist of only the subsets of form $\boldsymbol{O}\backslash\{O\}$, for all $O \in \boldsymbol{O}$. Intuitively however, $\boldsymbol{S}$ does not seem as expressive as $\boldsymbol{S}'$ at all.

We discuss two, in our opinion quite natural, alternative options.

Option 1.

We start with an intermediate step. We assess the expressive power of one specification with respect to a specification structure.

One intuition is that a specification is abstract with respect to a specification structure if it describes a property that abstracts away from some feature that the structure distinguishes. A perhaps intuitively easier way to view this is to assume that possessing a property is equivalent to possessing several subproperties. Abstracting then means that some subproperty is not taken into account any more, i.e. property is not used any more to distinguish between objects.

Thus, for this first option it is still the precision of a property, be it with respect to a collection of other properties, that is taken as relevant.

In the following we use I as an arbitrary index set.

Definition 3.4.
S from $< \mathbf{S}, \mathrm{SAT} >$ is *abstract* with respect to $< \mathbf{S}', \mathrm{SAT}' >$ if
there are $S_i' \in \mathbf{S}'$, $i \in I$, such that
for all $O \in \mathbf{O}$, $O \; \mathrm{SAT} \; S$ iff there is $i \in I$ such that $O \; \mathrm{SAT}' \; S_i'$.

Running example.
S is abstract with respect to $< \mathbf{S}', \mathrm{SAT}' >$ if
there are $S_i' \in \mathbf{S}'$, $i \in I$, such that $S \; = \; \bigcup_{i \in I} S_i'$.

Clearly, there is no dual notion of expressiveness. However, using the same intuition it is obvious how one can define abstractness, and expressiveness too, for one specification structure with respect to another.

Definition 3.5.
$< \mathbf{S}, \mathrm{SAT} >$ is *abstract* with respect to $< \mathbf{S}', \mathrm{SAT}' >$ iff
all $S \in \mathbf{S}$ are abstract with respect to $< \mathbf{S}', \mathrm{SAT}' >$.

$< \mathbf{S}, \mathrm{SAT} >$ is *expressive* with respect to $< \mathbf{S}', \mathrm{SAT}' >$ iff
for all $S \in \mathbf{S}$ there are $S_i \in \mathbf{S}$, $i \in I$ and $S' \in \mathbf{S}'$ such that
for all $O \in \mathbf{O}$, $O \; \mathrm{SAT}' \; S'$ iff $O \; \mathrm{SAT} \; S$ or there is $i \in I$ such that $O \; \mathrm{SAT} \; S_i$.

Running example.
$< \mathbf{S}, \mathrm{SAT} >$ is abstract with respect to $< \mathbf{S}', \mathrm{SAT}' >$ if for all $S \in \mathbf{S}$,
there are $S_i' \in \mathbf{S}'$, $i \in I$, such that $S \; = \; \bigcup_{i \in I} S_i'$.

$< \mathbf{S}, \mathrm{SAT} >$ is expressive with respect to $< \mathbf{S}', \mathrm{SAT}' >$ if for all $S \in \mathbf{S}$,
there are $S_i \in \mathbf{S}$, $i \in I$ and $S' \in \mathbf{S}'$ such that $S' \; = \; \bigcup_{i \in I} S_i \; \cup \; \{S\}$.

Remark.
Following the intuition, in order to be more expressive than $\mathbf{S}'$ $\mathbf{S}$ should allow the description of enough, possibly overlapping, subproperties to subdivide every property that $\mathbf{S}'$ can describe. However, for $\mathbf{S}$ to be more abstract than $\mathbf{S}'$ it is only required that every property that, again, $\mathbf{S}$ can describe can be subdivided into subproperties describable in $\mathbf{S}'$. It should not be misread as the ability of $\mathbf{S}$ to abstract every collection of subproperties that are describable in $\mathbf{S}'$ to a single property. This would in general be a requirement that is too strong to fulfill.

Proposition 3.4.
If $< \mathbf{S}, \mathrm{SAT} >$ is expressive with respect to $< \mathbf{S}', \mathrm{SAT}' >$ then
$< \mathbf{S}', \mathrm{SAT}' >$ is abstract with respect to $< \mathbf{S}, \mathrm{SAT} >$.

The converse does not hold.

Remark.
One might ask whether stronger requirements should be posed for abstractness or expressiveness. In our opinion this is not the case. The motivation is that the distinctions that specifications make between objects are less precise than the concrete properties one might decide to ignore.

A simple example is that for certain limited domains $\boldsymbol{O}$ the properties "is prime" and "is odd" might coincide. If one decides not to be interested any more in the property prime, this does not mean that one also wishes to loose the distinction between even and odd.

Some more intricate questions of the same nature are answered via an example.

> Consider the specification structure $\boldsymbol{O} = \{1,2,3,4\}$,
> $S_1 = \{1\}$, $S_2 = \{2\}$, $S_3 = \{3\}$, $S_4 = \{4\}$,
> $S_{13} = \{1,3\}$, $S_{24} = \{2,4\}$, $S_{23} = \{2,3\}$.

a. Should properties that are taken together be disjoint?
   No, consider S_{13} and S_{23}. One might decide to ignore differences in terms of "property" 3.

b. Would it be sensible to require that if the difference between S_1, S_2, S_3 and S_4 is ignored then the same should hold fro S_{13} and S_{24}?
   No, because the properties that the division in subsets reflects might be quite independent:
   $S_1 - S_4$ might reflect properties about ordering. O_3 for example might reflect being greater then 2 and smaller than 4,
   S_{13}, S_{24} being odd respectively even.

c. A similar argument forbids one to ignore the difference between, for example, S_1 and S_2 if the difference between S_{13} and S_{24} is ignored.

All the examples depend on the fact that intuitive subproperties can be chosen that cannot be distinguished when only the division of $\boldsymbol{O}$ into subsets of objects that they induce is considered. This is in general the case for specification structures.

Option 2.

Another possibility to define abstractness (expressiveness) is to require that a specification structure can characterise at most (as least) as many properties as some other structure, i.e. the number of different properties is taken as relevant here.

Definition 3.6.
$< \boldsymbol{S}, \text{SAT} >$ is weaker than $< \boldsymbol{S}', \text{SAT}' >$ if
for all $S \in \boldsymbol{S}$ there is $S' \in \boldsymbol{S}'$ such that $S \sim_{\boldsymbol{O}} S'$.

$< \boldsymbol{S}, \text{SAT} >$ is stronger then $< \boldsymbol{S}', \text{SAT}' >$ if
for all $S' \in \boldsymbol{S}'$ there is $S \in \boldsymbol{S}$ such that $S \sim_{\boldsymbol{O}} S'$.

Running example.
$< \boldsymbol{S}, \text{SAT} >$ is weaker than $< \boldsymbol{S}', \text{SAT}' >$ iff $\boldsymbol{S} \subseteq \boldsymbol{S}'$.

$< \boldsymbol{S}, \text{SAT} >$ is stronger than $< \boldsymbol{S}', \text{SAT}' >$ iff $\boldsymbol{S}' \subseteq \boldsymbol{S}$.

Proposition 3.5.
$< \boldsymbol{S}, \text{SAT} >$ is stronger than $< \boldsymbol{S}', \text{SAT}' >$ iff
$< \boldsymbol{S}', \text{SAT}' >$ is weaker than $< \boldsymbol{S}, \text{SAT} >$.

Definition 3.7.
$< \boldsymbol{S}, \text{SAT} >$ is equivalent to $< \boldsymbol{S}', \text{SAT}' >$, $< \boldsymbol{S}, \text{SAT} > \sim_{\boldsymbol{O}} < \boldsymbol{S}', \text{SAT}' >$, if for all $S \in \boldsymbol{S}$ there is $S' \in \boldsymbol{S}'$ such that $S \sim_{\boldsymbol{O}} S'$ and vice versa.

Running example.
$< \boldsymbol{S}, \text{SAT} > \sim_{\boldsymbol{O}} < \boldsymbol{S}', \text{SAT}' >$ iff $\boldsymbol{S} = \boldsymbol{S}'$.

Proposition 3.6.
$< \boldsymbol{S}, \text{SAT} > \sim_{\boldsymbol{O}} < \boldsymbol{S}'\text{SAT}' >$
iff
$< \boldsymbol{S}, \text{SAT} >$ is both stronger and weaker than $< \boldsymbol{S}', \text{SAT}' >$
iff
for all $O_1, O_2 \in \boldsymbol{O}$, $(O_1 \sim_{\boldsymbol{S}} O_2$ iff $O_1 \sim_{\boldsymbol{S}'} O_2)$.

Behaviour structures are a special case of specification structures; the extra feature being that the relation BEH is functional, i.e. BEH induces an equivalence relation, i.e., BEH induces a partitioning on $\boldsymbol{O}$.

Comparing a behaviour structure and a specification structure turns out to be no different from comparing two specification structures.

When comparing two behaviour structures the special nature of BEH does have some effect.

As, in the context of the running example, BEH induces a partitioning on $\boldsymbol{O}$, abstractness respectively expressiveness reduce to this partitioning being coarses respectively finer. The notions weaker and stronger collapse to equivalence.

Running example.
$\boldsymbol{B}$ is abstract with respect to $\boldsymbol{B}'$ iff $\boldsymbol{B}$ is a coarser partition of O than $\boldsymbol{B}'$.
$\boldsymbol{B}$ is expressive with respect to $\boldsymbol{B}'$ iff $\boldsymbol{B}$ is a finer partition of O than $\boldsymbol{B}'$.
$\boldsymbol{B}$ is equivalent to $\boldsymbol{B}'$ iff $\boldsymbol{B} = \boldsymbol{B}'$.
$\boldsymbol{B}$ is weaker than $\boldsymbol{B}'$ iff $\boldsymbol{B} = \boldsymbol{B}'$.
$\boldsymbol{B}$ is stronger than $\boldsymbol{B}'$ iff $\boldsymbol{B} = \boldsymbol{B}'$.

Proposition 3.7.
For behaviour structures,
$\boldsymbol{B}$ is stronger than $\boldsymbol{B}'$ iff $\boldsymbol{B} \sim_{\boldsymbol{O}} \boldsymbol{B}'$.

Remark.
We wish to point out that it is very easy to construct specification structures that possess the properties mentioned thus far. Namely, by choosing for $\boldsymbol{S}$ the standard representation in terms of subsets of $\boldsymbol{O}$. The problem is, that the specification structures thus obtained are not, in general, justifiable with respect to the intuition as discussed in Section 2. This was one of the reasons that led us to consider intuitive justification.

3.2.2.2. *Soundness and completeness*

Proof structures.

The discussion of formalisms has up till this point been confined to the following framework:

1. A set of formulae of form O SAT S;
2. A model to determine whether a formula is true, namely a specification structure $< \boldsymbol{S}, \text{SAT} >$; a formula is true if and only if this is the case according to the definition of the relation SAT.

To determine via a model whether a formula is true involves the use of properties of that model. At the current level, this requires only application of the relation SAT, but in more complex formalisms this may, for example, involve considerable use of intricate mathematical structure of the model.

To facilitate the discussion in this section, we restate this in slightly more formal notation. All we need as representation of a language is a set of formulae, Φ.

Definition 3.8 (cf. [Da80]).
A model M for $\boldsymbol{\Phi}$ is a structure that allows one to determine for each $\Phi \in \boldsymbol{\Phi}$ whether Φ is true.

$M \models \Phi$ denotes that Φ is true in M.

There is a small complication here in that in practice one is not so much interested in whether a formula is true for one particular model, but rather for all models that are of a certain form.

Definition 3.9.
A model structure $\boldsymbol{M}$ for Φ is a set of models for Φ.

Φ is valid for $\boldsymbol{M}$, notation $\boldsymbol{M} \models \Phi$, if for all $M \in \boldsymbol{M}$, $M \models \Phi$.

A proof system aims at avoiding the usually complicated procedure of establishing whether $\boldsymbol{M} \models \Phi$ by circumventing the use of models. A subset of formulae, axioms, and also a set of rules to derive formulae are defined. These definitions are purely in terms of the syntax, i.e., the form, of the formulae—no use is made of models.

Definition 3.10.
A proof structure $< \boldsymbol{A}, \boldsymbol{R} >$ for a set of formulae Φ is a syntactically defined

(i) set of formulae $\boldsymbol{A}$, called axioms and
(ii) set of rules $\boldsymbol{R}$, called proof rules.

$\Phi, \Phi_1, \Phi_2, ...$ are variables ranging over $\boldsymbol{\Phi}$, $A, A_1, A_2, ...$ range over $\boldsymbol{A}$ and $R, R_1, R_2, ...$ range over $\boldsymbol{R}$.

A formula Φ is provable under no assumptions in $< \boldsymbol{A}, \boldsymbol{R} >$, notation $\vdash_{<\boldsymbol{A},\boldsymbol{R}>} \Phi$, if Φ can be derived from axioms in $\boldsymbol{A}$ through use of rules in $\boldsymbol{R}$. Where no confusion is likely to occur, $< \boldsymbol{A}, \boldsymbol{R} >$ is omitted from the notation.

Although we motivated derivability as, in some sense, an alternative for being true in a model, this connection between proof structure and model needs to be established. Although there is very little structure present at this level, it is still possible to discuss most of the notions that apply to proof structures.
In fact, the property that a proof structure captures the interpretation of formulae in the model can be formulated in an even more abstract setting. All that is necessary is:

1. A set $\boldsymbol{\Phi}$ of formulae;
2. a model structure $\boldsymbol{M}$ to determine $\models \Phi$;
3. a proof structure $< \boldsymbol{A}, \boldsymbol{R} >$ to determine $\vdash \Phi$.

Definition 3.11.
For $\boldsymbol{\Phi}$, a proof structure $< \boldsymbol{A}, \boldsymbol{R} >$ is sound with respect to
a model structure $\boldsymbol{M}$ if
for all $\Phi \in \boldsymbol{\Phi}$, if $\vdash \Phi$ then $\models \Phi$.

For $\boldsymbol{\Phi}$, a proof structure $< \boldsymbol{A}, \boldsymbol{R} >$ is complete with respect to
a model structure $\boldsymbol{M}$ if
for all $\Phi \in \boldsymbol{\Phi}$, if $\models \Phi$ then $\vdash \Phi$.

In the case of formalisms to describe computations, complete proof structures usually do not exist. The reason is, that the domains over which is computed allow no complete axiomatisation of their properties, e.g., arithmetic of natural numbers.

The way out is, to split completeness into completeness with respect to the axiomatisation of all non-domain properties and completeness with respect to the domain properties and only require the former. This leads to the notion of relative completeness. Intuitively, this notion states that a formalism is complete, given that all domain properties that are true can be used as axioms. In the sequel, completeness always means relative completeness.

For a thorough treatment of this subject in the context of imperative sequential programming, see [Ap81]. On notions of soundness, see [OD82].

3.3. Level 2—Structured objects

In this subsection we discuss the role of structure in connection with formalisms. We only address the effects of structure that are present explicitly in the system under description.

Other ways of structuring descriptions can be envisaged, for instance conceptual structuring of behaviour independent of the structure of the system that generates that behaviour. An example is the Statecharts approach [Ha87], where descriptions are structured according to conceptual modes rather than actual states in which a system

can be operating. Another possibility is structuring according to the execution rather than the implementation of an algorithm. We do not investigate such other options in this paper.

Two well-known occurrences of structure are in compositional semantics for programmaing languages and proof systems for specification formalisms.
Our aim in this subsection is to investigate the relationship between these two via the models for specification formalisms.

3.3.1. The framework

When motivating the introduction of object structures we argued that, in order to discuss formalisms, one first needs something to describe. Similarly, in order to discuss structure, that structure needs to be introduced somewhere. In the framework presented thus far, no such structure is present.

In the literature, the most prominent places where structure is considered are the realm of compositional semantics for programming languages, and the realm of compositional or modular proof systems for specification formalisms. The structure is in both cases the syntactical structure of the programming language. The idea is, that this structure should be reflected in, respectively, the semantics of the programming language and the proof system.

The most obvious way to introduce structure in the present framework seems therefore to be to add a language, with structure, to describe objects. This would amount to regarding the objects as programs and the language as a programming language. However, since programs are constructed from atomic programs using constructors the most direct approach is to represent programs as objects. So to discuss structure, it is the object domain $\boldsymbol{O}$ that needs to be extended accordingly with constructors.

Definition 3.12.
An object structure $<\boldsymbol{O},\boldsymbol{C}>$ consists of

(i) a set of objects $\boldsymbol{O}$ and

(ii) a set of constructors $\boldsymbol{C}$;
a constructor is a function of finite arity, say n, from $\boldsymbol{O} \times ... \times \boldsymbol{O} \rightarrow \boldsymbol{O}$.

$O, O_1, O_2, ...$ are variables ranging over $\boldsymbol{O}$, $C, C_1, C_2, ...$ range over $\boldsymbol{C}$.

Running example.

(i) $\boldsymbol{O}$ again is some set of objects, but now these objects have structure:
They are either basic objects as used in the canonical example up till now, or complex objects that are generated using the constructor given in (ii);

(ii) $\boldsymbol{C}$ is a set consisting of one constructor, $C_{\cup} : \boldsymbol{O} \times ... \times \boldsymbol{O} \rightarrow \boldsymbol{O}$.

$O, O_1, O_2, ...$ are variables ranging over $\boldsymbol{O}$.

3.3.2. The notions

At the level of structuring that we consider in this subsection, the following entries along the axes will be discussed.

Concepts

Full abstractness, compositionality, modularity, completeness.

Areas

(Constructors in) behaviour structures,
(constructors in) specification structures,
(construction rules in) proof structures.

Investigations in the context of temporal logic based semantics can be found in [Fi87].

3.3.2.1. *Full abstractness*

We start by defining full abstractness of one behaviour structure with respect to another. This notion only just belongs to the subsection on structure—it does involve structure of the objects, but not of any of the other structures. The intuitive idea is that "fully abstract" behaviours should distinguish between objects if and only if these objects can be distinguished in a given behaviour structure after putting them in some context. The name is rather ill-chosen: the notion is about equivalence, i.e., as much about expressiveness as about abstractness.

Intuitively, a context is a piece of program with "holes" in which a subprogram can be "plugged", possibly at several places at the same time, resulting in a complete program. Of course, in general not all subprograms fit all contexts.

We take the formal definition form [HGR87], which in turn is based upon [Pl83]. We use the notion of equivalence defined in Subsection 3.2.2.1.

Definition 3.13.
A context is a (partial) function $\boldsymbol{O} \to \boldsymbol{O}$.
A context structure $\boldsymbol{H}$ is a set of contexts.
$H_1, H_2, \dots$ are variables ranging over $\boldsymbol{H}$.

Definition 3.14.
A behaviour structure $\boldsymbol{B}$ is fully abstract with respect to a behaviour structure $\boldsymbol{B}'$ and a context structure $\boldsymbol{H}$ if
for all $O_1, O_2 \in \boldsymbol{O}$,
$O_1 \sim_{\boldsymbol{B}} O_2$ iff for all $H \in \boldsymbol{H}$, $H(O_1) \sim_{\boldsymbol{B}'} H(O_2)$.

As can be seen from the definition, contexts are even defined independently of the structure $\boldsymbol{C}$ of the object structure. However, in all practical cases, $\boldsymbol{H}$ is defined via $\boldsymbol{C}$, justifying its inclusion in Subsection 3.3.

Remark.
A fully abstract behaviour structure with respect to a given behaviour structure $\boldsymbol{B}'$ can

of course always be constructed. Just define the behaviour structure $< \boldsymbol{B_f}, \text{BEH}_f >$ as follows.

$\boldsymbol{B_f} = \{B_f \subseteq \boldsymbol{O}$: for all $O_1, O_2 \in B_f$, $H \in \boldsymbol{H}$. $H(O_1) \sim_{\boldsymbol{B}'} H(O_2)$ and B_f is maximal with respect to this requirement},

BEH_f is "$\in$".

Again, apart from the abstractness property, there is no intuitive support for this representation of behaviour.

3.3.2.2. *Compositionality and modularity*

We aim in this subsection to establish a link between the instantiation of compositionality in the realm of behaviour structures and in the realm of proof structures via the realm of specification structures.

3.3.2.2.1. *Compositionality and modularity in the model*

We return to the question how structure of the object structure is reflected in the other structures.

The most well-known instantiation of compositionality is, firstly, in the area of a programming language and its semantics. Secondly, this concept, together with modularity, is used in the area of proof structures.

In our approach, the first case corresponds to an object structure with a behaviour structure. A behaviour structure, however, is just a special instantiation of specification structure. This is a reason to extend the notion to specification structures.

A more important motivation, however, derives from the second case. As presented by Zwiers in [Zw88], there are two related concepts, compositionality and modularity, in the area of proof structures. We argue that it is possible to distinguish compositionality and modularity before going to the realm of proof structures, namely in the area of specification structures.

We go from the particular to the general in this case and discuss behaviour structures first and specification structures later.

Behaviour structures

Intuitively, a behaviour structure is compositional if the behaviour of a structured object is obtained from the behaviours of its subobjects. Thus, compositionality is a property of the definition of the semantics, or, in our framework, of the definition of the relation BEH.

To further aid the intuition, we mention a typical example of a non-compositional behaviour structure. Assume that the first and last state of the execution of a program in isolation are taken as behaviour for shared variable programs. The parallel composition operator is then inherently not compositional—the information necessary to describe the interaction of parallel components during execution is simply not available.

An essential feature of a compositional behaviour structure, independent of the parti-

cular way in which the relation BEH is defined, is therefore, that enough information is present in the behaviours of the constituents of a structured object to obtain the behaviour of the structured object.

This, in fact, is the rather subtle difference between requiring that the behaviour of a structured component is obtained from its constituent objects or could in principle be obtained. In our formalism the latter could be expressed as follows.

Definition 3.15.
A behaviour structure $< \boldsymbol{B}, \text{BEH} >$ for object structure $< \boldsymbol{O}, \boldsymbol{C} >$ is potentially compositional with respect to a constructor $C \in \boldsymbol{C}$ of arity n
if
for all $O_1, ..., O_n, O'_1, ..., O'_n \in \boldsymbol{C}$,
if $C(O_1, ..., O_n) \not\sim C(O'_1, ..., O'_n)$
then there is $i \in I$ such that $O_i \not\sim O'_i$.

As this part still generalises directly to specification structures, we go to this setting for a moment to provide some further definitions and a proposition.

Definition 3.16.
A specification structure $< \boldsymbol{S}, \text{SAT} >$ for object structure $< \boldsymbol{O}, \boldsymbol{C} >$ is potentially compositional with respect to a constructor $C \in \boldsymbol{C}$ of arity n
if
for all $O_1, ..., O_n$ and $O'_1, ..., O'_n \in \boldsymbol{O}$,
if $C(O_1, ..., O_n) \not\sim C(O'_1, ..., O'_n)$ then there is $i \in I$ such that $O_i \not\sim O'_i$.

The obvious extension to specification structures as a whole is to require potential compositionality for all constructors.

Definition 3.17.
$< \boldsymbol{S}, \text{SAT} >$ is potentially compositional with respect to $< \boldsymbol{O}, \boldsymbol{C} >$ if $< \boldsymbol{S}, \text{SAT} >$ is potentially compositional with respect to all $C \in \boldsymbol{C}$.

Running example.
We now define a specification structure for the object structure in the running example.
A property is still represented as a set of basic objects.
We extend the specification structure to the structured objects by changing the satisfaction relation from "$\in$" to set inclusion, "$\subseteq$", between the set of the basic objects that the object is built from and the specification.
The only constructor, $C_\cup$, is potentially compositional, and so is therefore the whole structure. Note that no criterion has been formulated yet.

Proposition 3.8 (cf. [HGR86]).
If $< \boldsymbol{B}, \text{BEH} >$ is fully abstract for $< \boldsymbol{O}, \boldsymbol{C} >$ with respect to itself then
$< \boldsymbol{B}, \text{BEH} >$ is potentially compositional.

We now return to behaviour structures to discuss compositionality.

To focus the discussion, we provide the formal definitions. As these may seem rather complicated, some explanation and justification follows. As mentioned above, compositionality is a property of the definition of the BEH relation. To enable a formulation of the property, a little more about the form of the definition of this relation has to be required.

Definition 3.18.
A behaviour structure $< \boldsymbol{B}, \boldsymbol{K_C}, \text{BEH} >$ for an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ consists of

(i) a set of behaviours $\boldsymbol{B}$,

(ii) a functional relation $\text{BEH} \subseteq \boldsymbol{O} \times \boldsymbol{B}$ defined using

a. a set $\boldsymbol{K_C}$ of criteria (predicates) of finite arity on $\boldsymbol{B} \times ... \times \boldsymbol{B}$,

b. a one to one correspondence $K : \boldsymbol{C} \rightarrow \boldsymbol{K_C}$ between the constructors of arity n from $\boldsymbol{C}$ and the criteria of arity $n+1$ from $\boldsymbol{K_C}$.

$B, B_1, B_2, ...$ are variables ranging over $\boldsymbol{B}$. K_C denotes the criterion corresponding to the constructor denoted by C.

We start by defining compositionality with respect to one constructor. We again point to the fact that, in contrast to potential compositionality, compositionality is a property of the definition of the behaviour structure.

To avoid complicating the definitions unduly, the index set I is often left implicit.

Definition 3.19.
A behaviour structure $< \boldsymbol{B}, \boldsymbol{K_C}, \text{BEH} >$ for an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ is compositional with respect to a constructor $C \in \boldsymbol{C}$ of arity n
if
for all $O_1, ..., O_n \in \boldsymbol{O}$,
$C(O_1, ..., O_n)\ \text{BEH}\ B$ is defined as
$C(O_1, ..., O_n)\ \text{BEH}\ B$, $B \in \boldsymbol{B}$ iff there are $B_i \in \boldsymbol{B}$, $i \in I$, such that $O_i\ \text{BEH}\ B_i$ and $K_C(B_1, ..., B_n, B)$.

Running example.
Turn the specification structure into a, trivial, behaviour structure by changing "$\subseteq$" to "$=$".
A compositional definition of the $C_\cup$-operator is then
$C_\cup(O_1, ..., O_n)\ \text{BEH}\ B$ iff there are B_i, $i \in I$, such that $O_i\ \text{BEH}\ B_i$ and $\bigcup_{i \in I} B_i = B$.

Definition 3.20.
A behaviour structure $< \boldsymbol{B}, \boldsymbol{K_C}, \text{BEH} >$ is compositional with respect to an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ if $< \boldsymbol{S}, \text{SAT} >$ is compositional with respect to all $C \in \boldsymbol{C}$.

Remark.
This is equivalent to the well-known notion that a semantics is compositional iff the semantics of a component is defined as a function of the semantics of its subcomponents. The idea is that compositionality is captured by requiring, for each constructor,

a criterion to decide the behaviour of a structured object, given the behaviours of its subobjects. The criterion is functional in B, because BEH has to be functional, a requirement for a behaviour structure.

Bypassing the obvious analogeous proposition for single constructors the following proposition should now be evident.

Proposition 3.9.
$< B, \text{BEH} >$ is potentially compositional iff
there is a compositional $< B', K_C, \text{BEH}' >$ such that $< B, \text{BEH} > \sim_O < B', K_C, \text{BEH}' >$.

The equivalence relation is extended in the obvious manner: K_C plays no role in the definition.

Example.
Rather than using the running example here, we suggest to consider the more practical case of an operationally defined behaviour structure, e.g., an operational semantics, that is not compositional but is, by chance, equivalent to a compositional one. Then that operational semantics is potentially compositional.

One might argue that for compositionality there should be an operator that actually constructs the behaviour for the structured object from the behaviours of its constituents. This constructor can, if so desired, be defined immediately using the definition provided: define $\text{BEH}(C(O_1, ..., O_n))$ to be the behaviour B such that $K_C(B_1, ..., B_n, B)$. Again, as BEH has to be functional, the criterion should allow only one behaviour for any selection of $O_1, ..., O_n$.

We have chosen the form of the definition as given because it generalises more easily to specification structures; this generalisation is the next topic.

Specification structures

We now introduce compositionality and modularity for specification structures. As far as we know, the resulting notions in this area have not been introduced in the literature. Apart from showing how they arise as a natural extension of compositionality to specification structures, we also show the relationship with compositionality and modularity as introduced for proof structures by Zwiers in [Zw88]. The intention is to improve the understanding of the concept by showing the links—no claims are made as to practical relevance.

We differ slightly from Zwiers in that we first introduce the notions for single constructors rather than directly for whole structures. We shall explain differences and similarities during the development of the notions.

Similarly as above, to be compositional a specification structure should reflect the structure of the objects. Also, again compositionality is to be a property of the definition of the relation SAT. A point of difference is that objects have just one behaviour, but may satisfy many different specifications.

This influences the definition at two points:

(a) The structured object may satisfy more than one specification and

(b) the subobjects may satisfy more than one specification.

Recall that, intuitively, a behaviour structure was said to be compositional if the behaviour of a structured object was given as a function of the behaviours of its subobjects.

It is clear, that point (a) can only lead to an adaptation of the definition to the effect that for all specifications of a structured object satisfaction should be defined via specifications of the subobjects.

It is also clear, that the choice of specifications for the subobjects mentioned in point (b) cannot be arbitrary. For instance, very little about a specification of a structured object could be inferred from weak specifications of its subobjects.

The most obvious adaptation concerning point (b) is to just require that there exists a set of subspecifications that enables one to decide satisfaction of the specification. This choice we formalise now; another option will be discussed later. The name compositionality is retained because its use is already established for the corresponding notion for proof structures.

To define a notion of compositionality that is a property of the definition of the relation SAT, we have to extend this relation in a similar manner as the relation BEH before.

Definition 3.21.
A specification structure $< \boldsymbol{S}, \boldsymbol{K_C}, \mathrm{SAT} >$ for an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ consists of

(i) a set of specifications $\boldsymbol{S}$,

(ii) a relation $\mathrm{SAT} \subseteq \boldsymbol{O} \times \boldsymbol{S}$ defined using

a. a set $\boldsymbol{K_C}$ of criteria of finite arity on $\boldsymbol{S} \times ... \times \boldsymbol{S}$,

b. a one to one correspondence between the constructors of arity n from $\boldsymbol{C}$ and the criteria of arity $n+1$ from $\boldsymbol{K_C}$.

$S, S_1, S_2, ...$ are variables ranging over $\boldsymbol{S}$, $K_C, K_{C_1}, K_{C_2}, ...$ range over $\boldsymbol{K_C}$. K_C denotes the criterion corresponding to the constructor denoted by C.

Definition 3.22.
A specification structure $< \boldsymbol{S}, \boldsymbol{K_C}, \mathrm{SAT} >$ for an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ is compositional with respect to a constructor $C \in \boldsymbol{C}$ of arity n
if
for all $O_1, ..., O_n \in \boldsymbol{O}$,
$C(O_1, ..., O_n)\,\mathrm{SAT}\,S$ is defined as:
$C(O_1, ..., O_n)\,\mathrm{SAT}\,S$ iff
there are $S_i \in \boldsymbol{S}$, $i \in I$, such that
$O_i\,\mathrm{SAT}\,S_i$ and $K_C(S_1, ..., S_n, S)$.

Definition 3.23.
A specification structure $< \boldsymbol{S}, \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT} >$ is compositional with respect to an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ if $< \boldsymbol{S}, \text{SAT} >$ is compositional with respect to all $C \in \boldsymbol{C}$.

Running example.
A compositional definition of the constructor, $C_{\cup}$, is
$C_{\cup}(O_1, ..., O_n)\,\text{SAT}\,S$ if there are S_i such that $O_i\,\text{BEH}\,S_i$ and $\bigcup_{i \in I} S_i \subseteq S$.
Indeed, the criterion corresponding to $C_{\cup}$, say $K_{C_{\cup}}$, only depends on $S_1, ..., S_n$ and S.

Proposition 3.10.
$< \boldsymbol{S}, \text{SAT} >$ is potentially compositional iff
there is a compositional $< \boldsymbol{S}', \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT}' >$ such that $< \boldsymbol{S}, \text{SAT} > \sim_{\boldsymbol{O}} < \boldsymbol{S}', \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT} >$.

Compositionality poses only very modest requirements on the criterion. It is, for example, not required that the criterion evaluate to the same result on even equivalent specifications. Rather than to base an alternative notion on a loose remark like this, we investigate this problem in a more general manner to arrive at the second notion concerning structure.

We recall that the aim is to reflect the structure of an object in the way in which it is decided that it satisfies a specification. Compositionality requires only that there exists a set of subspecifications, satisfied by the subcomponents, that contains enough information to make the criterion hold. Compositionality thus allows much freedom in the choice of the criterion.

In particular the following rather problematic situation can occur:
Let $S_1, ..., S_n$ be specifications that are generated as a top down development step with respect to a constructor C of a specification S. The justification of this step should be that every set of subobjects such that $O_1\,\text{SAT}\,S_1, ..., O_n\,\text{SAT}\,S_n$ combines to an object such that $C(O_1, ..., O_n)\,\text{SAT}\,S$. Assume this to be the case. In this case one would like to have $K_C(S_1, ..., S_n, S)$ fulfilled. Compositionality does not ensure this. This is because the only requirement is that for each of the above mentioned sets of objects there exist corresponding specifications for which the criterion holds. In principle, there could be infinitely many such sets of objects; a different set, i.e. infinitely many sets, of subspecifications might be needed though to establish that each combination satisfies S via the criterion from the definition.

We now give a stronger requirement that, intuitively, states that the criterion uses all the information present in the set of subspecifications.

The idea is, more precisely, that if there is enough information in the subspecifications, to decide that the combination of any collection of objects that satisfy them satisfies the larger specification, then the criterion should also enable one to decide so.

Definition 3.24.
A specification structure $< \boldsymbol{S}, \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT} >$ for an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ is modular with respect to a constructor $C \in \boldsymbol{C}$ of arity n
if
$< \boldsymbol{S}, \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT} >$ is compositional and

for all $S_1, ..., S_n$, $S \in \boldsymbol{S}$ it holds that
if for all $O_1, ..., O_n \in \boldsymbol{O}$ such that $O_i \,\text{SAT}\, S_i$, $i \in I$, $C(O_1, ..., O_n) \,\text{SAT}\, S$
then $K_C(S_1, ..., S_n, S)$.

Note that in the compositional case one might have to find different collections of S_i's for each collection of O_i's to enable use of the criterion.

Definition 3.25.
$< \boldsymbol{S}, \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT} >$ is modular with respect to $< \boldsymbol{O}, \boldsymbol{C} >$ if it is modular with respect to all $C \in \boldsymbol{C}$.

It is easy to see, in the running example, that the criterion that made $C_\cup$ compositional, $\bigcup_{i \in I} S_1 \subseteq S$, also makes $C_\cup$ modular.

To obtain a criterion that is compositional but not modular, intuitively some information present in subspecifications needs to be thrown away by the criterion. However, in order to remain compositional it must always be possible to offset this loss by choosing a different subspecification for any particular subcomponent satisfying the subspecification.

There is a subtle distinction between cases where such "better" subspecifications can or can not be chosen solely on the basis of the original subspecifications.

The examples given here are based on work by Widom, [Wi87, WGS87], in the area of trace-based formalisms. We prefer to stay close to the running example, both for ease of exposition and to show that the examples are independent of the trace framework.

Example 1.
Assume the set of specifications, $\boldsymbol{S}$ to be extended, like before, by adding all $S \; U \; \{*\}$, where $*$ is not an object. Let the satisfaction relation remain "$\subseteq$".
Let also the criterion remain the same.

Now assume $O_i \,\text{SAT}\, S_i \cup \{*\}$. Then also $O_i \,\text{SAT}\, S_i$. It is therefore possible to infer, using the criterion with these specifications, that $C_\cup(O_1, ..., O_n) \,\text{SAT}\, S$, where $S = \bigcup_{i \in I} S_i$. However, the criterion does not hold for $S_i \cup \{*\}$, as $U_{i \in I} S_i \cup \{*\} \not\subseteq S$.

In this example, it is possible to find subspecifications for which the criterion does hold by only looking at the subspecifications: just delete the $*$.
This is the situation treated in [Zw88], where a kernel rule is added enabling to substitute specifications by better ones.

This can not always be done.

Example 2.
Assume the object set $\boldsymbol{O}$ is also extended, namely with objects "+" and "−". The specification structure is extended correspondingly with, for all S, $S \cup \{+\}$ and $S \cup \{-\}$, but also retains the extra $S \cup \{*\}$ specifications.

The satisfaction relation is canonically extended, in the obvious way, with the extra proviso that for $S' \subseteq S$ both $S' \cup \{+\}$ and $S' \cup \{-\}$ satisfy $S \cup \{*\}$.

The criterion is also canonically extended, but with the extra proviso that as soon as a subspecification contains $*$, it evaluates to false.

Now if a subspecification contains $*$, one has to choose a better one with either $+$ or $-$. Contrary to the situation before, this can not be done on the basis of the subspecification only: one has to know the syntax of the specified object.

This prompts a further refinement of the notion of modularity.

Definition 3.26.
A specification structure $< \boldsymbol{S}, \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT} >$ for an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ is weakly modular with respect to a constructor $C \in \boldsymbol{C}$ of arity n
if
$< \boldsymbol{S}, \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT} >$ is compositional and
for all $S_1, ..., S_n, S \in \boldsymbol{S}$ it holds that
if for all $O_1, ..., O_n \in \boldsymbol{O}$ such that $O_i \,\text{SAT}\, S_i$, $i \in I$, $C(O_1, ..., O_n) \,\text{SAT}\, S$
then there are $S_i', S' \in \boldsymbol{S}$ such that $S_i \sim_{\boldsymbol{O}} S_i'$, $S \sim_{\boldsymbol{O}} S'$ and $K_C(S_1', ..., S_n', S')$.
This is a quite useful property, as it states that the criterion is in fact independent of the syntax of the subcomponents, provided that the right subspecifications are used.

Proposition 3.11.
If $< \boldsymbol{S}, \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT} >$ is modular, then $< \boldsymbol{S}, \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT} >$ is weakly modular.

The converse does not hold.

Proposition 3.12.
If $< \boldsymbol{S}, \text{SAT} >$ is potentially compositional
then there is a modular $< \boldsymbol{S}', \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT}' >$ such that $< \boldsymbol{S}, \text{SAT} > \sim_{\boldsymbol{O}} < \boldsymbol{S}', \boldsymbol{K}_{\boldsymbol{C}}, \text{SAT}' >$.

The converse does not hold.

Remarks.
It may seem strange that (potential) compositionality is enough to guarantee the existence of an equivalent modular specification structure. We attempt an intuitive explanation.

Potential compositionality means, that there is enough information in some collections of subspecifications to define a criterion (to establish, for all collections of subobjects plus a specification, whether their combination satisfies that specification). However, it is not required to use this information to the full; this is the additional requirement to achieve (weak) modularity.

Therefore, potential compositionality is about the presence of information, compositionality and (weak) modularity about its use in different ways.

Note, that it is not enough to require about the criterion that it is insensitive to specification equivalence to obtain modularity. More formally this can be stated as follows.

Definition 3.27.
K_C is insensitive with respect to specification equivalence $\sim_{\boldsymbol{O}}$ if

for all $S_1, ..., S_n, S, S'_1, ..., S'_n, S' \in \boldsymbol{S}$
such that $S_i \sim {}_O S'_i$, $i \in \{1, ..., n\}$, and $S \sim {}_O S'$,
$K_C(S_1, ..., S_n, S)$ iff $(S'_1, ..., S'_n, S')$.

Proposition 3.13.
If K_C is modular, then all K_C are insensitive to specification equivalence.

The converse does not hold.

3.3.2.2.2. *Compositionality and modularity for proof structures*

The concept of structuring is also applicable to proof systems. The interpretation here is, that the rules of a proof structure reflect the structure of the object structure.

Definition 3.28.
A constructor rule $R \in \boldsymbol{R}$ for a constructor $C \in \boldsymbol{C}$ from a proof structure $< \boldsymbol{A}, \boldsymbol{R} >$ for a specification structure $< \boldsymbol{S}, \text{SAT} >$ for an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ is compositional if
R is of the form

if $\vdash O_i \text{ SAT } S_i$, $i \in I$, and $\vdash G_C(S_1, ..., S_n, S)$ (where G_C denotes an $n+1$-ary predicate if C is an n-ary constructor)

then $\vdash C(O_1, ..., O_n) \text{ SAT } S$.

Definition 3.29.
A proof structure $< \boldsymbol{A}, \boldsymbol{R} >$ for a specification structure $< \boldsymbol{S}, \text{SAT} >$ for an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ is compositional if all constructor rules are compositional.

Definition 3.30.
A constructor rule $R \in \boldsymbol{R}$ for a constructor $C \in \boldsymbol{C}$ from a proof structure $< \boldsymbol{A}, \boldsymbol{R} >$ for a specification structure $< \boldsymbol{S}, \text{SAT} >$ for an object structure $< \boldsymbol{O}, \boldsymbol{C} >$ is compositionally complete
if
R is compositional and
if $\models C(O_1, ..., C_n) \text{ SAT } S$
then there are S_i, $i \in I$, such that $\models O_i \text{ SAT } S_i$ and $\vdash G_C(S', ..., S_n, S)$.

The idea is, that for the right choice of specifications for given objects if their combination satisfies a specification, then this can be proven using the rule. It is the obligation of the proof structure as a whole to ensure that also $\vdash O_i \text{ SAT } S_i$ can be obtained.

Remark.
We wish, at this point, to draw attention to the fact that although in the definitions in the specification structure as well as in the rules in the proof structure criteria can occur, they need neither be connected nor be present in both frameworks at the same time.

Definition 3.31.
A proof structure $< \mathbf{A}, \mathbf{R} >$ for a specification structure $< \mathbf{S}, \mathrm{SAT} >$ for an object structure $< \mathbf{O}, \mathbf{C} >$ is constructor compositionally complete if all constructor rules are compositionally complete.

Note, that completeness of the proof structure still depends upon completeness of possibly other rules as well.

Similarly as in the case of the model, one wishes to state independence of object syntax in the proof system.

Definition 3.32.
A constructor rule $R \in \mathbf{R}$ for a constructor $C \in \mathbf{C}$ from a proof structure $< \mathbf{A}, \mathbf{R} >$ for a specification structure $< \mathbf{S}, \mathrm{SAT} >$ for an object structure $< \mathbf{O}, \mathbf{C} >$ is modular if
R is compositional
and
for all $S_1, ..., S_n, S \in \mathbf{S}$ it holds that
if for all $O_1, ..., O_n \in \mathbf{O}$ such that $\models O_i \,\mathrm{SAT}\, S_i$, $i \in I$, $\vdash C(O_1, ..., O_n) \,\mathrm{SAT}\, S$
then $\vdash G_C(S_1, ..., S_n, S)$.

Note that this only states that the G_C is independent with respect to choice of possibly better subspecifications—the rule need not be complete.

Definition 3.33.
A constructor rule $R \in \mathbf{R}$ for a constructor $C \in \mathbf{C}$ from a proof structure $< \mathbf{A}, \mathbf{R} >$ for a specification structure $< \mathbf{S}, \mathrm{SAT} >$ for an object structure $< \mathbf{O}, \mathbf{C} >$ is modular complete if
R is modular
and
if $\models C(O_1, ..., O_n) \,\mathrm{SAT}\, S$
then there are S_i, $i \in I$, such that $\models O_i \,\mathrm{SAT}\, S_i$ and $\vdash G_C(S', ..., S_n, S)$.

Definition 3.34.
A proof structure $< \mathbf{A}, \mathbf{R} >$ for a specification structure $< \mathbf{S}, \mathrm{SAT} >$ for an object structure $< \mathbf{O}, \mathbf{C} >$ is constructor modular complete if all constructor rules are constructor modular complete.

Comparison to Zwier's approach

Another way to arrive at compositionality is the approach by Zwiers, presented in [Zw88], using the notion of black boxes.

We briefly introduce this approach, referring to [Zw88] for further details. The motivation there is mainly from the proof theoretic point of view.

Intuitively, compositionality states that for given, concrete, programs the proofs reflect the structure of the program whereas modulatity additionally requires that this is the case for programs that are constructed of black boxes for which only specifications are

given.

To formalise these intuitions, it is necessary to have variables in the programming language to denote black boxes.

Definition 3.35.
An object structure $< \boldsymbol{O}, \boldsymbol{C} >$ is defined as above, with the addition that $\boldsymbol{O}$ is partitioned into:

(a) A set of concrete objects $\boldsymbol{O}'$ and

(b) a set of black box objects $\boldsymbol{\theta}$.

Let $O, O_1, O_2, ...$ be variables ranging over $\boldsymbol{O}$ and let $\theta, \theta_1, \theta_2, ...$ range over $\boldsymbol{\theta}$. Let $O(\theta_1, ..., \theta_n)$ denote a structured object that has black box objects $\theta_1, ..., \theta_n$ among its constituents. Again, the object structure is taken to represent a programming language. It is allowed to mix programming language and black box variables: a mixed term framework.

Definition 3.36 [Zw88].
$\theta_1 \text{ SAT } S, ..., \theta_n \text{ SAT } S_n$ semantically implies $O(\theta_1, ..., \theta_n) \text{ SAT } S$, notation
$\theta_1 \text{ SAT } S, ..., \theta_n \text{ SAT } S_n \models O(\theta_1, ..., \theta_n) \text{ SAT } S$, if
for all $O_1, ..., O_n \in \boldsymbol{O}$,
if $\models O_1 \text{ SAT } S_1, ..., \models O_n \text{ SAT } S_n$ then $\models O(O_1, ..., O_n) \text{ SAT } S$.

A similar definition applies for objects rather than black boxes.

Definition 3.37 [Zw88].
A proof structure is compositionally complete if
if $\models O_1 \text{ SAT } S_1, ..., \models O_n \text{ SAT } S_n$ and
$O_1 \text{ SAT } S_1, ..., O_n \text{ SAT } S_n \models O(O_1, ..., O_n) \text{ SAT } S$
then there are S_i' such that
$\models O_i \text{ SAT } S_i$ and $O_1 \text{ SAT } S_1, ..., O_n \text{ SAT } S_n \vdash O(O_1, ..., O_n) \text{ SAT } S$.

Definition 3.38 [Zw88].
A proof structure is modular complete if
for all satisfiable $S_1, ..., S_n$ and all S,
if $\theta_1 \text{ SAT } S_1, ..., \theta_n \text{ SAT } S_n \models O(\theta_1, ..., \theta_n) \text{ SAT } S$
then $\theta_1 \text{ SAT } S_1, ..., \theta_n \text{ SAT } S_n \vdash O(\theta_1, ..., \theta_n) \text{ SAT } S$.

The relationship between the two approaches is as follows.

Proposition 3.14.
If a proof structure is constructor compositionally complete and complete with respect to the other rules then it is compositionally complete.

The converse generally does not hold; see [Zw88] for explanation involving kernel rules.

Proposition 3.15.
If a proof structure is constructor modular complete and it is complete with respect to the other rules then it is modular complete.

The converse generally does not hold.

Remark.
We wish to draw attention to the fact that compositionality or modularity in a model and for a proof structure are to some extent independent. K_C and G_C need not be the same, for instance.

Proposition 3.16.
Let expression of arbitrary $\boldsymbol{K_C}$ be allowed, then
there exists a constructor compositionally complete proof structure and a constructor modular complete proof structure for $< \boldsymbol{S}, \text{SAT} >$
iff
$< \boldsymbol{S}, \text{SAT} >$ is potentially compositional.

The usefulness of these properties for proof structures is discussed thoroughly in [Zw88]. We consider these properties useful for models as well for two reasons.

At the design state of a formalism it helps if these properties, when desired for a proof structure, are already present in the model—both to design proof rules as well as to verify their correctness.
When justifying a semantics given in the form of axioms, it is necessary to prove that it is consistent, i.e. that there is a model. In principle it is enough to give a model in which each formula gets a value. However, it is even better, and also standard practice, to represent operators in the model. In this case, the operators are the constructors. Just as compositional (denotational) models are used to clarify compositional proof systems, so are modular ones to clarify modular systems.

Also, when the formalism is used to develop programs it appears in practice that the more similarities there are between behaviour structure and specification structure, the easier it is to write correct specifications. In the same way, the more similarities there are between the structure of behaviour structure, specification structure and proof structure (in the case of compositionality), or between specification and proof structure (in the case of modularity), the easier it is to write correct specifications for composition.

3.3.2.3. *Interpretations of the concepts compositionality and modularity*

Apart from rather narrow definitions that depend on restrictions specific to the particular framework at hand, there appear to be at least three interpretations in the literature. Most of the time, even when only compositionality is claimed, in fact modularity is achieved.

(i) The specification language has rules for different constructors. For example parallel and sequential execution, guarded commands with nondeterministic or in some respect fair choice.

    Examples of such a formalism are [BKP84, 85].

(ii) The specification language has only one constructor rule; in practice this means, a rule for parallel combination. This makes quite a lot of difference in the use of a specification method, as we will show in the following examples. A further consequence is that the specification method can only deal with programming languages that have no other constructs than the parallel constructor, unless extra rules for the last development step (of the specification into a program) are added.

An example of such a formalism is [NDGO86].

(iii) The specification language has, one or more, constructor rules, but the subspecifications describe subcomponents only as subcomponents of the larger component that they belong to rather than independently. In the rules this shows in the form of what has to be checked about subcomponents. The premises of the rule to derive $\vdash C(O_1, ..., O_n)\,\mathrm{SAT}\,S$ that were of the form

$$\vdash P_i\,\mathrm{SAT}\,S_i\ , \quad i \in I\ ,$$

are replaced by

$$\vdash P\,\mathrm{SAT}\,S_i\ , \quad i \in I\ .$$

This means, that it can only be decided that a subcomponent satisfies the requirements of its subspecification when the whole component is available. The application of such a formalism seems to be mostly to structure and ease correctness proofs and guide the design in an intuitive way. An example of such a formalism is [Sta85], where liveness properties give rise to this form of compositionality.

Example.
A very simple example showing the difference between the use for formalisms like (i) and (ii) is the parallel or sequential combination of two assignments, say $O_1 :: x := 1$ and $O_2 :: x := 2$.

Obvious, informal, temporal specifications are

$$S_1 ::= \text{inactive}\,O_1\ \mathcal{U}\ (\text{active}\,O_1 \ \wedge\ O(x = 1 \ \wedge\ \Box\,\text{inactive}\,O_1))$$

respectively

$$S_2 ::= \text{inactive}\,O_2\ \mathcal{U}\ (\text{active}\,O_2 \ \wedge\ O(x = 2 \ \wedge\ \Box\,\text{inactive}\,O_2))\ .$$

These subspecifications are, intuitively, all that should be told about the subcomponents.

In a formalism as mentioned under (i), there would be two different rules for parallel and sequential composition. As can be read in, e.g., [BKP84], the difference between the two rules is mainly, that the premisse of the parallel rule would be of the form

$$(S_1 \ \wedge \ S_2 \ \wedge \ \Box(\text{not}(\text{active}\, O_1 \ \wedge \ \text{active}\, O_2))) \rightarrow S \ .$$

For the sequential rule this would require additionally

$$(S_1 \ \wedge \ S_2 \ \wedge \ \Box((\text{not}(\text{active}\, O_2)\ \text{Unless}\ \text{active}\, O_1))) \rightarrow S \ .$$

In the formalisms as mentioned under (ii), the only way to mimmick sequential composition via the first rule—the only available one—is to include the necessary information in the subspecifications. For instance as follows.

$$S_1' \ ::= \ \text{inactive}\, O_1 \ U \ (\text{active}\, O_1 \ \wedge \ O(x = 1 \ \wedge \ \Box\, \text{inactive}\, O_1))$$

and

$$S_2' \ ::= \text{inactive}\, O_2 \ U \ ((\text{active}\, O_1 \ \wedge \ \text{inactive}\, O_2) \ U \ (\text{active}\, O_2 \ \wedge \ O(x = 2 \ \wedge \ \Box\, \text{inactive}\, O_1))) \ .$$

respectively.

Remarks.

1. In principle it is possible to transform approach (ii) into approach (i) through using the general rule to find rules for specific constructors. This, however, is not much less difficult than designing such rules from scratch, as one has to separate the information about a component from the information about constructors. Both of these have to be formulated in a general way, as several kinds of subcomponents may be combined via, also, several kinds of constructors.
2. To avoid confusion, we draw attention to the fact that in (i) and (ii) it is only information about the specific environment in which the subcomponent is placed that is absent. It is not at all forbidden to refer to a subcomponents potential environment. Examples of this are again [NDGO86] and [BKP84, 85].

3.4. Further levels?

All notions thus far have been defined at quite abstract levels. The next level would, in our opinion, consider, in abstracto, languages, with or without structure, with some semantics, both to write specifications in and for programs, and a satisfaction relation with, presumably, some special properties.

This level of abstraction is closer to the world of real descriptions formalisms than the ones before. Surprisingly enough, it appears that no further relevant notions can be defined than at the previous levels.

It seems that a much greater increase in concreteness is called for to enable description of further notions. Most of these are then depending on the formalism at hand to such an extent that attempts at abstract definitions are not likely to be fruitful.

We therefore claim that the two levels considered in this paper suffice to describe general notions.

4. REACTIVITY AND BIAS FREEDOM

There are two concepts, reactivity and bias freedom, that have not been addressed in either Section 2 or Section 3. The reason is, that they need for their assessment a combination of the considerations in these sections.

4.1. Reactivity

In this section we use the framework developed thus far to investigate the concept of reactive system as introduced by Harel and Pnueli in [HP85].

The point made in [HP85] is, that there are two views of systems.

The first one is the transformational one, regarding programs as functions, or, in the nondeterministic case, relations between inputs and outputs. Systems that maintain an ongoing interactive relationship with their environment, however, are hard to describe in this manner. Even a relation between a sequence of input events and a sequence of output events may not suffice, for example because the effect of an input may depend on the output thus far. Such systems are called reactive.

Harel and Pnueli point out carefully, that certain systems are more amenable to transformational description while other ones allow reactive description, rather than stating that these terms classify systems.

The reason to do so is, that every system can be viewed as transformational and most of them also as reactive. When observed in enough detail, a system bases its next move on the current state, thus being transformational. Conversely, as in the example about sequences mentioned above, as soon as some order information is left out, transformational description might fail.

Clearly, the level of abstraction at which the system is viewed is crucial. Every system surely becomes transformational at the machine state level. It is less clear though, when exactly the view of a system becomes reactive. We investigate and define this dependency on the level of abstractness more formally.

To decide the reactivity of a system the following notions are required, that can be seen as extensions to the features of behaviour considered in Section 2. Behaviour is split into input and output behaviour at the level of executions.

We define, what kind of extra notions are required, rather than giving concrete ones. In this way, the definition becomes parameterised by these notions but remains generally applicable.

(i) A notion to compare inputs using some notion of order-in-time relationship between input and output.

For example, let ... $< i, o >< i', o' >$ ... denote the change over time of input and output. Then, in an obvious interpretation, input ab differs with respect to timing in

$$< a, \epsilon >< a, a >< ab, a >< ab, ab >$$

and

$$< a, \epsilon >< ab, \epsilon >< ab, a >< ab, ab >$$

but also in

$$< a, \epsilon >< ab, \epsilon >< ab, b >< ab, ba > \ .$$

(ii) A notion to compare inputs with less precise time relationship to outputs.

For example, one could consider the input sequence belonging to a complete sequence of i/o pairs as input. $< ab, * >$ and $< ab, + >$ with $*$ and $+$ arbitrary would then be considered to be the same as far as input is concerned.

(iii) A notion to compare outputs—usually, but not necessarily, without a time relation to inputs.

Again, output sequences of complete sequences of i/o pairs could be chosen to represent output. $< *, ab >$ and $< +, ab >$ would then be considered to be the same as far as input is concerned.

Definition 4.1.
A system is reactive with respect to some notions (i), (ii) and (iii) of observation like introduced above if
there are two inputs that

differ according to (i), but do not differ according to (ii)

and lead to outputs that

differ according to (iii).

Remark.
When the level of abstraction of observation is given, a system is either reactive or transformational.

4.2. Bias freedom

The concept bias freedom intuitively concerns the question in how far a specification or a behaviour avoids suggesting undesired detail about the implementation. Much of this ground has already been covered by abstractness notions. Up till now, however, the only assessment of information content has been in how far descriptions distinguish programs, i.e., as to how much information is present. Consider the following notion of bias freedom, introduced by Jones in [Jo86].

Definition 4.2.
A description of behaviour is bias free if
if in the description two states are different
then this difference can be observed via the application of (a sequence of) observation operators.

We would like to paraphrase this definition to arrive at a slightly more general notion:

A semantical object representing behaviour is bias free if
if two states differ
then this is observable.

This of course is an informal notion. We investigate which extra notions are required to provide a formalisation.

Firstly, a notion of state needs to be provided. This belongs to the investigations in Section 2. Here again a justification as demanded of the notions there is required.

Secondly, a notion of observation needs to be provided. Note, that this in general will be a much more powerful notion than the one introduced in Section 2. There, observations were restricted to momentaneous ones. Here, on the contrary, the idea is that any way in which differences can be observed is admissable. We propose one intuitive restriction: the observations should only depend on future evolution of the behaviour from the states that are compared.

Assuming that formal definitions of these notions are provided, the definition is then formalised.

5. COMPARISON OF SPECIFICATION SYSTEMS

In this section we use some of the insights obtained in previous sections to discuss the aims and solutions presented in [BKP84, 85, 86] and [Ku87]. We also attempt to compare our solutions to the part of the work by Lamport (cf. [La83a, La83b, La85, AL88, La89]) that adresses similar topics and has been an inspiration in many ways.

More incidental remarks are made about other approaches, notably those by Nguyen et al [NDGO86] and Stark [Sta85, 88].

We make the implicit assumption that temporal logic is a good tool to describe behaviour, providing a useful level of abstraction to reason about time. More thorough going defenses of this claim can be found in, e.g., [La83b] and [Pn86].

As it turns out that the aims we can formulate both coincide well with the papers we wish to introduce as well as involve quite a lot of papers where the other approaches are concerned it seems most useful to structure the discussion according to the aims.

Some more specific remarks about [BKP84, 85, 86] and [Ku87] are made after this.

5.1. General aims and problems

The following aims can be distinguished.

AIM 1. PARALLEL COMPOSITION

There are two obvious changes in representation when going from sequential to parallel systems.

(i) Points where interference can occur need to be characterised.

(ii) For every interface state change it needs to be modelled which component performed the change.

Lamport presents two contrasting approaches here, namely [La83] and [La85].

In [La83] the basic idea is that actions are labelled by the name of the performer. Subcomponents are combined via requirements about these actions. The advantage here is abstractness: only names of subcomponents are added to the observables.

In [La85] the basic idea is, that possible interference points are characterised via program locations. Subcomponents are then combined via aliasing relations between their names. The advantage here is independence of future choice of combinators. The descriptions of subcomponents can, because of the powerful description mechanism using locations, be independent of the future environment that the subcomponent has to function in. All the information with regard to combining subcomponents can be put into the combinator using aliasing.

In [BKP84] we are quite close to the approach in [La83], but push the abstractness of the observable state a little further by using a component/environment view introduced by Jones [Jo81] and Aczel [Ac83]. Recent work by Stirling [Sti88] suggests that this is precisely the "right" amount of additional information to achieve compositionality and

modularity. Further details are given in the discussion [BKP84, 85].

In [NDGO86] a very elegant rule for parallel combination of networks of processes is presented. It comes closest to just intersecting subspecifications of all the rules we know. Also the temporal logic used is a simple linear one, although it contains the next operator.

To achieve these desirable features, a rather strong notion of observation is used. The main point is, that observations on a port contain the history of all communications via that port thus far. We feel that this information should really be part of the structure of behaviour rather than of an observation, but again, this is a matter of balancing concerns.

All formalisms discussed, with the exception of [BKP86] and [Ku87] where this was not an issue, are modular—in retrospect, it is not so much this property as well as the way in which it is achieved that is important.

AIM 2. EXPRESSIVENESS AND ABSTRACTNESS

Bearing in mind the justification obligation with respect to the intuition as discussed in Section 2, we aim for the following modelling ideal.

(i) Observations should be represented by interface states that are valuations of interface variables.

(ii) Executions should be represented by sequences of interface states, where interface state changes correspond to changes in interface variables. On the one hand this implies that any two interface variable changes at different moments in time (to be defined with respect to the intuition!) give rise to different interface states. On the other hand this implies that no two identical interface states occur consecutively— stuttering is not represented ([La83b]).

(iii) Specifications are sets of interface state sequences. If a set contains more than one sequence, this represents that the specification allows each of these sequences as execution. Consequently, the satisfaction relation should be set inclusion.

ACHIEVING AIM 2

Unfortunately, differently from the first aim, it proves to be quite difficult to use temporal logic to achieve the second aim. We evaluate the problem.

Problem with Aim 2

There are several reasons to deviate from the ideals described in Aim 2. We discuss the one that seems to be a concern to all that use some form of temporal logic as the language to write specifications in.

In reality, every next move of a system depends on the current machine state only; this holds even true for liveness and fairness properties.

This is no longer the case for interface states, where only interface variable values are present. The choice of interface variables can be made on various grounds, for example

that they are shared between component and its environment. It is usually not at all the case that they are chosen because they represent enough information to decide the possible move to the next interface state.

The sequence structure is used to compensate for this loss of information.
For liveness and fairness properties this leads to requirements about the sequences that are easy to formulate in temporal logic. However, to express other structure, for instance caused by recursion and hiding, temporal logic is less adequate.

Solutions to the problem

Accepting that temporal operators are not powerful enough to replace certain internal state information we discuss two solutions, each causing its own complications. We emphasize, that only sequence structure with regard to interface variables is desired.

1. Incorporate the information in the state via extra variables in such a manner that these extra variables do not in themselves add more information. The idea is, that they only serve to express structure requirements regarding the visible ones.

   This is the existentially quantified local logical variable approach, as used for example in [BKP84, 85, 86].
   The idea is comparable to the use of y or the first order formula

   $$\exists y \cdot x = y \wedge z = y, \text{ which is equivalent to } x = z .$$

   The term "local" refers to the fact that these variables are allowed to change over time, in contrast to global logical variables, or freeze variables, that remain the same during an execution.

2. Allow more internal information in the state via internal variables and change the satisfaction relation to ignore the internal state information itself. Lamport introduced such a solution using state functions to equalise internal state information.

Problem with Solution 1

If temporal logic over a discrete domain is used, the only reasonable way to abstract away from stuttering seems to be, as Lamport observed [La83b], to allow all finite repetitions of identified states. In the logic one tries to achieve this through forbidding the O operator.
The problem is, that existential quantification introduces the possibility to make stuttering matter. Consider the following example, using the until operator U.

$$S_1 \;::\; x = 0 \wedge (x = 0 \; U \; x = 1)$$

$$S_2 \;::\; \exists i \cdot (x = 0 \wedge i = 0) \wedge ((x = 0 \wedge i = 0) \; U \; (x = 0 \wedge i = 1)) \; U \; (x = 1) \; .$$

If i really did not matter, S_1 and S_2 should implement each other, i.e. $S_1 \rightarrow S_2$.
Unfortunately, $S_1 \not\rightarrow S_2$, as S_1 allows $x = 0 \wedge Ox = 1$ and S_2 does not.

The remedy proposed in [BKP86] is to use dense domains, more detail is given in the discussion of that paper. A similar solution is presented in [St88].

Problem with Solution 2

Instead of trying to quantify internal variables away, Lamport in [La83b, 89] advocates, in principle, to reinterpret the lower level states in terms of the higher level ones.
Formally: $S_2 \text{ SAT } S_1$ iff
there is for each x at level 1 a state function F_x
such that $S_2 \rightarrow S_1[F_x/x]$.

The problem here is, that this requires more difference between states at the lower level than at the higher ([HW87]). As the last example showed, this needs not be the case.

The remedy of going to a dense domain does not work here, as the sets of variables are fixed.

In [AL88] a solution is presented, based on adding extra variables at the lower level to enforce enough different low level states.

Remark.
It is interesting to notice, that explicit addition of variables at the lower level solves the problem for the discrete time domain, because it rules out the undesired case of less states at the lower level than at the higher one. It is clear that this cannot be achieved with existential quantification as one cannot control where the extra states will occur: at the desired (low) level or ar the undesired (high) level.

Conversely, the transition to a dense domain does not help in the state function based approach, as the variables to be connected are explicitly mentioned. Whether they are available and have the correct values is obviously independent of the time domain.

Remark.
A structured temporal semantics is much more difficult to express in temporal logic in the "ideal" manner than a specification. The reason is, that the semantics has to "automatically" construct a formula, whereas in specification is "hand tuned".

Take for example hiding.
The semantics of begin new x; $x := y$; $x := x + 1$; $y := x$ end

It seems to leave few other options than something like $\exists x.\ Ox = y \wedge O(Ox = x + 1 \wedge O(Oy = x))$. A specification on the other hand would look like $Oy = y + 1$.

AIM 3. REPRESENTATION OF CHOICE IN LINEAR TIME TEMPORAL LOGIC

This aim is of less fundamental nature than the other two. The idea was to see how far linear temporal logic could be pushed to represent choice without having to extend the interface to represent choice options. This is the subject of [Ku87].

5.3. Discussion of [BKP84, 85, 86] and [Ku87]

5.2.1. [BKP84]

In this paper, a behaviour, specification and proof structure are presented for a shared variable language with recursion. As it also forms the basis for [BKP85], we discuss the contents in some detail.

The main aim was to provide a compositional specification structure using temporal logic—it even turned out to be modular in the strongest sense as defined in Section 3.

The main ideas in the paper are the following.

(i) Modularity

(a) At the time of writing this paper, it was not universally understood whether temporal logic is endogeneous or exogeneous, i.e., whether formulae always apply to one global program or not. It was later realised, that temporal formulae as such are endogeneous but can be coupled to programs by just adding this extra information to the formal language. This is the same idea as the extension of first order formulae to Hoare formulae.

We use the notation $\{S\}\,\phi$ to indicate formally that a program S satisfies a (temporal) formula ϕ. The link with Hoare's notation, $\{p\}\,S\,\{q\}$ is, that temporal formulae have an inbuild notion of time, so that no extra indication of when to evaluate them—like p before, q after execution—is necessary. $\{p\}\,S\,\{q\}$ can be written as a formula $\phi = (\text{at}\,S\ \&\ p) \rightarrow \Box(\text{after}\,S \rightarrow q)$.

This idea is, of course, not sufficient to obtain modularity or even compositionality. As pointed out in Section 3, information content and the form in which the information is defined determine that property. However, the coupling of formulae to components is a crucial prerequisite.

It appears that, as mentioned in [La85], Abrahamson was the first to use temporal logic in this manner (cf. [Ab80]). Lamport couples formulae to programs already, in a rather informal way, in [La83], but as far as we know did not incorporate this in a formal language. This informal coupling is still used quite often.

(b) To obtain modularity, there are two rather different approaches available. A component can be viewed in isolation; enough information to construct its behaviour when combined with other components should then be provided. The other option is, to give the behaviour of a component with respect to some general environment, including as much information about this environment as can without loss of generality of abstractness be defined.

To achieve this, we follow Jones's ([Jo81]) component/environment view of concurrent shared variable computations. The idea is, to model the behaviour of a component as sequences of states that record the values of its observable variables in all possible environments, from the viewpoint of that component. We then use the model provided by Aczel ([Ac83]) for this approach. The important feature of this model is that it incorporates the "viewpoint" idea by including all possible changes in the sequences, but adding to all transitions labels indicating who caused the change. To not mention more about the

environment than is allowable with respect to both abstractness and modularity, label information is restricted to two possibilities: component action or environment action.

(c) To give the behaviour structure as well as the rules in the proof structure we use label predicates as parameters in the temporal formulae (cf. [BK83]). These parameters are then manipulated to reflect the change in viewpoint that constitutes the combination of subcomponents into a larger component.

We feel that in retrospect the main contribution of [BKP84] lays in the way that Aczel's model has been formalised using temporal logic.

Although we were mainly inspired by this model, the idea of describing components by temporal formulae that also contain information about the environment was already present in [La83]. There, however, environment behaviour was not grouped together under environment action predicates nor were these used as parameters.

To some extent unwittingly, we formulated the rules in such a manner that the objective of simple combination as mentioned above was served. When as much general information about the environment that is available is incorporated in the definition of a subcomponent, combination rules can be simple.

For example, the fact that the parallel combinator is fair with respect to progress of subcomponents could be expressed by adding a fairness requirement in the definitions and rules for parallel combination. However, in [BKP84], this is already incorporated in the description of the subcomponents by only allowing finitely many environment steps before, e.g., an assignment statement is executed. The price to be paid for this simplicity, as Lamport observed in [La85], is that making changes to the language, e.g., adding an unfair cobegin, may necessitate changes to the definitions of other components and combinators. This is, however, not so much an inherent feature of this approach; one could equally well leave the fairness requirement out at the linear subcomponent stage and include it in the parallel combinator.

Another instance of aiming for simplification is the introduction of the chop operator to handle sequential composition and iteration. As explained in [BKP84], this is not necessary—careful use of labels can achieve the same result.

(ii) Abstractness

(a) As to observations. Existential quantification is used to describe hiding of variables in the definition of the behaviour structure.

Although it is nice that it is possible to be abstract, it would be wrong to write specifications that are de jure bias free and abstract but not de facto: in terms of suggesting implementations it does not make much difference whether or not variables are quantified away.

Luckily, recursion forced us to extend the logic with a fixed point operator with respect to the implication ordering on formulae. This operator enables to avoid use of quantification in most cases, at least in specifications. A topic of further research is to find syntactical constraints on the use of quantified variables.

(b) As to executions. Abstractness with respect to stuttering is present to only a limited extent. Environment activity is generally handled in an abstract manner by avoiding the O operator. This operator, however, is used to indicate the moment of change through component activity. This is not desirable, both because of loss of abstractness and because further refinement now requires action refinement of one action into more. In [BKP86] we investigate this problem further.

5.2.2. [BKP85]

At the end of [BKP84] an attempt was made to cover synchronised communication. This was not very satisfactory—for instance, associativity did not hold. Here a more thorough assessment of this problem is given. The main purpose of this paper is to show the following:

(i) In the standard linear time logic approach alternative communication options can, but also need to be represented in the observables, e.g., as ready sets.

(ii) The same problems with respect to abstractness as occur with these sets in the traditional safety models (cf. [Ho85], [He88]) occur again with essentially the same solutions—we choose saturation of readiness sets.

In the paper a rather non-abstract use is made of these sets as they are added to each observation. The motivation is, that this eases the formulation of different fairness assumptions. Later it became clear to us, that full abstractness of a semantics is in fact relative to fairness assumptions. Of course, this is to be expected as behaviour in contexts depends on these assumptions.

5.2.3. [BKP86]

As mentioned before, the framework in [BKP84], and hence similarly in [BKP85], is not abstract with respect to, essentially, stepcounting.

A solution is presented, changing from discrete time domains to dense ones. The reals are used, but the treatment goes through for any dense domain.

Referring to the example in 5.1, the problem was that some intuitively equivalent specifications put different lower bounds on the number of states between two particular states.

The essence of the solution is, that in dense time domains, as soon as there are two timepoints, there are also infinitely many inbetween, so no lower bound is possible and the problem disappears.

An important feature is, that because the domain is dense, an additional notion of finite variability is necessary to exclude the possibility of infinitely many computation steps in finite time. The idea is, that this can be avoided by demanding that all variables, be they program or logical ones, do no change infinitely often in finite time.

The treatment does not make the extension to modular formalisms as in the previous papers. In principle this seems to be possible; it is the subject of further research.

A similar model can be found in [Sta88]. There some extension towards a compositional formalism is made, but the proof rule provided is not complete. In our opinion a complete proof rule for this model could be given using either the approach [AL88] or quantification over local logical variables. Further investigations into abstract semantics given by temporal logics can be found in [Fi87].

There are two puzzling things left:

(i) The use of a dense domain is unsatisfactory in the following sense. As explained in [BKP86], dense time allows infinitely many state changes in finite time. This is incorrect for the discrete value domains that we describe. Therefore an additional restriction, finite variability has to be imposed on the behaviour of variables.

    This is unfortunate: the time domain has properties that one does not want to use to the full. It would be more satisfactory, if the time domain could be chosen as to not enable one to express such properties.

    The question is, is there a time domain that, combined with existential quantification gives the right expressive power without extra restrictions?

(ii) Intuitively, one would expect that a concrete value domain would fit well with a concrete time domain. Namely, changes are what determines timepoints, and changes are discrete.

    The situation seems not to be this simple, as the problems with discrete domains show.

One answer might be found in the direction that the existential quantification over infinitely many variables somehow destroys the discreteness of the domain.

Alternatively, maybe the conclusion form these remarks has to be that a discrete domain plus state functions as proposed in [AL88] is the natural framework.

5.2.4. [Ku87]

In the discussion of [BKP85] it was mentioned that choice alternatives were represented via readiness-sets. As mentioned in Section 2, we do not consider this a very well justifiable complication of observations.

A better alternative would, in our opinion, be to incorporate the choice information in the structure of the behaviour, e.g., as branching structure. The obvious solution would be to use branching time temporal logic. We consider this the best approach, but it is very rigorous in that the logic framework changes completely. This in fact is the subject of current investigations.

In [Ku87] we show, that linear time temporal logic can be used, pressing it to the limits of its expressiveness, to describe a less complicated form of alternatives: required nondeterministic choice (cf. [Fr77]). The idea is, that is should be possible in specifications to state that if some alternatives are described in the specification, then it is not allowed to implement only a subset of these.

To achieve this, we change the notion of specification and satisfaction rather than the

logic. Intuitively speaking, rather than only specifying one set of sequences as an upper bound to the allowed ones, we also specify a lower bound.

This gives rise to a problem with development: Lower bounds do not allow any non-determinism for not yet mentioned variables. This problem is solved via a judicious use of hiding. The specification language thus obtained is clearly more expressive than the one from [BKP84]. It is possible to add the abstraction made in [BKP86] to this formalism, as this is a completely orthogonal change.

ACKNOWLEDGEMENTS

This work has much benefitted from the interaction with members of the TEMPLE group at Manchester University, members of the Theoretical Computer Science group at Eindhoven University of Technology and students at Eindhoven University of Technology. More in particular, the formalisation of the notion of reactive system was developed together with Kees Huizing; Eduard Diepstraten was instrumental in understanding the role of state function and existential quantification over logical variables and Antonio Cau provided much of the information about [Sta85]. I thank Anita Klooster for her excellent typing.

REFERENCES

[AL88] Abadi, M., Lamport, L., The Existence of Refinement Mappings, 3rd IEEE-Symposium on Logic in Computer Science, pp. 165–175, 1988.

[Ab80] Abrahamson, K.M., Decidability and Expressiveness of Logics of processes, Ph.D. Thesis, issued as Technical Report No. 80-08-01, Dept. of Comp. Sci., Univ. of Washington, 1980.

[Ac83] Aczel, P., Semantics for a proof rule by C.B. Jones, Unpublished note, University of Manchester, 1983.

[Ap81] Apt, K.R., Ten years of Hoare's logic: a survey Part I, ACM TOPLAS 3, pp. 431–483, 1981.

[BK83] Barringer, H., Kuiper, R., Towards the Hierarchical Temporal Logic Specification of Concurrent Systems, The Analysis of Concurent Systems, Cambridge, (B.T. Denvir et al eds.), LNCS 207, pp. 157–184, 1984.

[BKP84] Barringer, H., Kuiper, R., Pnueli, A., Now You May Compose Temporal Logic Specifications, 16th ACM Symposium on the Theory of Computing, pp. 51–63, 1984.

[BKP85] Barringer, H., Kuiper, R., Pnueli, A., A Compositional Temporal Approach to a CSP-like Language, IFIP Formal Models in Programming, (E.J. Neuhold, G. Chroust eds.), Elsevier, pp. 207–227, 1985.

[BKP86] Barringer,H., Kuiper, R., Pnueli, A., A Really Abstract Concurrent Model and its Temporal Logic, 13th ACM-Symposium on Principles of Programming Languages, pp. 173–183, 1986.

[Da80] Dalen, D. van, Logic and Structure, Springer, 1980.

[Fi87] Fisher, M.D., Temporal Logics for Abstract Semantics, Ph. D. Thesis, Univ. of Manchester, 1987, also UNCS-87-12-4.

[Fr77] Francez, N., A Case for a Forward Predicate Transformer, Inf. Proc. Letters IEEE 6:6, 1977.

[HGR87] Huizing, C., Gerth, R., Roever, W.P. de, Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language, 14th ACM-Symposium on Principles of Programming Languages, pp. 223–238, 1987.

[HP85] Harel, D., Pnueli, A., On the Development of Reactive Systems, Logic and Models of Concurrent Systems (NATO ASI Series, Vol. 133, K.R. Apt, ed.), Springer, 1985, pp. 447–498.

[Ha87] Harel, D., Statecharts: A Visual Approach to Complex Systems, Science of Computer Programming, Vol. 8-3, pp. 231–274, 1987.

[He88] Hennessy, M.S., Algebraic Theory of Processes, MIT Press, 1988.

[HW87] Herlihy, M.P., Wing, J.M., Axioms for concurrent objects, 14th ACM-Symposium on Principles of Programming Languages, pp. 13–26, 1987.

[Ho85] Hoare, C.A.R., Communicating Sequential Processes, Prentice-Hall, 1985.

[HR86] Hooman, J., Roever, W.P. de, The Quest goes on: a survey of proof systems for partial correctness of CSP, Current Trends in Concurrency, LNCS 224, pp. 343–395, 1986.

[Jo81] Jones, C.B., Development Methods for Computer Programs Including a Notion of Interference, D. Phil. Thesis, Oxford, Tech. Monograph PRG-25, June 1981.

[Jo86] Jones, C.B., Systematic Software Development Using VDM, Prentice-Hall, 1986.

[Ku87] Kuiper, R., Enforcing Nondeterminism via Linear Time Temporal Logic Specifications using Hiding, Proc. of the Colloquium on Temporal Logic and Specification, Altrincham, 1987, (H. Barringer, B. Banieqbal, A. Pnueli eds.). To appear in LNCS, 1989.

[La83a] Lamport, L., Specifying Concurrent Program Modules, ACM Transactions on Prog. Logic and Sys. 5.2, pp. 190–222, 1983.

[La83b] Lamport, L., What Good is Temporal Logic? Information Processing 83, (R.E. Mason ed.), Elsevier, pp. 657–668, 1983.

[La85] Lamport, L., An Axiomatic Semantics of Concurrent Programming Languages, NATO ASI Series, Vol. F13, Logics and Models of Concurrent Systems, (K.R. Apt ed.), Springer, pp. 77–122, 1985.

[La89] Lamport, L., A Simple Approach to Specifying Concurrent Systems, Comm. ACM, Vol. 32, No. 1, pp. 32–45, 1989.

[NDGO86] Nguyen, V., Demers, A., Gries, D., Owicki, S., A model and temporal proof system for networks of processes, Distributed Computing 1, 1986, pp. 7–25.

[OD82] O'Donnell, M.J., A Critique of the Foundation of Hoare-style Programming Logics, Proc. of the Conference on Logics of Programs, Yorktown Heights (D. Kozen, ed.), LNCS 131, pp. 349–374, 1982.

[Pn86] Pnueli, A., Specification and development of reactive systems, Information Processing 86, (H.-J. Kugler ed.), North-Holland, pp. 845–858, 1986.

[Pl83] Plotkin, G., An Operational Semantics for CSP, IFIP Conference on the Formal Description of Programming Concepts II, pp. 199–225, Elsevier, 1983.

[dR85] Roever, W.P. de, The Quest for Compositionality—a survey of assertion-based proof systems for concurrent programs, Part 1: Concurrency based on shared variables, IFIP Formal Models in Programming, (E.J. Neuhold, G. Chroust eds.), Elsevier, pp. 187–205, 1985.

[Sta85] Stark, E.W., A proof Technique for Rely/Guarantee Properties, 5th Conference on the Foundations of Software Technology and Theoretical Computer Science, (S.N. Maheshwari ed.), LNCS 206, pp. 369–391, 1985.

[Sta88] Stark, E.W., Proving Entailment between Conceptual State Specifications, Theoretical Computer Science 56, pp. 135–154, 1988.

[Sti88] Stirling, C., A generalization of Owicki-Gries's Hoare logic for a concurrent while language, Theoretical Computer Science 85, pp. 347–359, 1988.

[Wi87] Widom, J., Trace-based network proof systems: Expressiveness and Completeness, Ph.D. Thesis 87-833, Cornell University, Ithaca, New York, 1987.

[WGS87] Widom, J., Gries, D., Schneider, F.B., Completeness and incompleteness of trace-based network proof systems, 14th ACM-Symposium on Principles of Languages, pp. 27–38, 1987.

[ZBR83] Zwiers, J., Bruin, A. de, Roever, W.P. de, A proof system for partial correctness of Dynamic Networks of Processes, Proc. of the Conference on Logics of Programs 1983, LNCS 164, pp. 513–527, 1984.

[Zw88] Zwiers, J., Compositionality, Concurrency and Partial Correctness: Proof theories for networks of processes, and their connection, Ph.D. Thesis, Eindhoven University of Technology, 1988; also as LNCS 321, 1989.

Now You May Compose Temporal Logic Specifications

by

Howard Barringer*, Ruurd Kuiper*

and Amir Pnueli†

Abstract

A compositional temporal logic proof system for the specification and verification of concurrent programs is presented. Versions of the system are developed for shared variables and communication based programming languages that include procedures.

1. Introduction

When we review the state of the art in formal specification and verification of concurrent programs we observe the following situation.

On one hand the formalism of temporal logic is a powerful tool for expressing and verifying a wide spectrum of properties of concurrent programs. Almost any reasonable property one would like to state and prove about a program can be handled in TL (temporal logic). See [MP1] for a partial glimpse of the variety of properties that can be expressed, such as invariance (safety), liveness, precedence, etc.

However, TL has been severely criticized for being global, non-modular and non-compositional. By that we mean that in order to formulate and verify a temporal property we must have before us the complete program. The temporal formula reasons about the *global* state of the program that includes the values of all the variables as well as the control

---

*University of Manchester, Manchester England.

†Weizmann Institute of Science, Rehovot, Israel.

locii in all the concurrent modules. There seemed to be no natural way by which temporal specifications that were derived separately for programs P_1 and P_2 could be combined into a temporal specification for the parallel composition $P_1 \parallel P_2$. This deficiency was already indicated in [Pn] where the TL system is described as being ***endogenous***, i.e. assuming the complete program as fixed context.

Consequently, while the temporal language provides a most useful global specification tool, and the temporal system provided rigorous verification of existing programs, they offered very little support for the most important activity—that of rigorous systematic ***development*** of concurrent systems. The current approach in verification systems for sequential programs evaluates proposed formal systems almost solely on the extent of support they give to the activity of modular development. (See for example [J]).

Spurred by the great success of sequential formal systems such as [H1], [D] to provide such a developmental methodology, there has been a continuous effort to derive a similar modular and compositional system for concurrent programs. The first such system was presented in [OG] and provided a methodology for proving partial correctness of the shared variable model of concurrent systems. A considerable time later, a syntax-directed proof system for partial correctness of communication-based concurrent programs was presented in [AFR]. Since then, additional efforts were made to extend the scope of compositional proof systems to more than just the class of invariance (safety) properties. Among these efforts we should count [L1], [H2] and [MSC]. However, in all cases, the class of properties that can be proved by the suggested compositional proof system is restricted.

In this paper we present a compositional system based on TL. The system will be presented both for the shared variables and the communication-based models of concurrent programs. In a modest sense this may be viewed as obtaining the best of two worlds, having both a powerful and expressive language that can express numerous interesting program properties, and a compositional methodology that is conducive to modular specification and verification, and most importantly, supports systematic development of concurrent programs.

2. Temporal Logic Over Labelled Sequences

In this section we introduce the language of temporal logic that we propose as the vehicle for expressing program specifications. Since the emphasize in this paper is on the feasibility of compositional temporal systems we have adopted a very powerful temporal language and obtain as a result simple and intuitive proof rules. The price that is to be paid is the high complexity of the decision and proof procedures for the powerful language. In a later section we will indicate how the language can be restricted at the expense of more complicated proof rules.

For our maximal temporal language we adopt constructs taken from several sources, including [MP1], [HKP], [W] and [HMM].

The basic alphabet of the logic language includes *local symbols* of the following types:

$P_1, P_2, \cdots$—State Propositions,
$E_1, E_2, \cdots$—Edge Propositions,
$y_1, y_2, \cdots$—Local individual variables.

It also includes as *global symbols:*

x_1, x_2—Global individual variables.

A temporal formula is built out of *atomic formulas* which are propositions and predicates applied to terms that are constructed out of individual variables. To these we may successively apply the boolean connectives, quantification over global individual variables and the temporal operators in any order consistent with their arity. The temporal operators that we use include the following unary operators:

$$\bigcirc \text{(next)}, \ \Box \text{ (always), and } \diamond \text{ (sometimes)}$$

and the following binary operators:

$$\mathcal{U} \text{ (until)}, \ \mathcal{C} \text{ (chop or combine)}, \ \mathcal{C}^* \text{ (iterated combine)}.$$

For simplicity we assume a fixed domain D over which the individual variables range, and a fixed interpretation for the predicate and function symbols over the domain.

A model over this fixed interpretation consists of:

a) A global assignment α to the global individual variables, assigning a D-value to each variable.

b) A nonempty sequence

$$\sigma : s_0 \underset{t_0}{\rightarrow} s_1 \underset{t_1}{\rightarrow} s_2 \rightarrow \cdots$$

consisting of named states $s_0, s_1, s_2, \cdots$ and named transitions $t_0, t_1, t_2, \cdots$ connecting them. We allow both finite and infinite sequences in our models.

c) A local state interpretation I, assigning to each state $s_i \in \sigma$ and each state proposition P a truth value $I(s_i, P) \in \{T, F\}$ and to each state s_i and local individual variable y a D-value $I(s_i, y) \in D$.

d) A local edge interpretation J, assigning to each transition t_i and each edge proposition E a truth value $J(t_i, E) \in \{T, F\}$.

For a finite sequence $\sigma : s_0 \underset{t_0}{\rightarrow} \cdots \rightarrow s_k$ we define the length of σ to be $\ell(\sigma) = k$. For an infinite sequence σ, $\ell(\sigma) = \omega$ the first infinite ordinal.

Given two sequences σ_1 and σ_2 we define their concatenation (*fusion* in the terminology of [HKP]) $\sigma_1 \circ \sigma_2$ as follows:
If $\ell(\sigma_1) = \omega$ then $\sigma_1 \circ \sigma_2 = \sigma_1$
If $\sigma_1 = s_0 \underset{t_0}{\rightarrow} s_1 \rightarrow \cdots \rightarrow s_k$ and $\sigma_2 = s_k \underset{t_k}{\rightarrow} s_{k+1} \rightarrow \cdots$
then
$\sigma_1 \circ \sigma_2 = s_0 \underset{t_0}{\rightarrow} s_1 \rightarrow \cdots \rightarrow s_k \underset{t_k}{\rightarrow} s_{k+1} \rightarrow \cdots$

Thus if σ_1 is finite then $\sigma_1 \circ \sigma_2$ is defined only if the last state of σ_1 is identical to the first state of σ_2.

Given a sequence

$$\sigma : s_0 \underset{t_0}{\rightarrow} s_1 \rightarrow \cdots$$

we define σ' as a suffix of σ, denoted by $\sigma' < \sigma$ if there exists a finite sequence σ'', $\ell(\sigma'') > 0$ such that $\sigma = \sigma'' \circ \sigma'$. In the case that $\ell(\sigma'')$ may also equal 0 we write $\sigma' \leq \sigma$.

Given a model $M = (\alpha, \sigma, I, J)$, we define the (truth) value of formulas over the alphabet that M interprets. This definition is given inductively by using some derived models of the form $M' = (\alpha, \sigma', I, J)$. These models differ from M only by the sequence σ'.

Consequently we assume fixed α, I, J and define the value of a formula over a sequence σ, denoted by $\varphi\big|_\sigma$. In the case that we may want to consider different global assignments we will extend the notation to $\varphi\big|_\sigma^\alpha$.

In the following we assume σ to be generically given by

$$\sigma : s_0 \xrightarrow[t_0]{} s_1 \xrightarrow[t_1]{} \cdots$$

First we define the value of terms. For a given σ, these values are determined either by the global assignment α or by the first state of σ. For a global individual variable x

$$x\big|_\sigma = \alpha(x) \in D.$$

For a local individual variable y:

$$y\big|_\sigma = I(s_0, y) \in D.$$

For a function application $f(t_1, \ldots, t_m)$:

$$f(t_1, \ldots, t_m)\big|_\sigma = F_f(t_1\big|_\sigma, \ldots, t_m\big|_\sigma) \in D$$

where we assumed a fixed interpretation $F_f : D^m \to D$ of the proper arity for each function symbol f. For a sequence σ of positive length, we define

$$\bigcirc t\big|_\sigma = t\big|_{s_1 \xrightarrow[t_1]{} s_2 \rightarrow \cdots}$$

Thus, the value of $\bigcirc t$ depends on the *second* state of σ.

Next, we define the valuation of atomic formulas: For a predicate application $p(t_1, \cdots, t_m)$:

$$p(t_1, \ldots, t_m)\big|_\sigma = Q_p(t_1\big|_\sigma, \ldots, t_m\big|_\sigma) \in \{T, F\}$$

where we assumed a fixed interpretatin $Q_p : D^m \to \{T, F\}$ for each predicate symbol p.

For a state proposition P:

$$P\big|_\sigma = I(s_0, P) \in \{T, F\}.$$

For an edge proposition E:

$$E\big|_\sigma = J(t_0, E) \in \{T, F\}.$$

In the case that $\ell(\sigma) = 0$, there is no t_0 transition and then we define $E\big|_\sigma = F$ for every edge proposition E. As is implied by the last two definitions the evaluation of atomic propositions depend only on the first state and transition in σ. According to [HKP] this classifies our logic as *local*.

The valuation of the boolean connectives is obtained by applying the connective to the evaluated subformulas, e.g.

$$(p \vee q)\big|_\sigma = (p\big|_\sigma) \vee (q\big|_\sigma).$$

The valuation of a quantifier over a global variable x:

$\exists\, xp\big|_\sigma^\alpha = T$ *iff* $p\big|_\sigma^{\alpha'} = T$ for some α' that may differ from α only in $\alpha'(x)$.

Next we consider the evaluation of the temporal operators:

For a sequence σ of positive length, $\ell(\sigma) > 0$,

$$\bigcirc p\big|_\sigma = p\big|_{s_1 \xrightarrow{t_1} s_2 \rightarrow \cdots}$$

Note that once we allow finite sequences, there are two types of *next* operators, the existential brand which is the one defined above, and its dual, a universal nexttime operator. The universal nexttime operator can be defined by $\circledcirc p \equiv \sim \bigcirc \sim p$ and would automatically hold true for an arbitrary p and a singleton sequence $\sigma = (s_0)$.

$\Box p\big|_\sigma = T$ *iff* for every suffix sequence $\sigma' \leq \sigma$, $p\big|_{\sigma'} = T$.

$\Diamond p\big|_\sigma = T$ *iff* there exists a suffix $\sigma' \leq \sigma$ such that $p\big|_{\sigma'} = T$.

$p\mathcal{U}q\big|_\sigma = T$ *iff* $\exists \sigma'' \leq \sigma$ such that $q\big|_{\sigma''} = T$ and for every σ', $\sigma'' < \sigma' \leq \sigma$ it follows that $p\big|_{\sigma'} = T$.

Next, we consider the *combine* operator C.

$pCq\big|_{\sigma} = T$ *iff* σ is representable as $\sigma = \sigma' \circ \sigma''$ such that $p\big|_{\sigma'} = q\big|_{\sigma''} = T$.

Note that this is the *weak* version of the combine operator, since in the case that σ is infinite we allow $\sigma = \sigma'$ and then no σ'' is needed.

The iterated combine is defined by:

$pC^*q\big|_{\sigma} = T$ *iff* Either: a) There exist subsequences $\sigma_1, \sigma_2, \ldots, \sigma_k, \sigma_{k+1}$ such that $\sigma = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_k \circ \sigma_{k+1}$ and $p\big|_{\sigma_i} = T$ for $1 \leq i \leq k$, $q\big|_{\sigma_{k+1}} = T$, or

b) There exist an infinite sequence of subsequences $\sigma_1, \sigma_2, \ldots,$ such that $\sigma = \sigma_1 \circ \sigma_2 \circ \cdots$ and $p\big|_{\sigma_i} = T$ for every $i \geq 1$.

Intuitively we may offer the following explanations for the operators:

$\bigcirc p$ is true now if p holds in the next state of the sequence.

$\Box p$ holds now if p is true from now on.

$\Diamond p$ holds now if p is guaranteed to hold in some future state of the sequence.

$p\mathcal{U}q$ holds if q is guaranteed to hold in the future and p holds continuously until then.

pCq holds if σ can be decomposed into a prefix and a suffix, such that p holds over the prefix and q holds over the suffix.

pC^*q holds if σ is decomposeable into a finite or infinite sequence of subsequences such that q holds over the last subsequence and p over each of the preceding subsequences.

One of the significant extensions of the temporal system presented here over the one, say, in [MP1], is the introduction of the edge propositions. As we will see below, the edge propositions are essential in specifying a module since they distinguish between transitions effected by the module and those performed by the environment.

3. Computations and Specifications of Modules—The Shared Variables Model

As our first model of a concurrent system we consider a language in which interaction between parallel components is done via shared variables.

At first, we consider programs without procedures. The syntax of this subset is given as follows:

Let $y_1, \ldots, y_n$ be a fixed set of program variables. We define *statements* inductively as follows:

- An assignment $u := e$, where $u \in \{y_1, \ldots, y_n\}$ and e is an expression over these variables is a statement.
- If S_1, S_2 are statements then so is $S_1; S_2$.
- If $S_1, \ldots, S_m$ are statements and $t_1, \ldots, t_m$ are tests over $y_1, \ldots, y_n$ then

$$\mathop{\square}_{i=1}^{m} t_i \rightarrow S_i$$

  is a (conditional) statement.
- If S is a statement and t a test, then *while* t *do* S is a (while) statement.
- If S_1 and S_2 are statements then $S_1 \parallel S_2$ is a (parallel) statement.

Let s be a *state* in the execution of a statement S. As such it contains a value assignment to each variable y_i, denoted by $y_i(s)$ that can be extended to the evaluation of expressions and tests over s, denoted respectively by $e(s)$ and $t(s)$. In addition, s will contain a control component that describes the location of the execution in the program. Without entering into the exact form of the control component, we assume that there exists some operational semantics for the statement S. In particular, this semantics should define for each state s a set of possible successors under a single atomic step of the program S. The only state that has no successors is one in which all the components of S have terminated. We refer to such a state as *terminal*. The fact that a state is terminal is indicated solely by the value of its control component and is independent of the values of the program variables in that state.

The state s may contain assignment of values to variables not appearing in S as well as control components belonging to other processes. In that

case any action of S must preserve all these values which are inaccessible to S. We refer therefore to s as an *extended* state of S.

Consider the following labelled sequence:

$$\sigma : s_0 \xrightarrow{\lambda_0} s_1 \xrightarrow{\lambda_1} s_2 \rightarrow \cdots$$

where:

1. Each s_i is an extended state of S.
2. The labels λ_i, $i = 0, 1, \ldots$ may only assume one of the two values E, Π.
3. Whenever $s_i \xrightarrow{\Pi} s_{i+1}$, then s_{i+1} is an S-successor of s_i. This is considered to be a step of the statement S.
4. Whenever $s_i \xrightarrow{E} s_{i+1}$, at least the control component corresponding to S is identical in s_i and s_{i+1}. This is considered to be a step performed by the environment.
5. The sequence σ is finite iff its last state is terminal for S.

We consider σ to be an execution sequence in which atomic actions of S, labelled by Π, are arbitrarily interleaved by actions of an environment labelled by E. Environment actions may change the values of all the variables except for the control component for S.

A sequence satisfying these conditions is called a Π computation of S.

The sequence σ can easily be extended to a model $M = (\alpha, \sigma, I, J)$ for temporal formulas as follows:

Take α to be an arbitrary global assignment.
The sequence σ is the given sequence.
For a state s we define $I(s, y_j) = y_j(s)$, for $j = 1, \ldots, n$.

For a transition t_i, labelled by $\lambda_i \in \{E, \Pi\}$ we define $J(t_i, \Pi) = T$ iff $\lambda_i = \Pi$. For other edge propositions we define J arbitrarily.

With this definition we may interpret temporal formulas over M. We refer to M as a Π model corresponding to S. Note that we may regard the label E as an abbreviation for $\sim \Pi$.

Let $\varphi = \varphi(\Pi)$ be a temporal formula whose only edge proposition is Π.

We say that S *satisfies* φ (alternately, φ is a specification for S) if all Π models corresponding to S satisfy φ. We write $\{S\}\varphi$ to denote this fact. As an example for these concepts consider the following two modules (statements):

$S_1 :: y_1 := 1$

$S_2 :: \textit{when } y_1 = 1 \textit{ do } y_2 := 1.$

(this statement is equivalent to *while* $y_1 \neq 1$ *do skip*; $y_2 := 1$). A valid specification for S_1 is:

$$\varphi_1(\Pi) : \diamond[(y_1 = 1) \wedge \square(\Pi \supset (y_1 = \bigcirc y_1))].$$

It states that under any Π computation of S_1 there will be an instant in which $y_1 = 1$ and such that S_1 will never again change the value of y_1. Note that this still allows the environment to arbitrarily change y_1 both before and after S_1 assigns it the value 1.

A valid specification for S_2 is:

$$\varphi_2(\Pi) : \diamond[(y_1 = 1) \wedge \square(E \supset (y_1 = \bigcirc y_1))] \supset \diamond(y_2 = 1).$$

That is, if the environment guarantees that y_1 will eventually assume the value 1 and that the environment will never again modify the value of y_1, then y_2 is guaranteed to assume the value 1. We use here again the abbreviation of E for $\sim \Pi$.

Proof Rules for the Satisfaction Relation

Following, we present several rules that illustrate the basic approach for building a proof incrementally.

The first rule is the *consequence* rule that allows us to adopt as a valid specification any formula that follows logically from a valid specification.

(CONS):

$$\frac{\vdash \{S\}\varphi \qquad \vdash \varphi \supset \psi}{\vdash \{S\}\psi}$$

The following rules correspond each to one of the rules for the construction of statements:

The assignment axiom:

(ASGN):

$$\vdash \{ u := e \}[(E)\ \mathcal{U}\ (\Pi \wedge (\bigcirc \overline{y} = y \circ [u \leftarrow e]) \wedge \bigcirc(E\ \mathcal{U}\ \textit{fin}))]$$

The assignment axiom states that a proper execution of the assignment $u := e$ must always be describable as follows: it starts with a finite sequence of E steps followed by a single Π step. This Π step is such that the $\overline{y}$ variables after the assignment equal their values before the assignment with the difference that the value of e is substituted for u. The Π step is followed by a *finite* sequence of E steps. The predicate *fin* is an abbreviation for $\circledcirc$ *false* which is true for the *last* state in a sequence. In subsequent rules we will abbreviate $E\mathcal{U}\textit{fin}$ by E^*.

The last two rules may be used together with some temporal reasoning to derive the specification φ_1 for the statement S_1 shown above.

The following rule corresponds to the construction of a statement by concatenating two smaller statements:

(CONC):

$$\frac{\vdash \{ S_i \}\varphi_i \text{ for } i = 1,2}{\vdash \{ S_1 ; S_2 \}[\varphi_1 \mathcal{C} \varphi_2]}$$

To argue for the soundness of this rule, let φ_i be a valid specification for S_i, $i = 1,2$. Let σ be an execution sequence of $S_1; S_2$. Then σ must be representable as $\sigma = \sigma_1 \circ \sigma_2$ where σ_1 is an execution of S_1 and σ_2 an execution of S_2. The case that σ_1 is infinite due to divergence of S_1 is also included. By the validity of φ_1, φ_2, σ_1 satisfies φ_1 and σ_2 satisfies φ_2 and hence σ must satisfy $\varphi_1 \mathcal{C} \varphi_2$.

Let us introduce the following abbreviation:

$$EP(u_1, \ldots, u_m) = (E \supset \bigwedge_{i=1}^{m} [u_i = \bigcirc u_i])$$

Its meaning is that if the current step is an environment step then it preserves the values of the variables $u_1, \ldots, u_m$. Symmetrically $PP(u_1, \ldots, u_m)$ means that the process preserves these values.

Let us illustrate the application of the CONC rule in order to establish the specification φ_2 for the module S_2 above.

1. $\vdash \{ \textit{while } y_1 \neq 1 \textit{ do skip} \}[\Diamond[(y_1 = 1) \wedge \Box EP(y_1)] \supset \Diamond \textit{fin}]$

This statement says that if the environment promises to eventually set y_1 to 1 and never modify y_1 again, the loop is guaranteed to terminate. Its derivation must be done using the while rule below.

2. $\vdash \{ y_2 := 1 \}[\Diamond(y_2 = 1)]$

This specification can be derived using the ASGN axiom and the CONS rule.

3. $\vdash \{ S_2 \}[(\Diamond[(y_1 = 1) \wedge \Box EP(y_1)] \supset \Diamond \textit{fin})\, \mathcal{C}\, (\Diamond(y_2 = 1))]$

By the CONC rule applied to 1,2.

4. $\vdash (\Diamond[(y_1 = 1) \wedge \Box EP(y_1)] \supset \Diamond \textit{fin}) \mathcal{C} (\Diamond(y_2 = 1)) \supset$
$[\Diamond[(y_1 = 1) \wedge \Box EP(y_1)] \supset \Diamond(y_2 = 1)]$

This is a pure temporal logic statement and should be provable in an appropriate proof system for the temporal language we consider. We will not discuss the details of such a proof system here. A proof system for the language that does not include the $\mathcal{C}$ operator is provided in [MP3] and its extension to include this operator is under process.

To justify the statement semantically, we observe that it has the general form equivalent to:

$$([(p \supset \Diamond \textit{fin})\, \mathcal{C}\, (\Diamond q)] \wedge p) \supseteq \Diamond q.$$

Consider a sequence σ that satisfies both $(p \supset \Diamond \textit{fin})\, \mathcal{C}\, (\Diamond q)$ and p. Then either it is an infinite sequence that satisfies $p \supset \Diamond \textit{fin}$ and p, or it is representable as $\sigma = \sigma_1 \circ \sigma_2$ where σ_1 is finite and σ_2 satisfies $\Diamond q$. The first case is impossible since no infinite sequence can satisfy $\Diamond \textit{fin}$. Hence $\Diamond q$ holds in a suffix of σ, consequently $\Diamond q$ holds in σ.

Applying the CONS rule to 3 and 4 we obtain

5. $\vdash \{ S_2 \} \varphi_2$

Next we consider a rule for the conditional statement. Let S be the process defined by

$$S :: \mathop{\Box}_{i=1}^{m} (t_i \rightarrow S_i).$$

Introduce the abbreviation:

$$\text{test }(p) = \Pi \wedge p \wedge [\bigcirc \bar{y} = \bar{y}]$$

It describes a step done by the process in which the predicate p evaluates to T and all the values of the variables are preserved.

We make here the simplifying assumption of ***exhaustiveness*** by which for every $\bar{y}$, $\vdash \bigvee_{i=1}^{m} t_i(\bar{y})$. This implies that there is always a branch that can be taken. The more general case will be considered later.

Then we have the rule:

(COND):

$$\frac{\vdash \{ S_i \} \varphi_i \text{ for } i = 1, \ldots, m}{\vdash \{ \square_{i=1}^{m} t_i \rightarrow S_i \} [(E)\, \mathcal{U}\, (\bigvee_{i=1}^{m} [\text{ test }(t_i) \wedge \bigcirc \varphi_i])]}$$

By this rule the execution of a conditional statement, consists of a finite prefix of environment steps followed by a successful testing step for some i followed by the execution of S_i.

The next type of statement to be considered is the while statement.

(LOOP):

$$\frac{\vdash \{ S \} \varphi}{\vdash \{ \textit{while } t \textit{ do } S \} [((E)\, \mathcal{U}\, [\text{ test }(t) \wedge \bigcirc \varphi])\ \mathcal{C}^* \ ((E)\, \mathcal{U}\, (\text{test }(\sim t) \wedge \bigcirc E^*))]}$$

Thus if φ is a valid specification for the loop's body S, then every σ, an execution of the while statement must be representable as $\sigma = \sigma_1 \circ \sigma_2 \cdots$. Let $\{\sigma_i,\ i = 1, 2, \ldots\}$ be the finite or infinite sequence of subsequences into which σ decomposes. Then, the above rule allows several possibilities. The first is that $\{\sigma_i\}$ is infinite and each σ_i, $i = 1, 2, \ldots$ is finite and satisfies $p : (E)\mathcal{U}[\text{test }(t) \wedge \bigcirc \varphi]$. This implies that each σ_i consists of a finite prefix of E steps, followed by a successful t test, followed by a finite suffix that satisfies φ. This corresponds to an execution of the while statement in which the loop never terminated but each execution of the body was finite.

The second possibility is that there are only finitely many subsequences, $\sigma_1, \ldots, \sigma_{m+1}$ all of which satisfy φ such that each of $\sigma_1, \ldots, \sigma_m$

is finite and σ_{m+1} is infinite. This corresponds to an execution of the while statement that iterated $m+1$ times and the $m+1$'st execution of the body diverges.

The last possibility is that there are $m+1$ subsequences $\sigma_1,\ldots,\sigma_{m+1} (m \geq 0)$, all of which are finite, such that $\sigma_1,\ldots,\sigma_m$ satisfy φ and σ_{m+1} satisfies $(E)\mathcal{U}(\text{test } (\sim t) \wedge \bigcirc E^*)$. Thus σ_{m+1} contains a failed t test and a finite suffix of E-steps. This corresponds to the case that the execution of the loop terminated. Note that, by the definition of C^* we also allow the case of $m = 0$ in which the loop terminated immediately.

We may use now the LOOP rule in order to derive the specification for the waiting loop in S_2. This is done as follows:

1. $\vdash \{ \textit{skip} \} [\Box PP(y_1) \wedge \Diamond \textit{fin}]$

This can be derived by a combination of ASGN and CONS. It ensures that every execution of *skip* is finite and each process step in it preserves y_1.

2. $\vdash \{ \textit{while } y_1 \neq 1 \textit{ do skip} \} [(\Box PP(y_1) \wedge \Diamond(y_1 \neq 1) \wedge \Diamond \textit{fin})\, C^* (\Diamond \textit{fin})]$

This is derivable from 1 by the LOOP and CONS rules and some temporal reasoning. We have used in particular the obvious implication $E \vee \text{test}(t) \supset PP(y_1)$.

3. $\vdash [(\Box PP(y_1) \wedge \Diamond(y_1 \neq 1) \wedge \Diamond \textit{fin})\, C^* (\Diamond \textit{fin})] \supset [\Diamond \textit{fin} \vee (\Box(\sim \textit{fin} \wedge PP(y_1)) \wedge \Box \Diamond (y_1 \neq 1))]$

This is a true temporal logic statement. It implies that a sequence satisfying the C^* formula can be infinite only if $y_1 \neq 1$ infinitely many times.

4. $\vdash \Diamond((y_1 = 1) \wedge \Box EP(y_1)) \supset\ \sim [\Box(\sim \textit{fin} \wedge PP(y_1)) \wedge \Box \Diamond (y_1 \neq 1)]$

This statement says that the promise of the environment to eventually set y_1 to 1 and never again modify its value is inconsistent with an infinite sequence in which infinitely many times $y_1 \neq 1$ and the process does not modify y_1 either.

We may now combine 2,3 and 4 to obtain

5. $\vdash \{ \textit{while } y_1 \neq 1 \textit{ do skip} \} [\Diamond((y_1 = 1) \wedge \Box EP(y_1)) \supset\ \Diamond \textit{fin}]$

Considering the expression of a potentially infinite loop by the $\mathcal{C}^*$ operator as it appears in the LOOP rule, we may propose the following rule for the more general case of the conditional statement. In this rule we no longer assume exhaustivity of the branch conditions.

(GCOND):

$$\frac{\vdash \{S_i\}\varphi_i \text{ for } i = 1, \ldots, m}{\vdash \{\ \Box_{i=1}^{m} t_i \rightarrow S_i\}[p\mathcal{C}^* q]}$$

where: p is $(E)\mathcal{U}(\text{test}(\bigwedge_{i=1}^{m} \sim t_i))$
and q is $(E)\mathcal{U}(\bigvee_{i=1}^{m}[\text{test}(t_i) \wedge \bigcirc\varphi_i])$

Thus, the successful execution of the conditional statement may be preceded by a finite or infinite number of unsuccessful attempts in which all the tests failed.

The last rule in our system is the one for parallel composition. It must be realized that the added complication of introducing edge labels and the distinction between E and Π transitions were all done in preparation for the parallel composition.

(PAR):

A. $\vdash \{S_i\}\varphi_i$ for $i = 1, 2$
B. $\vdash [(\Box \sim (\Pi_1 \wedge \Pi_2)) \wedge \Box \Diamond(\Pi_1 \vee \Pi_2 \vee \mathit{fin}) \wedge (\varphi_1(\Pi_1))\, \mathcal{C}\, (\Box(\sim \Pi_1)) \wedge (\varphi_2(\Pi_2))\, \mathcal{C}\, (\Box(\sim \Pi_2))] \supset \varphi(\Pi_1 \vee \Pi_2)$

$\vdash \{S_1 \parallel S_2\}\varphi$

The parallel rule contains two premises. The first premise simply states that φ_i is a valid specification for S_i.

The second premise considers a sequence σ that satisfies four requirements. The first requirement is that the edge propositions Π_1 and Π_2 are exclusive, i.e. never jointly true on the same edge.

The second requirement, $\Box \Diamond(\Pi_1 \vee \Pi_2 \vee \mathit{fin})$, allows σ to be infinite only if there are infinitely many transitions which are either Π_1 or Π_2 transitions.

The third requirement postulates that σ is representable as $\sigma = \sigma_1 \circ \sigma_2$, where σ_2 contains no Π_1 transitions and σ_1 satisfies $\varphi_1(\Pi_1)$.

The fourth requirement is a symmetric postulate for φ_2 and Π_2.

Under these four conditions, premise B requires that σ satisfies $\varphi(\Pi_1 \vee \Pi_2)$. In this notation we imply that the only reference to edge propositions in φ is via the disjunction $\Pi_1 \vee \Pi_2$.

Note that if $\varphi_1(\Pi)$ referred to the environment in statements such as $E \supset (\bigcirc y_1 = y_1)$, then $\varphi_1(\Pi_1)$ contains the clause $(\sim\Pi_1) \supset (\bigcirc y_1 = y_1)$, that states the property of preservation for all non Π_1 transitions, including the Π_2 transitions. Thus we may interpret $\varphi_1(\Pi_1)$ as considering Π_2-steps as environment steps.

The general idea underlying the PAR rule is that any execution of $S_1 \parallel S_2$ can always be viewed both as an S_1 execution in which both the S_2 steps and the true environment steps are regarded as environmental, and symmetrically as an S_2 execution in which both the S_1 and the E steps are regarded as environmental.

Indeed, let us argue for the soundness of the rule. Let σ be a Π execution of $S_1 \parallel S_2$ where we label each non-Π transition by $E = \sim \Pi$. Assume also that each φ_i is a valid specification for S_i, $i = 1,2$ and that premise B holds. It is always possible to additionally label each Π transition in σ by Π_1 or Π_2 according to whether it corresponds to an S_1-step or an S_2-step in the interleaved execution that is the execution of $S_1 \parallel S_2$. This extra labelling obviously satisfies $\square \sim (\Pi_1 \wedge \Pi_2)$ since each Π step is either a Π_1-step or a Π_2-step but not both.

Next we observe that if there are only finitely many Π_1 and Π_2 steps then both processes eventually terminate, and hence σ must be finite. This shows that σ satisfies $\square \diamond (\Pi_1 \vee \Pi_2 \vee \mathit{fin})$.

Next, it is clear that with the additional labelling σ is *almost* a Π_1 execution of S_1. The *almost* qualifier refers to the possibility that the corresponding S_1 execution is finite while σ is infinite due to a divergent execution of S_2. Thus, at any case, we can always represent σ as $\sigma = \sigma_1 \circ \sigma_2$ where σ_1 is a complete Π_1-execution of S_1 and σ_2 contains no Π_1 transitions. By the assumption that φ_1 is a valid S_1-specification it follows that σ satisfies $(\varphi_1(\Pi_1))\mathcal{C}(\square(\sim \Pi_1))$.

By a fully symmetric argument σ satisfies also $(\varphi_2(\Pi_2))\mathcal{C}(\square(\sim \Pi_2))$.

By premise B we conclude that σ satisfies $\varphi(\Pi_1 \vee \Pi_2)$.

By recalling that an edge is labelled by Π_1 or Π_2 iff it is labelled by

Π we obtain that σ also satisfies $\varphi(\Pi)$.

Let us use the PAR rule and the previously derived specifications for S_1 and S_2 of our example, in order to derive a specification for $S_1 \parallel S_2$.

Let X denote an edge predicate, i.e. Π or $\sim \Pi$. Then we define:

$$P_r(X, \bar{u}) = [X \supset \bigcirc \bar{u} = \bar{u}]$$

This formula states that if the current transition is an X-transition, it preserves the values of the variables $\bar{u}$.

1. $\vdash \{ S_1 \} [\Diamond ((y_1 = 1) \wedge \Box P_r(\Pi, y_1))]$

   This specification for S_1 was derived earlier.

2. $\vdash \{ S_2 \} [\Diamond ((y_1 = 1) \wedge \Box P_r(\sim \Pi, y_1)) \supset \Diamond (y_2 = 1)]$

   Previously derived.

3. $\vdash \{ \Box \sim (\Pi_1 \vee \Pi_2) \wedge \Box \Diamond (\Pi_1 \vee \Pi_2 \vee \mathit{fin}) \wedge$
   $(\Diamond ((y_1 = 1) \wedge \Box P_r(\Pi_1, y_1)) \mathcal{C} (\Box (\sim \Pi_1)) \wedge$
   $[\Diamond ((y_1 = 1) \wedge \Box P_r(\sim \Pi_2, y_1)) \supset \Diamond (y_2 = 1)] \mathcal{C} (\Box (\sim \Pi_2)) \} \supset$
   $[\Box P_r(\sim (\Pi_1 \vee \Pi_2), y_1) \supset \Diamond (y_2 = 1)]$

   This is a pure temporal logic statement. To argue for its validity consider a (Π_1, Π_2) sequence σ that satisfies all the antecedents of the implication and also $\Box P_r(\sim (\Pi_1 \vee \Pi_2), y_1)$.By the third antecedent eventually $y_1 = 1$ and henceforth $\Box P_r(\Pi_1, y_1)$. By $\Box P_r(\sim \Pi_1 \wedge \sim \Pi_2, y_1)$ it follows that from that instant on we actually have $\Box P_r(\sim \Pi_2, y_1)$. By the fourth antecedent, these two conditions imply that eventually $y_2 = 1$. We conclude that σ satisfies $\Diamond (y_2 = 1)$.

4. $\vdash \{ S_1 \parallel S_2 \} [\Box P_r(\sim \Pi, y_1) \supset \Diamond (y_2 = 1)]$

   By the PAR rule applied to 1,2 and 3.

This establishes a specification for $S_1 \parallel S_2$ by which if the environment promises not to tamper with y_1, $S_1 \parallel S_2$ guarantees to set y_2 to 1.

4. Comments Regarding Soundness and Completeness of the System

The soundness of the system we have presented was considered for each rule as it was introduced. In this section we consider the complete-

ness of the system. First, we show that our temporal language is powerful enough to express in closed form the temporal semantics of each statement of our programming language. Similar approaches for unstructured computational models are considered in [Pn] and [MP2]. For each statement S we define $\varphi(\Pi) = \mathcal{M}[S]$, the temporal semantics of S, which is a temporal formula with one free edge proposition, Π. The meaning of φ is that a Π sequence satisfies $\varphi(\Pi)$ iff it is an execution sequence of S. Following is the inductive definition of $\mathcal{M}[S]$ for the different types of statements, where we abbreviate $\sim \Pi$ to E.

$$\mathcal{M}[u := e] = (E)\,\mathcal{U}\,[\Pi \wedge \bigcirc \bar{y} = \bar{y} \circ [u \leftarrow e]) \wedge \bigcirc E^*]$$

$$\mathcal{M}[S_1; S_2] = (\mathcal{M}[S_1])\,\mathcal{C}\,(\mathcal{M}[S_2])$$

For an exhaustive conditional:

$$\mathcal{M}[\Box_{i=1}^{m} t_i \rightarrow S_i] = (E)\,\mathcal{U}\,(\bigvee_{i=1}^{m}[test(t_i) \wedge \bigcirc \mathcal{M}[S_i])$$

$$\mathcal{M}[while\ t\ do\ S] = pC^*q$$

where

$$p = (E)\mathcal{U}[test(t) \wedge \bigcirc \mathcal{M}[S]], \text{ and}$$

$$q = (E)\mathcal{U}[test(\sim t) \wedge \bigcirc E^*]$$

$$\mathcal{M}[S_1 \parallel S_2] = \exists \Pi_1, \Pi_2 \cdot [\Box((\Pi \equiv \Pi_1 \vee \Pi_2) \wedge \sim (\Pi_1 \wedge \Pi_2)) \wedge \Box \Diamond (\Pi \vee fin) \wedge (\mathcal{M}[S_1](\Pi_1))\mathcal{C}(\Box(\sim \Pi_1)) \wedge (\mathcal{M}[S_2](\Pi_2))\mathcal{C}(\Box(\sim \Pi_2))]$$

In the last definition we extended our temporal language even further, by allowing quantification over edge propositions. A proper definition of the semantics of this construct is straightforward. Such constructs are allowed and considered in [W].

It is now a simple matter to consider each of the rules and show, by induction on the structure of a statement S, that for each statement S the following is provable in our system:

$$\vdash \{S\}[\mathcal{M}[S]]$$

Let now φ be any specification valid for the statement S. By the definition of the semantics of S, any sequence satisfying $\mathcal{M}[S]$ is an execution of S

and hence satisfies φ. Thus, the implication $M[S] \supset \varphi$ is semantically valid over all sequences. If we accept all semantically valid purely temporal formulas as axioms then trivially:

$$\vdash M[S] \supset \varphi$$

We may now use the CONS rule to obtain that $\{ S \} \varphi$ is provable ***relative*** to the theory of pure temporal logic. This gives us the desired relative completeness result.

5. Is This the Best that Can be Done?

As we have repeatedly emphasized, this paper presents but a preliminary version of a compositional temporal proof system, intended mainly to demonstrate that such approach is feasible. In constructing the system presented here we encountered several design choices that we have decided to resolve in a certain way. In this section we consider some of the alternatives.

One such choice involved the issue whether we should allow both finite and infinite executions or force all executions to be infinite, as is done for example in [MP1]. The first alternative, that we have adopted here, simplifies the sequential composition rules such as CONC, COND and LOOP but complicates the parallel composition rule PAR. If we would have forced all executions to be infinite by extending finite ones by an infinite tail of E steps, we can get the following simpler PAR rule:

(PAR-II):

$$\frac{\vdash \{ S_i \} \varphi_i \text{ for } i = 1,2 \qquad \vdash [\Box \sim (\Pi_1 \wedge \Pi_2) \wedge \varphi_1(\Pi_1) \wedge \varphi_2(\Pi_2)] \supset \varphi(\Pi_1 \vee \Pi_2)}{\vdash \{ S_1 \parallel S_2 \} \varphi}$$

On the other hand the concatenation rule would have a form somewhat like the following:

(CONC-II):

$$\frac{\vdash \{ S_i \} \varphi_i \text{ for } i = 1,2 \qquad \vdash [\varphi_1(\Pi_1) \wedge (\sim \Pi_2 \wedge \Diamond \Pi_1) \, U (\varphi_2(\Pi_2) \wedge \Box \sim \Pi_1)] \supset \varphi(\Pi_1 \vee \Pi_2)}{\vdash \{ S_1 ; S_2 \} \varphi}$$

In this rule we use the ***unless*** operator U which is the weaker version of the *until* operator $\mathcal{U}$. In this rule we consider a sequential composition to be a special case of the parallel composition in which we require all the Π_1 transitions, corresponding to S_1 steps, to precede all the Π_2 transitions. The case that S_1 never terminates is also considered. Note that in this version we do not need the C operator and can use instead the U operator.

Another issue that we had to consider is whether to use at all the C operator and the heavier C^* operator. As is observed in [HMM], the C operator (not to mention C^*) can greatly increase the complexity of the decision procedure where one exists, and in general complicate the proof procedure. Consequently, one may look for rules which use simpler operators. One such rule was given above for concatenation where we replaced the C operator by U. Another example is a loop rule in which we replace C^* by C and introduce auxiliary edge propositions.

(LOOP-II):

$$\frac{\begin{array}{l} \vdash \{\, S \,\}\varphi' \\ \vdash [\Box \sim (\Pi_1 \wedge \Pi_2) \wedge \Box Pr(\Pi_1, \bar{y}) \wedge [\sim (\Pi_1 \vee \Pi_2)]\, \mathcal{U}\, \Pi_1 \wedge \\ \quad \Box(\Pi_1 \wedge t \supset \bigcirc[(\varphi'(\Pi_2)) C(\Pi_1)]) \wedge \\ \quad \Box(\Pi_1 \wedge \sim t \supset \bigcirc[(\sim (\Pi_1 \vee \Pi_2))\mathcal{U}\, fin])] \supset \varphi(\Pi_1 \vee \Pi_2) \end{array}}{\vdash \{\, \mathit{while}\ t\ \mathit{do}\ S \,\}\varphi}$$

In this rule we labelled the tests by Π_1 and the executions of the loop's body by Π_2.

It is obvious that many different versions of similar systems can be constructed according to different design decisions. Only time and experience in applying these systems can point out the relative advantages of one system in comparison to the others.

6. Adding Procedures

We sketch now the additions and extensions that are needed in order to allow in our programming language procedures, recursion and blocks.

We start with a rule for a block or a new variable declaration.

(DECL):

$$\frac{\vdash \{S\}(\Box P_r(E,u) \supset \varphi) \qquad u \notin \text{free}(\varphi)}{\vdash \{\, \textit{begin new } u; S \textit{ end}\}\varphi}$$

The rule states that if under the assumption that the environment does not modify u we can obtain that φ is a valid specification for S where φ does not contain u freely, then, unconditionally, φ is a valid specification for the statements: (*begin new* u; S *end*). This depends on the interpretation that since u is not known outside of S no external environment can modify it. The only modifications to u can come from within S.

Next we consider the simple case of a *single* procedure defined by $f(x,y) \Leftarrow B$ where B is a statement, the procedure body, that contains x and y as free variables. The body may contain instances of procedure calls '<u>call</u> $f(u,v)$'. We assume here that the only actual arguments allowed are simple disjoint variables and that the parameter transfer mechanism is call by reference.

Let us extend the definitions of the temporal semantics given above to the temporal semantics under the assumption of φ *for* f. Denoted by $\mathcal{M}_\varphi[S]$, its definition for each type of statement excluding a call statement is identical to that of $\mathcal{M}[S]$. The definition of the semantics of a call statement is given by:

$$\mathcal{M}_\varphi[\textit{call } f(u,v)] = (E)\mathcal{U}(\varphi(u,v))$$

Similarly we can define the notion of ψ being a valid specification for S under the assumption of φ *for* f. This is denoted by $\{\, S \,\}_\varphi \psi$. The proof rules for this construct are identical to the ones presented above with the addition of:

(CALL):

$$\vdash \{\, \textit{call } f(u,v) \,\}_\varphi [(E)\mathcal{U}(\varphi(u,v))]$$

Provability and the semantics under an assumption are connected by the following rule:

(SEM):

$$\frac{\vdash \{S\}_{\varphi}\psi}{\vdash \mathcal{M}_{\varphi}[S] \supset \psi}$$

We are ready to formulate the rule for procedure declaration:

(PROC):

$$\frac{\{\vdash \varphi \supset \mathcal{M}_{\varphi}[B]\} \Rightarrow \{\vdash \varphi \supset \psi\}}{\vdash \{\textit{call } f \textit{ where } f \Leftarrow B\}\psi}$$

The premise to the rule is that under the hypothesis $\varphi \supset \mathcal{M}_{\varphi}[B]$ we are able to deduce $\varphi \supset \psi$. We may then conclude that the actual call to f satisfies ψ.

The soundness of the rule is based on the fact that the semantics of the recursive definition $f \Leftarrow B$ is the *maximal* fixpoint of the temporal implication: $\varphi \supset \mathcal{M}_{\varphi}[B]$. Consequently, if, by the premise, ***every*** fixpoint of this equation implies ψ then in particular the maximal fixpoint implies it.

We illustrate the style of proof according to these rules by the following simple example (the factorial function)

```
f(x, y) ⇐ (if x = 0 then y := 1
          else begin new t;
               t := x − 1;
               call f(t, y);
               y := y*x
           end)
```

We wish to prove (under the context of this declaration):

$$\vdash \{\textit{call } f(x,y)\}[(x = a \geq 0) \wedge \Box EP(x,y) \supset \Diamond(y = a! \wedge E^*)]$$

The proof proceeds as follows:

1. $\vdash \varphi \supset \mathcal{M}_{\varphi}[B]$ Assumption.
   B is the body of the procedure declaration.

2. $\vdash \varphi(x,y) \wedge x = 0 \wedge \Box EP(x,y) \supset \Diamond(y = 1 \wedge E^*)$
   By considering $\mathcal{M}_{\varphi}[B]$ for $x = 0$.

Next we prove by induction on $a \geq 0$ that

$$\varphi \wedge (x = a \geq 0) \wedge \Box EP(x,y) \supset \Diamond(y = a! \wedge E^*)$$

The base for the induction is given in 2.

3. $\vdash [\varphi \wedge (x = a \geq 0) \wedge \Box EP(x,y)] \supset \Diamond(y = a! \wedge E^*)$
This is the induction hypothesis.

4. $\vdash \{\, t := x - 1 \,\}_\varphi [(x = a + 1) \wedge \Box EP(x,y,t) \supset \Diamond(t = a \wedge x = a + 1 \wedge E^*)]$
By ASGN.

5. $\vdash \{\, call\ f(t,y) \,\}_\varphi \varphi(t,y)$
By CALL.

6. $\vdash \{\, call\ f(t,y) \,\}_\varphi [(t = a) \wedge x = a + 1 \wedge \Box EP(x,y,t) \supset \Diamond(y = a! \wedge x = a + 1 \wedge E^*)]$
By 3,5 and temporal reasoning.

7. $\vdash \{\, t := x - 1; call\ f(t,y); y := y{*}x \,\}_\varphi [(x = a + 1) \wedge \Box EP(x,y,t) \supset \Diamond(y = (a + 1)! \wedge E^*]$
By various rules and 4,6.

8. $\vdash \{\, B \,\}_\varphi [(x = a + 1 > 0) \wedge \Box EP(x,y) \supset \Diamond(y = (a + 1)! \wedge E^*)]$
By DECL, COND and temporal reasoning.

9. $\vdash \mathcal{M}_\varphi[B] \supset [(x = a + 1 > 0) \wedge \Box EP(x,y) \supset \Diamond(y = (a + 1)! \wedge E^*)]$
By SEM.

10. $\vdash \varphi \wedge (x = a + 1 > 0) \wedge \Box EP(x,y) \supset \Diamond(y = (a + 1)! \wedge E^*)$
By 1 and 9.

11. $\vdash \varphi \supset [(x = a \geq 0) \wedge \Box EP(x,y) \supset \Diamond(y = a! \wedge E^*)]$
By induction, based on 2,3–10.

12. $\vdash \{\, call\ f(x,y) \,\} [(x = a \geq 0) \wedge \Box EP(x,y) \supset \Diamond(y = a!) \wedge E^*)]$
By PROC 1 and 11. ∎

In order to prove completeness for the system with procedures we have to extend the expressibility of the temporal language even further. Let $\chi(\varphi)$ be a temporal formula containing several positive instances of the (temporal) predicate symbol φ. An instance of a subformula is defined as positive if it appears under an even number of negations.

Consider the implication $\varphi \supset \chi(\varphi)$ which can be regarded as an equation (inequality in fact) for φ. A temporal predicate Φ is a ***solution*** to $\varphi \supset \chi(\varphi)$ if $\Phi \supset \chi(\Phi)$ is a valid statement, holding for all models. It can be shown that every positive implication $\varphi \supset \chi(\varphi)$ has a ***maximal*** solution $\hat{\Phi}$ such that $\hat{\Phi} \supset \chi(\hat{\Phi})$, in fact $\hat{\Phi} \equiv \chi(\hat{\Phi})$, and it bounds from above any other solution. The later statements mean that every Φ that satisfies $\Phi \supset \chi(\Phi)$ must also satisfy $\Phi \supset \hat{\Phi}$.

Thus the ordering that we use and that underlies the fixpoint theory of positive implications is the one defined by: $\Phi \sqsubseteq \Psi \Leftrightarrow \Phi \supset \Psi$.

Let us denote the maximal fixpoint solution to the positive implication $\varphi \supset \chi(\varphi)$ by $\nu\varphi \cdot \chi(\varphi)$. As an example it is not difficult to see that $pC^*q \equiv \nu\varphi \cdot [q \vee pC\varphi]$.

Fixpoint operators were first introduced to temporal logic in [W], and the extension made here is to allow an arbitrary positive implication, removing the restriction to right linear forms which is imposed in [W].

With this extension we can give a closed form expression for the semantics of a procedure call, namely:

$$\mathcal{M}[\mathit{call}\ f(x,y)\ \mathit{where}\ f \Leftarrow B] = \nu\varphi \cdot [\varphi \supset \mathcal{M}_\varphi[B]]$$

The relative completeness follows directly.

7. Communication Based Models

In this section we consider an appropriate proof theory for a model of concurrent systems that is based on explicit communication instead of shared variables. The basic ideas are similar to these of the shared variables model but some of the details emphasize the inherent differences between the models.

The programming language that we study includes the following statements: Assignment, Concatention, Parallel and While statements as before. The conditional or selection statement is where communication is introduced. Let $\sum = \{c_1, c_2, \dots\}$ be a set of ***channel names***. The general form of a selection statement is:

$$[(\underset{j\in K}{\Box}\ b_j \to S_j)\ \Box\ (\underset{j\in L}{\Box}\ b_j; c_j!e_j \to S_j)\ \Box\ (\underset{j\in M}{\Box}\ b_j; c_j?u_j \to S_j)]$$

where we have partitioned the set of alternatives into three disjoint sets, indexed respectively by K, L, M. Alternatives belonging to K represent purely boolean choices. Alternatives belonging to L represent output guards that are ready to send a value e_j along a channel c_j. Alternatives belonging to M represent input commands that are ready to accept a value from a channel c_j and store it in u_j. A single input-output command of the form $c!e$ can always be represented by a single alternative selection $[\Box c!e \rightarrow skip]$.

For the definition of the models that represent executions in the COM language (communication based language) we generalize the labelling on the transitions from propositions to variables. Thus, we allow ***edge-variables*** $\lambda_1, \lambda_2 \ldots$ that are assigned values for each transition t_i in the sequence by the edge interpretation J.

When a model represents an execution sequence of a COM statement S, there is an edge variable λ that identifies the execution step that occured at each transition. It may assume the following values: $\lambda = \Pi$ denotes that the current step is an internal computation step of S. The value $\lambda = E$ denotes that the current step is an internal computation step of the environment, which may ***not*** include communication with S. $\lambda = c!v$ implies that S sent the value v along the channel c. $\lambda = c?v$ implies that S received the value v from the channel c. $\lambda = W(k_1, \ldots, k_r)$, where each k_i is either $c!$ or $c?$ for some $c \in \sum$, means that S has ***tried*** to communicate with any channel of the set $k_1, \ldots, k_r$ but failed to find a communication partner.

Since the proof rules are closely connected with the definition of the temporal semantics, as established by the SEM rule, we will give here only the temporal semantics for each of the statements.

We abbreviate $\lambda = E$ and $\lambda = \Pi$ to E and Π respectively, and denote $E\, \mathcal{U}\, fin$ by E^*.

$$\begin{aligned}
\mathcal{M}[u := e] &= E\, \mathcal{U}\, [\Pi \wedge (\bigcirc \bar{y} = \bar{y} \circ [u \leftarrow e]) \wedge \bigcirc E^*] \\
\mathcal{M}[S_1; S_2] &= \mathcal{M}[S_1]\, C\, \mathcal{M}[S_2] \\
\mathcal{M}[\textit{while } t \textit{ do } S] &= [E\, \mathcal{U}\, (test(t) \wedge \bigcirc \mathcal{M}[S])]\, C^* \\
&\quad\ [E\, \mathcal{U}\, (test(\sim t) \wedge \bigcirc E^*]
\end{aligned}$$

Next we consider a selection statement S that has the general form outlined above. That is, we distinguish three sets of alternatives, indexed

respectively by the sets K, L and M, and denoting the set of alternatives with purely boolean guards, the set of alternatives with output guards and the set of alternatives with input guards.

$$\mathcal{M}[S] = [[E\,\mathcal{U}\,[(\lambda = W(\bigcup_{\substack{j\in L\\ b_j=true}} e_j! \cup \bigcup_{\substack{j\in M\\ b_j=true}} c_j?)) \wedge \bigwedge_{j\in K} \sim b_j \wedge \bigcirc\bar{y} = \bar{y}] \wedge \bigcirc fin]$$

$$C^*\ [E\mathcal{U}\{\bigvee_{j\in K} (test(b_j) \wedge \bigcirc\mathcal{M}[S_j]) \vee$$

$$\bigvee_{j\in L} (b_j \wedge \lambda = c_j!e \wedge \bigcirc\bar{y} = \bar{y} \wedge \bigcirc\mathcal{M}[S_j]) \vee$$

$$\bigvee_{j\in M} (b_j \wedge \lambda = c_j?v \wedge \bigcirc\bar{y} = \bar{y} \circ [u_j \leftarrow v] \wedge \bigcirc\mathcal{M}[S_j])\}]$$

According to this definition the execution of a selection statement consists of a prefix that describes several failed attempts to select an alternative followed by a suffix that describes a successful selection and execution of an alternative. A failed attempt consists of a $\lambda = W(k_1, \ldots, k_m)$ step where each k_i, $i = 1, \ldots, m$ has the form $c?$ or $c!$ for $c \in \sum$. It implies that at that step, all the purely boolean guards evaluated to *false*, and the boolean parts of the guards corresponding to $k_1, \ldots, k_m$ evaluated to *true*. The current attempt at selection is considered to have failed because no true purely boolean guard was found and no willing communication partner matched the communications that S was ready to participate in. The possibility of deadlock is represented by an infinite prefix containing infinitely many failed attempts. The suffix represents a success in testing a purely boolean guard, or a successful communication followed by execution of the corresponding body.

Let k be of the form $c!$ or $c?$, then $\bar{k}$ denotes the matching communication, i.e., $c?$ or $c!$ respectively. We introduce the following abbreviation:

$$wait\ (\lambda, k) = \Box\bigcirc\, true \wedge \Diamond\, \Box((\lambda = E) \vee \lambda = W(k_1, \ldots, k, \ldots, k_m))$$

This formula states that the sequence is infinite, and from a certain state on, all the process steps are failed selection attempts, waiting for several possible communications one of which is always k. This may be described as "λ endlessly waiting for k". Of course, it may be waiting for several different k's at the same time.

The semantics for the parallel statement $S : S_1 \parallel S_2$ is given by:

$$
\begin{aligned}
&\mathcal{M}[S_1 \parallel S_2](\lambda) = \exists(\lambda_1, \lambda_2) \cdot \{ \\
&\quad \Box[(\lambda = \Pi) \supset [(\lambda_1 = E \wedge \lambda_2 = \Pi) \vee (\lambda_1 = \Pi \wedge \lambda_2 = E) \\
&\qquad\qquad \vee \exists c, v[(\lambda_1 = c!v \wedge \lambda_2 = c?v) \vee (\lambda_1 = c?v \wedge \lambda_2 = c!v)]] \wedge \\
&\quad\; (\lambda \neq \Pi) \supset [(\lambda_1 = E \wedge \lambda_2 = \lambda) \vee (\lambda_1 = \lambda \wedge \lambda_2 = E)]]] \\
&\quad \wedge \Box \Diamond \sim (\lambda = E) \wedge \\
&\quad [\mathcal{M}[S_1](\lambda_1)]\ \mathcal{C}\ [\Box(\lambda_1 = E \vee fin)] \wedge [\mathcal{M}[S_2](\lambda_2)]\ \mathcal{C}\ [\Box(\lambda_2 = E \vee fin)] \\
&\quad \wedge \sim \exists k[\text{wait}\ (\lambda_1, k) \wedge \text{wait}\ (\lambda_2, \bar{k})]\}
\end{aligned}
$$

This formula states that a λ-sequence σ is an admissible execution sequence of $S_1 \parallel S_2$ iff there exist two additional edge labelling λ_1 and λ_2 that satisfies a list of conditions. The intended meaning of λ_1 and λ_2 is that they reflect the local views of the joint execution σ as seen by S_1 and S_2 respectively. The conditions that $\lambda_1, \lambda_2, \lambda$ and σ have to satisfy are the following:

Whenever $\lambda = \Pi$, signifying a local transition, it can arise due to one of the statements S_i, $i = 1, 2$ taking a local transition while the other takes an idle step. Alternately, it can be caused by a matched communication between S_1 and S_2 that is externally visible as a local transition. Note that a communication step is the only step in which both processes are active at the same time. If $\lambda \neq \Pi$, this is either an idling step of S and hence of both S_1 and S_2, or a communication, failed or successful, with the external world. This is obtained by S_1, or S_2 performing this step and the other idling.

In addition we require that σ is infinite only if it has infinitely many non-idling steps. Then, we require, that σ with the λ_1 labelling is acceptable as an execution of S_1 to which we may have added an infinite suffix of $\lambda_1 = E$ steps. Similarly for S_2 and λ_2. Finally we disallow sequences in which from a certain point on S_1 is deadlocked waiting for some communications that include k while at the same time S_2 is deadlocked waiting for $\bar{k}$. Any such finite prefix should be eventually resolved by a $k - \bar{k}$ communication between S_1 and S_2.

This laconic description shows that communication based concurrent systems are also amenable to compositional temporal systems. The addition of procedures can be done in a way that is similar to their introduction in the previous model.

8. Conclusions

In this paper we presented a very preliminary approach that provides a compositional temporal proof system for concurrent programs. We are sure that a more careful study of some of the design decisions made here would probably lead to a more streamlined system which would be easier to apply and use for both the verification and the systematic development of concurrent programs.

Several research topics that require immediate attention are a better understanding of the tradeoffs between the power of the temporal operators used, the complexity of the proof procedures and the simplicity of the proof rules. Another very important topic is the incorporation of *past* operators into the proof system. In expressions such as pCq, any past operators embedded in q may only refer to the suffix of the sequence in which q holds and cannot (unless defined differently) penetrate to the p prefix of the sequence. It is important to analyze whether this is a limitation or rather the preferred situation.

The approach presented here combines the expressive power of temporal logic for specification of a rich class of properties with the modularity and compositionality of syntax directed methods.

Acknowledgement

The authors wish to acknowledge the hospitality of the STL/SERC Concurrency Workshop at Cambridge under whose auspices the initial ideas of this paper started their germination. The first two authors gratefully acknowledge support under S.E.R.C. grant GR/C/05670. All of us wish to thank Carol Weintraub for her excellent and speedy $\TeX$ing.

REFERENCES

[AFR] Apt, K.R., Francez, N., De Roever, W.P., A Proof System for Communicating Sequential Processes, ***TOPLAS*** 2,3(July 1980) pp. 359-385.

[BK] Barringer, H. and Kuiper, R., A Temporal Logic Specification Method Supporting Hierarchical Development, Manuscript, University of Manchester (November 1983).

[D] Dijkstra, E.W., A Discipline of Programming, Prentice Hall (1976).

[H1] Hoare, C.A.R., An Axiomatic Basis for Computer Programming, *CACM 12* 10 (Oct. 1969) pp. 576-580.

[H2] Hoare, C.A.R., A Calculus of Total Correctness for Communicating Processes, ***Science of Computer Programming, 1*** 1 (1981) pp. 49-72.

[HKP] Harel, D., Kozen, D. and Parikh, R., Process Logic: Expressiveness, Decidability and Completeness, *JCSS* **vol. 25** (1982) pp. 144–170.

[HMM] Halpern, J., Manna, Z. and Moszkowski, B., A Hardware Semantics Based on Temporal Intervals, ***Automata Languages and Programming, 10th Colloquium*** 1983,Lecture Notes in Computer Science #154, Springer Verlag, pp. 278–291.

[J] Jones, C.B., Specification and Design of (Parallel) Programs, ***Proceedings of IFIP,***North Holland, Paris (Nov. 1983) pp. 321–332.

[L] Lamport, L., The 'Hoare' Logic of Concurrent Programs, *Acta Informatica* **14** (1980) pp. 21–37.

[MCS] Misra, J., Chandy, K.M. and Smith, T., Proving Safety and Liveness of Communicating Processes with Examples, ***Proceeding of the ACM Conference on the Principles of Distributed Computing,*** Ottowa, Canada (Aug. 1982).

[MP1] Manna, Z. and Pnueli, A., Verification of Concurrent Programs: The Temporal Framework, in the "Correctness Problem in Computer Science", ed. R.S. Boyer and J.S. Moore, Academic Press, London 1982, pp. 215–273.

[MP2] Manna, Z. and Pnueli, A., How to Cook A Temporal Proof System for your Pet Language, ***Proc. of the ACM Symposium on Principles of Programming Languages,*** Austin, Texas, **10** (January 1983) pp. 101–154.

[MP3] Manna, Z. and Pnueli, A., Verification of Concurrent Programs: A Temporal Proof System, in "Foundations of Computer Science IV", J.W. DeBakker and J. Van Leeuwen, editors,Mathematical Centre Tracts #159, Amsterdam (1983) pp. 163–255.

[OG] Owicki, S. and Gries, D., An Axiomatic Proof Technique for Parallel Programs, *Acta Informatica* **6** (1976) pp. 319–340.

[Pn] Pnueli, A., The Temporal Semantics of Concurrent Programs, *Theoretical Computer Science,* **13** (1981) pp. 45–60.

[W] Wolper, P., Temporal Logic can be More Expressive, *22nd Annual Symp. on Foundations of Computer Science* (1981) pp. 340–347.

FORMAL MODELS IN PROGRAMMING
E.J. Neuhold and G. Chroust (Editors)
Elsevier Science Publishers B.V. (North-Holland)
IFIP, 1985

A COMPOSITIONAL TEMPORAL APPROACH TO A CSP-LIKE LANGUAGE

Howard Barringer and Ruurd Kuiper

Department of Computer Science
University of Manchester
Manchester, England

Amir Pnueli

Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel

A temporal semantics enabling the independent description of components and dealing with the liveness aspects of synchronised message based communication is developed for a CSP-like language with dynamic process creation. A closely related compositional temporal proof system for total correctness exploiting the compositional features of the semantics follows. The use of the proof system to provide program correctness proofs is exemplified on a program to compute factorials; its application in hierarchical program specification is illustrated by the development of an implementation of a queue as a dynamic network of processes.

1. INTRODUCTION

In communication based languages, as opposed to shared variable ones, interaction between parallel components is clearly distinguished from activity within each component. Various partial correctness proof systems for such languages have been developed which reflect the inherent compositional characteristics, i.e. enabling the synthesis of a proof concerning a program via independent proofs about its subcomponents. Examples of such approaches are Misra & Chandy (1981), Zhou & Hoare (1981) and Zwiers et al. (1983). There have also been efforts to extend compositional proof systems to more than just invariance properties, e.g. Lamport (1983), Hoare (1981) and Misra et al. (1982). A total correctness proof system, using temporal logic as a natural tool to handle the complicated liveness behaviour induced by synchronised communication primitives has been given by Manna & Pnueli (1983); this approach, however, does not enable composition of proofs.

The aim of this paper is to present a total correctness proof system for a CSP-like language with dynamic process creation (cf. Zwiers et al. (1983)) based on temporal logic whilst retaining compositionality. The paper, here, builds upon techniques presented in Barringer et al. (1984) where a compositional temporal system was developed for a shared variable parallel language. The central idea presented therein was to obtain compositionality by describing a component via its behaviour in any environment using an interleaved model of parallel execution. Such descriptions required distinguishing transitions made by a component from those made by its environment and hence labelled transitions were introduced into the model and transition propositions into the temporal language. There are two main problems which result from handling synchronised message based communication primitives. Firstly, two components may share an action (e.g. an internal communication) and hence a more complex transition labelling convention is required. Secondly, the rather intricate synchronisation behaviour (e.g. blocking under just and/or fair execution) demands more information than that provided by

the above sequence model. These aspects are handled here by the introduction of special transition labels (in particular, waiting sets) to the model and of appropriate transition variables to the temporal language.

In the following sections, a natural, temporal logic based, denotational semantics is developed for the CSP-like language and, based on that, a closely related compositional temporal proof system is given and exemplified. Although the more detailed restrictions that are applicable when the system is used as a hierarchical specification method are not given, its potential for that application is demonstrated by the stepwise development of a queue as a dynamic network of components.

The paper is organised as follows. Section 2 describes the CSP-like language. Section 3 introduces the temporal language used in section 4 for describing the formal semantics and used in section 5 for the proof system. Section 6 contains two examples, i) verification of a factorial program, and ii) the development of a queue. Finally, conclusions are given in section 7.

2. CSP-LIKE LANGUAGE

Communicating Sequential Processes (CSP), introduced by Hoare (1978), is a language to describe computations distributed over several parallel components which interact by means of synchronised message based communication. A similar language is chosen here, both because CSP (or one of its numerous dialects) is frequently used to illustrate other proof methods, e.g. Apt et al. (1980), Levin & Gries (1981), Lamport & Schneider (1984), Misra & Chandy (1981), Zhou & Hoare (1981) and Zwiers et al. (1983), hence facilitating comparison, and also because many other communication mechanisms can be based upon the CSP primitives, e.g. Ada rendezvous, Gerth & de Roever (1984) and Pnueli & de Roever (1982).

To indicate that the method developed here can be applied to reasonably realistic programming languages and examples, a fairly general CSP-like language is presented, including variable and channel declaration, and dynamic process creation by means of nested parallelism in recursive procedures.

To provide some intuition before entering the more formal sections, the syntax of the language is accompanied by some informal description of the language element's meaning.

Statements

S :: $u := e$	assignment
skip	
$c!e$	output command, send value e along channel c
$c?u$	input command, receive the value sent on channel c and assign to the variable u
$[\ (\square_{i \in K}\ b_i \rightarrow S_i)$	guarded choice, by purely boolean guards
$\square\ (\square_{i \in L}\ b_i ; c_i!e_i \rightarrow S_i)$	by booleans and output guards
$\square\ (\square_{i \in M}\ b_i ; c_i?u_i \rightarrow S_i)\ ]$	by booleans and input guards
$S_1 ; S_2$	sequential composition
$S_1 \Vert S_2$	parallel composition
while b **do** S **od**	while loop
begin **chan** c; **var** u; S **end**	channel and variable declaration

call $p(c_a, u_a)$ procedure call with actual channels c_a and actual variables u_a

Programs

P :: (*proc* p_i(*chan* c_{if}; *var* u_{if}) ≙ S_i); S
program P is a statement S headed by a set of possibly mutually recursive procedures p_i with formal arguments c_{if} and u_{if}

The special feature of CSP, synchronised communication, is here represented via bi-directional channels over which values may be sent as a synchronised action of two parallel processes. Consider the program fragment *chan c; S*, where S contains the parallel statement $S_1 || S_2$ with *c!e* occurring in S_1 and *c?u* occurring in S_2. Assuming the send and receive semantically match, the value of *e* is sent over the channel *c* to be assigned to the variable *u* as a simultaneous action by S_1 and S_2. This communication is synchronised in that when, for example, *c!e* is encountered in a statement, it can not be executed, i.e. passed, until the moment that, in some other parallel statement, the complementary primitive is encountered. Hence the application as guards in the guarded command: any S_i for which the corresponding boolean or boolean/communication guard can be passed may be chosen for execution. As will be seen from the formal semantics in section 4, fair interleaving is used for non-communication steps and both a form of *justice* and of *fairness* concerning channel communications are treated.

An example of the use of nested parallelism in recursive procedures is given in the following factorial function.

```
proc fact(chan in) is
begin
  var x;
  in?x;
  [ x=0 -> in!1
  □ x>0 ->
     begin
       chan t; var y;
       ( (t!x-1; t?y) || call fact(t) );
       in!x*y
     end
  ]
end.
```

3. TEMPORAL LANGUAGE

To achieve both the objectives of capturing liveness properties and ofobtaining compositionality, program behaviour will be described by means of labelled state sequences. A temporal language is the natural choice for expressing the required sequence properties. The version used here includes a *chop* or *combine* operator (C) and its iterated version (C^*) and includes the expression of maximal fixpoint solutions to temporal implications enabling straightforward treatment of, respectively, sequential composition, iteration and recursion.

In the following the temporal language and its interpretation is presented.

3.1. Basic Alphabet

The basic symbols of the language are divided as three groups.

i) local symbols, i.e. symbols whose values are state or transition dependent

$P \in P$	state propositions
$y \in Y$	state variables
$l \in L$	transition variables

ii) global symbols, i.e. symbols whose values are fixed for the complete sequence

$x \in X$	global variables

iii) constant symbols, i.e. symbols whose values are the same for all sequences

$f \in F$	function symbols
$q \in Q$	predicate symbols

Terms are constructed in the usual way from state, transition and global variables, or by the application of appropriate function symbols to terms. Atomic formulae can then be built from state propositions or by the application of predicates to terms.

The logical constants are the standard truth constants,

true, false

the standard first order logical operators,

$\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\forall$, $\exists$

the unary temporal operators

$\bigcirc$ (next time) , $\Diamond$ (eventually) , $\Box$ (always)

the binary temporal operators

$\underline{U}$ (strong until), U (weak until),
C (combine) , C^* (iterated combine)

and the maximal fixpoint operator ν.

Temporal formulae are then constructed from atomic formulae by the appropriate application of logical operators.

3.2 Interpretation over Labelled Sequences

Assuming a fixed domain, D, and fixed interpretations for the function and predicate symbols, a model M over which a temporal formula is interpreted is a 4-tuple,

$$M = (\alpha, \sigma, I, J) ,$$

where

α assigns D-values to the global variables,
σ is a non-empty finite or infinite sequence of states and transitions

$$\sigma \triangleq s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots\ldots$$

I is a state interpretation assigning D-values to each state variable and truth values {T, F} to state propositions,
J is a transition interpretation assigning D-values to each transition variable.

Given a model $M=(\alpha,\sigma,I,J)$ it is possible to define, inductively, the

interpretation of temporal formulae over M. In general, this interpretation only involves change to the sequence σ and in the following $\phi|_\sigma$ ($t|_\sigma$) abbreviates the value of the formula ϕ (term t) over the model M.

The sequence operators $<$ (proper suffix), $\leq$ (suffix) and ^{o} (fusion) are required, and are defined as below, where the length of a sequence σ is defined as the number of transitions occurring in σ.

$\sigma_1{}^o\sigma_2$ is if $\sigma_1 = s_0 \to s_1 \to \ldots s_k$, and $\sigma_2 = s_k \to s_{k+1} \to \ldots$.
then $s_0 \to s_1 \to \ldots s_k \to s_{k+1} \to \ldots$
otherwise σ_1.

$\sigma_1 < \sigma$ holds if there exists a σ_2 of length > 0 such that $\sigma = \sigma_2{}^o\sigma_1$
and $\sigma_1 \leq \sigma$ holds if $\sigma_1 < \sigma$ or $\sigma_1 = \sigma$.

Terms

$x|_\sigma \triangleq \alpha(x)$	global variables
$y|_\sigma \triangleq I(s_0, y)$	state variables
$l|_\sigma \triangleq J(t_0, l)$	transition variables
$f(t_1,\ldots,t_n)|_\sigma \triangleq F_f(t_1|_\sigma,\ldots,t_n|_\sigma)$	function applications where F_f is the fixed function value for f
$Ot|_\sigma \triangleq t|_{\sigma(1)}$ where $\sigma(1) = s_1 \to s_2 \to \ldots$	the next time operator applied to a term is the term evaluated in the suffix starting from the next state.

Atomic Formulae

$P|_\sigma \triangleq I(s_0, P)$	state propositions
$q(t_1,\ldots,t_n)|_\sigma \triangleq Q_q(t_1|_\sigma,\ldots,t_n|_\sigma)$	predicate applications where Q_q is the fixed predicate value for q

Now assuming the standard interpretation for the standard logical symbols (i.e. for **true**, **false**, $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall$), formulae ϕ constructed using the temporal operators are interpreted over $\sigma = s_0 \to s_1 \to \ldots$ as follows.

$O\phi|_\sigma$ if and only if $\phi|_{\sigma(1)}$ where $\sigma^{(1)} = s_1 \to s_2 \to \ldots$

Note that if σ is of length 0 then $O\phi|_\sigma$ is F and hence the formula $\neg O$**true** is true only at the end of the sequence σ.

$\Box\phi|_\sigma = T$ iff for all $\sigma' \leq \sigma$ $\phi|_{\sigma'} = T$.

$\Diamond\phi|_\sigma = T$ iff there is some $\sigma' \leq \sigma$ such that $\phi|_{\sigma'} = T$.

$\phi \underline{U} \psi|_\sigma = T$ iff there is some $\sigma'' \leq \sigma$ such that
a) $\psi|_{\sigma''} = T$.
and b) for all $\sigma'' < \sigma' \leq \sigma$ $\phi|_{\sigma'} = T$.

$\phi U \psi|_\sigma = T$ iff $((\phi \underline{U} \psi) \vee \Box\phi)|_\sigma = T$.

$\phi C \psi|_\sigma = T$ iff there are σ' and σ'' where $\sigma'{}^o\sigma'' = \sigma$ such that
if σ' is infinite then $\phi|_{\sigma'} = T$
otherwise both $\phi|_{\sigma'} = T$ and $\psi|_{\sigma''} = T$

$\phi C^* \psi|_\sigma = T$ iff either there are $\sigma_1, \sigma_2, \ldots, \sigma_k, \sigma_{k+1}$, $\sigma = \sigma_1{}^o\sigma_2 \ldots \sigma_k{}^o\sigma_{k+1}$
such that $\phi|_{\sigma_i} = T$ for $i \in \{1..k\}$ and $\psi|_{\sigma_{k+1}} = T$

or there are $\sigma_1, \sigma_2, \ldots$, $\sigma = \sigma_1{}^o\sigma_2{}^o\ldots$
such that for all i $\phi|_{\sigma_i} = T$

or there are $\sigma_1, \sigma_2, \ldots, \sigma_k$, $\sigma = \sigma_1 \circ \sigma_2 \circ \ldots \sigma_k$
such that σ_k is infinite and $\phi|_{\sigma_i} = T$, $i \in \{1..k\}$

$\nu \xi . \chi(\xi)|_\sigma = T$ iff for all $k > 0$, $\chi^k(true)|_\sigma = T$
where $\chi^k(true)$ is the temporal formula
$\underbrace{\chi(\chi(\ldots.\chi(true)\ldots.))}_{k\text{-times}}$

Assume the temporal formula $\chi(\xi)$ contains only positive occurrences of the temporal variable ξ, i.e. ξ occurs only under an even number of negations. The temporal formula $\nu\xi.\chi(\xi)$, in fact, then denotes the maximal fixpoint solution, with respect to implication ordering, of the implication $\xi \to \chi(\xi)$. The extension of ν to vectors is also used and some explanation is given in section 4.2 where procedure semantics are developed.

4. TEMPORAL SEMANTICS

4.1. The Computation Model

The meaning of a program *S* is taken to be the set of finite or infinite computation sequences *S* can generate in any environment. The more detailed features of a computation sequence are derived from the following considerations. The states s of the sequences σ,

$$\sigma = s_0 \to s_1 \to s_2 \to \ldots\ldots ,$$

are global in that a state assigns values to all state variables. Although CSP is restricted to message based communication only, here, the more general global states are used to be able to also express Ada-like sharing of variables, if so desired! However, non-sharing can always be obtained either by not using the same variables in different parallel components or by declaring new variables locally for each parallel component. The set of variables is infinite (to enable recursion) but for any terminating program only finitely many will be used.

The transitions between these states are labelled to achieve compositionality. The idea is that every component has its own view as to what caused a change to the program variables. Transitions can be labelled by:-

- Π indicating an internal step of a component;
- E indicating an environment step;
- c!a indicating an external send of value a on channel c
- c?a indicating an external receive of value a on channel c

Thus a sequence labelled from the view of P_1 as

$$s_0 \xrightarrow{\Pi} s_1 \xrightarrow{E} s_2 \xrightarrow{c!a} s_3 \xrightarrow{E} s_4 \xrightarrow{c?a} \ldots$$

and from the view of P_2 as

$$s_0 \xrightarrow{E} s_1 \xrightarrow{\Pi} s_2 \xrightarrow{c?a} s_3 \xrightarrow{E} s_4 \xrightarrow{E} \ldots$$

would be labelled from the view of $P_1||P_2$ as

$$s_0 \xrightarrow{\Pi} s_1 \xrightarrow{\Pi} s_2 \xrightarrow{\Pi} s_3 \xrightarrow{E} s_4 \xrightarrow{c?a} \ldots \quad .$$

This example indicates that a communication between P_1 and P_2 is an internal step of $P_1||P_2$ considered as a single component; an external communication of P_1 but not to P_2 is clearly an external communication of $P_1||P_2$ considered as a single component.

Finally, to describe blocking behaviour it is necessary to indicate what communications are on offer by a component but not (yet) satisfied by its environment. As this too is dependent upon what component's view of the system is taken, it is added as an extra label to the transition in the form of a waiting set.

So, overall, the computation sequences of a program *S* are of the form

$$\sigma = s_0 \xrightarrow{\lambda_0,\omega_0} s_1 \xrightarrow{\lambda_1,\omega_1} s_2 \xrightarrow{\lambda_2,\omega_2} \ldots\ldots \rightarrow s_i \xrightarrow{\lambda_i,\omega_i} \ldots.$$

and are called (λ,ω)-sequences. Such a sequence σ can be extended to a model $M=(\alpha,\sigma,I,J)$, as given in section 3, by taking:

- α as an arbitrary global assignment to the global (logical) variables;
- σ as the given sequence;
- I as defined by $I(s_i,y)=y(s_i)$, the value of the state variable y in state s;
- J as defined by $J(t_i,\lambda)=\lambda(t_i)$ and $J(t_i,\omega)=\omega(t_i)$ for the two distinct types of transition variable.

A model M for a program S is then called a (λ,ω)-model.

4.2. Semantic Equations

For each form of statement S in the language, its temporal semantics, $[\![S]\!]_\Theta$, is defined as a temporal formula $\phi(\lambda,\omega)$ with two free transition variables λ and ω. The meaning of ϕ is that a (λ,ω)-sequence satisfies ϕ if and only if it is an execution sequence of S. Since the language allows procedures, the meaning of a statement is dependent on some environment $\Theta = \langle\theta_1,\ldots,\theta_n\rangle$ containing the meaning of the visible procedures. Initially, a semantics providing just treatment of channel communication is given, modifications for fair treatment are considered later.

To make the descriptions a little more presentable, the following abbreviations are assumed:

$\lambda=E$ as $\underline{E}$, $\lambda=\Pi$ as $\underline{\Pi}$, $\lambda=c!a$ as $\underline{c!a}$, etc.,
$\omega=W$ as $\underline{W}$, $\omega=\emptyset$ as $\underline{\emptyset}$, where $\emptyset$ is the empty set, etc.,
$\neg\bigcirc true$ as fin
$\underline{E} \wedge \omega=W$ as $idle(W)$
$\underline{\Pi} \wedge \omega=\emptyset \wedge b \wedge \bigcirc\underline{y}=\underline{y}$ as $test(b)$
$\underline{\Pi} \wedge \omega=W \wedge \bigcirc\underline{y}=\underline{y}$ as $load(W)$
$\Box(idle(W) \wedge \neg fin)$ as $fail(W)$
$load(W) \wedge \bigcirc fail(W)$ as $blocked(W)$

<u>Assignment</u>

$$[\![u:=e]\!]_\Theta \;\hat{=}\; idle(\emptyset)\ \underline{U}\ (\underline{\Pi} \wedge \underline{\emptyset} \wedge \bigcirc\underline{y}=\underline{y}[e/u] \wedge \bigcirc fin)$$

The meaning of an assignment *u:=e*, which of course desires no communications, in any just parallel environment is thus given as some finite number of arbitrary environment transitions followed by an internal step which updates only the state variable u after which it terminates.

<u>Skip</u>

$$[\![skip]\!]_\Theta \;\hat{=}\; idle(\emptyset)\ \underline{U}\ (\underline{\Pi} \wedge \underline{\emptyset} \wedge \bigcirc\underline{y}=\underline{y} \wedge \bigcirc fin)$$

The meaning of *skip*, here, is just taken as enforcing an internal transition with no effect on the state variables.

Output Command

To enable expression of justice for the parallel construct with regard to channel communications, communication possibilities must be identified for output (and input) commands. Thus the fact that a component is blocked on a channel c is described by the continual presence of that channel in the waiting set associated with that component.

$$[\![c!e]\!]_\Theta \triangleq \mathrm{idle}(\emptyset)\ \underline{U}\ (\ (\exists a.a{=}e \wedge \underline{c!a} \wedge \underline{\emptyset} \wedge \bigcirc y{=}y \wedge \bigcirc\mathrm{fin}) \vee \mathrm{blocked}(\{c!\})\ )$$

For later use we abbreviate $\exists a.a{=}e \wedge \underline{c!a} \wedge \underline{\emptyset} \wedge \bigcirc y{=}y$ as $\mathrm{send}(c,e)$

Input Command

$$[\![c?u]\!]_\Theta \triangleq \mathrm{idle}(\emptyset)\ \underline{U}\ (\ \mathrm{receive}(c,u) \wedge \bigcirc\mathrm{fin} \vee \mathrm{blocked}(\{c?\})\ )$$

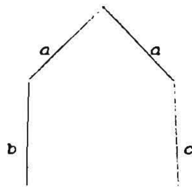
Here, $\mathrm{receive}(c,u)$ abbreviates the temporal formula $\exists a.\underline{c?a} \wedge \underline{\emptyset} \wedge \bigcirc y{=}y[a/u]$.
The semantics is clearly analogous to the output command.

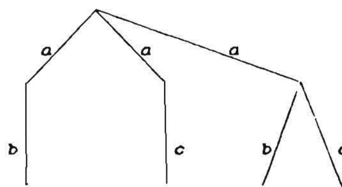
Guarded Choice

Although full abstractness is not an aim of this semantics, it is desirable to avoid distinguishing certain blocked behaviours which can not in reality be distinguished. Consider the following example from de Nicola & Hennessy (1983). Abbreviating communications by a, b, and c:-

P :: [a -> b □ a -> c]
Q :: [a -> b □ a -> c □ a -> [b -> *skip* □ c -> *skip*]]

i.e. P Q





Assuming communication a has occurred, P allows sequences blocked with the waiting sets $\{b\}$ or $\{c\}$, whereas Q allows the waiting sets $\{b\}$, $\{c\}$ or $\{b,c\}$. However, no program context can distinguish P from Q. For the programs to have the same semantics, extra sequences are incorporated which are blocked on supersets of the original waiting sets. Thus the following semantics is obtained for the guarded choice.

$$[\![\,[(\Box_{i\in K}\, b_i \to S_i)\ \Box\ (\Box_{i\in L}\, b_i; c_i!e_i \to S_i)\ \Box\ (\Box_{i\in M}\, b_i; c_i?e_i \to S_i)]\,]\!]_\Theta \triangleq$$

$$\begin{array}{l} \mathrm{idle}(\emptyset)\ \underline{U} \\ \quad (\ (\exists i\in K.\mathrm{test}(b_i) \wedge \bigcirc[\![S_i]\!]_\Theta) \vee \\ \quad\ \ (\exists i\in L.b_i \wedge \mathrm{send}(c_i,e_i) \wedge \bigcirc[\![S_i]\!]_\Theta) \vee \\ \quad\ \ (\exists i\in M.b_i \wedge \mathrm{receive}(c_i,u_i) \wedge \bigcirc[\![S_i]\!]_\Theta) \vee \\ \quad\ \ (\forall i\in K.\neg b_i) \wedge \exists S.\mathrm{blocked}(C \cup S)\ ) \end{array}$$

where C is the set $\bigcup_{b_i, i\in L} c_i! \ \cup \bigcup_{b_i, i\in M} c_i?$

The semantics asserts some arbitrary initial amount of environment action followed by either a successful evaluation of a guard (the first three disjuncts) and thence execution of the corresponding statement, or a blocked situation.

Sequential Composition

$$[\![S_1;S_2]\!]_\Theta \triangleq [\![S_1]\!]_\Theta \; C \; [\![S_2]\!]_\Theta$$

The reason that the combine operator (C) was chosen to be weak is that should a computation of S_1 diverge, S_2 will not, of course, be executed.

Parallel Composition

The essence of parallel composition is that sequences of $S_1 || S_2$ are sequences, by appropriate relabelling of transitions, of both S_1 and S_2. Furthermore, to illustrate how liveness conditions can be incorporated in the parallel semantics, component justice is treated here (and later extended to fairness). The intuitive meaning of component justice is the following:

> no two components can be simultaneously blocked if there is a channel along which they could communicate with each other.

The waiting set, as already introduced, is used as the minimal additional transition information necessary to describe this liveness condition in the semantics; it is only added to sequences in the case that there is a blocked component.

These notions are captured by the following temporal operator **par**.

$$
\begin{array}{ll}
(\phi_1 \;\mathbf{par}\; \phi_2) \triangleq & \\
\quad \exists \lambda_1, \omega_1, \lambda_2, \omega_2 . & \\
\qquad (\; \phi_1(\lambda_1, \omega_1) \wedge \phi_2(\lambda_2, \omega_2) \wedge & (1)\\
\qquad\quad \Box((\lambda = \Pi \Rightarrow (\lambda_1 = \Pi \wedge \lambda_2 = E \;\vee\; \lambda_1 = E \wedge \lambda_2 = \Pi \;\vee & (2)\\
\qquad\qquad\qquad \exists c, a.(\lambda_1 = c!a \wedge \lambda_2 = c?a \;\vee\; \lambda_1 = c?a \wedge \lambda_2 = c!a)) \wedge & \\
\qquad\qquad (\lambda = E \Rightarrow \lambda_1 = \lambda_2 = E) \;\wedge & (3)\\
\qquad\qquad (\lambda \notin \{\Pi, E\} \Rightarrow (\lambda = \lambda_1 \wedge \lambda_2 = E \;\vee\; \lambda_1 = E \wedge \lambda = \lambda_2)) \;\wedge & (4)\\
\qquad\qquad (\omega = \omega_1 \cup \omega_2)\;) \;\wedge & (5)\\
\qquad\quad \mathrm{just})\;) &
\end{array}
$$

$$\text{where } \mathrm{just} \triangleq \neg \exists c . \Diamond\Box(\{c!, c?\} \subseteq \omega)$$

The intuition behind this semantic parallel operator **par** is as follows. Provided the arguments ϕ_1 and ϕ_2 describe infinite sequences, it is claimed that (ϕ_1 **par** ϕ_2) describes exactly the set of sequences obtained by the parallel composition of ϕ_1 and ϕ_2. Line 1 states that a sequence of (ϕ_1 **par** ϕ_2) can be relabelled as both a sequence of ϕ_1 and of ϕ_2 in such a way, lines 2 to 5, that is consistent with an interleaving semantics with synchronised communication steps. Line 2 gives the relabelling for internal actions, line 3 gives the relabelling for environment actions, line 4 gives the relabelling for external communications and line 5 is the relabelling of blocked sequences. The last line ensures the sequences are just.

As in this semantics finite computations are described by finite sequences, the above operator is used as follows to describe the semantics of parallel composition.

$$
\begin{array}{rl}
[\![S_1 || S_2]\!]_\Theta \triangleq & (\; [\![S_1]\!]_\Theta \; C \; (\Box(\mathrm{idle}(\emptyset) \wedge \neg \mathrm{fin})) \;\mathbf{par} \\
& \quad [\![S_2]\!]_\Theta \; C \; (\Box(\mathrm{idle}(\emptyset) \wedge \neg \mathrm{fin})) \;) \;\wedge \\
& \neg\Diamond\Box\underline{E} \wedge (\Diamond \mathrm{fin} \Rightarrow \Diamond(\underline{\Pi} \wedge \bigcirc \mathrm{fin}))
\end{array}
$$

All component sequences are extended to infinity with idle steps (lines 1 and 2), and then after application of the parallel operator **par** unnecessary idle steps are removed (line 3).

While Loop

$$[\![\textit{while } b \textit{ do } S \textit{ od}]\!]_\Theta \ \hat{=}\ (\mathrm{idle}(\emptyset)\ \underline{U}\ (\mathrm{test}(b) \wedge \bigcirc[\![S]\!]_\Theta))\ C^*$$
$$(\mathrm{idle}(\emptyset)\ \underline{U}\ (\mathrm{test}(\neg b) \wedge \bigcirc\mathbf{fin}))$$

By virtue of the iterated combine operator, the meaning of the while statement can be expressed directly without the use of fixpoints. The three cases of C^* correspond to finite iteration and termination of the loop, to finite iteration but divergence of the body S on the last iteration, and finally to infinite looping.

Channel and Variable Declaration

$$\begin{aligned}[\![\textit{begin chan } c;\ \textit{var } u;\ S\ \textit{end}]\!]_\Theta \ \hat{=}\ & \\ (\ \exists\omega_1, c, u.(\ & [\![S]\!]_\Theta(\lambda,\omega_1) \wedge \\ & \Box\mathrm{PrVar}(\underline{E},u) \wedge \\ & \Box\neg\exists a.(\underline{c!a} \vee \underline{c?a}) \wedge \\ & \Box(\omega = \omega_1 \backslash \{c!, c?\})\ ) \wedge \\ \Box\mathrm{PrVar}(\neg\underline{E},u)\ )\end{aligned}$$

$$\text{where } \mathrm{PrVar}(X,u) \ \hat{=}\ X \rightarrow \bigcirc u{=}u$$

The meaning of a local channel declaration is that a new channel is introduced (globally) in such a way that internal communications by S may occur but no external communications from S to the environment may occur. This requires that such sequences where S communicates (or attempts to) over the channel c to its environment are eliminated (lines 3 and 4). Similarly, a local variable declaration restricts access to the variable to the statement S. This is achieved by eliminating those sequences where the environment changes the variable (line 2). Furthermore a non-local variable of the same name can not be changed by the component S (line 5).

Procedure Call

$$[\![\textit{call } p_i(c_a,u_a)]\!]_\Theta \ \hat{=}\ \mathrm{idle}(\emptyset)\ \underline{U}$$
$$(\underline{\Pi} \wedge \omega{=}\emptyset \wedge \bigcirc\underline{\gamma}{=}\underline{\gamma} \wedge \bigcirc\Theta_i(c_a,u_a))$$

The environment vector Θ provides the meanings of all the procedures in the program. Thus the meaning of a call of some procedure p_i with actual arguments c_a and u_a is given by the ith component of the vector having replaced its free state variables (i.e. the procedure's formal arguments) by the actual arguments. The parameter mechanism is thus call by reference. At least one component (idle) step is forced thus ensuring that the semantics of a call of procedure p defined as *proc p **is** call p **end*** yields a divergent result.

Program (Procedure Declarations)

To facilitate the understanding of procedure declarations we first present the semantics of a program with just one procedure declaration.

$$[\![\textit{proc } p_1(\textit{chan } c_f;\ \textit{var } u_f)\ \textit{is } S_1\ \textit{end};\ S]\!] \ \hat{=}\ [\![S]\!]_\Theta$$

$$\text{where } \Theta \ \hat{=}\ \langle \nu\iota.[\![S_1]\!]_{\langle\iota\rangle}\rangle$$

The meaning of S is dependent upon the meaning the procedure p_i contained in the procedural environment Θ. Following the usual practice in denotational semantics, the meaning of the recursive procedure p_1 is given as a fixpoint of the inequality $\xi \sqsubseteq [S_1]_{\langle\xi\rangle}$; here the ordering $\sqsubseteq$ is an implication ordering on temporal formulae, i.e.

$$\phi \sqsubseteq \psi \text{ if and only if } \phi \dashv \psi,$$

and maximal fixpoints are used; note the minimal fixpoint in this ordering is trivially false. The implication ordering is used to remain in step with set containment.

By the continuity of the temporal implications of the form $\xi \dashv \chi(\xi)$ when only positive occurrences of ξ exist in χ, it follows that $\nu\xi.\chi(\xi)$ (i.e. $\bigwedge_i \chi^i(\textbf{true})$) is the maximal fixpoint of the implication.

The given semantics can easily be extended to handle programs headed by a collection of mutually recursive procedures, cf. de Bakker (1980) and Manna (1974). It requires taking fixpoints over a collection of n temporal implications

$$\langle \Xi_i \dashv \chi_i(\Xi) \mid i=1..n \rangle, \text{ where } \Xi \text{ is a vector}$$

i.e. to obtain the maximal fixpoint solution of the inequality

$$\Xi \sqsubseteq \langle \chi_1, \ldots, \chi_n \rangle(\Xi)$$

using the obvious vector extension of the implication ordering.

Furthermore ν is extended to vectors in the following way. $\nu\Xi.\langle\chi_1,\ldots,\chi_n\rangle_i(\Xi)$ is defined as the vector of temporal formulae

$$\langle \bigwedge_i ((\Xi^i)_j) \mid j=1..n \rangle$$

where the Ξ_i are defined by:-

$$\Xi^0 \triangleq \langle \textbf{true}, \underset{n\text{-times}}{\ldots} \textbf{true} \rangle$$

$$\Xi^i \triangleq \langle \chi_1(\Xi^{i-1}), \ldots, \chi_n(\Xi^{i-1}) \rangle$$

and $(\Xi^i)_j$ is the jth element of the vector.

Under similar restrictions of only positive occurrences of the ξ_j in the formulae χ_i, the vector of infinite conjunctions is the maximal fixpoint of the above inequality. Thus the following is obtained.

$$[\![(\textit{proc } p_i(\textit{chan } c_f; \textit{ var } u_f) \textit{ is } S_i \textit{ end};)^i \; S]\!] \triangleq [\![S]\!]_\Theta$$

$$\text{where } \Theta = \nu\Xi.\langle [\![S_i]\!]_\Xi \mid i=1..n \rangle$$

4.3 Fairness

Stronger and more intricate liveness requirements than component justice can be described in a similar way to above by adding more transition information as necessary. This is demonstrated through the notion of channel communication fairness:

> communication along a channel is not indefinitely postponed if on one end a component is continuously trying to communicate and on the other end the other component infinitely often tries the matching communication.

For example, input continuously offered to a buffer, dividing its attention fairly between input and output channels, will eventually be accepted.

To handle this fairness notion in the semantics, communication attempts by non-blocked components must be visible; this is achieved, below, by loading the

waiting set with these attempted communications.

The semantics for output command thus becomes:

$$
\begin{aligned}
[\![c!e]\!]_\Theta \;\hat{=}\; \mathrm{idle}(\emptyset)\; \underline{U}\; (\; & \mathrm{load}(\{c!\}) \wedge \\
& \bigcirc(\; (\mathrm{idle}(\{c!\})\; \underline{U}\; (\mathrm{send}(c,e) \wedge \bigcirc\mathrm{fin})) \;\vee \\
& \quad \mathrm{fail}(\{c!\})\;)\;)
\end{aligned}
$$

and similarly for input. For guarded choice the following is obtained

$$
[\![[(\mathop{\square}_{i\in K} b_i \to S_i) \;\square\; (\mathop{\square}_{i\in L} b_i;c_i!e_i \to S_i) \;\square\; (\mathop{\square}_{i\in M} b_i;c_i?e_i \to S_i)]]\!]_\Theta \;\hat{=}
$$

$$
\begin{aligned}
& \mathrm{idle}(\emptyset)\; \underline{U} \\
& \;\exists S.(\mathrm{load}(C \cup S) \wedge \\
& \qquad \bigcirc(\mathrm{idle}(C \cup S)\; \underline{U} \\
& \qquad\qquad (\; (\exists i\in K.\mathrm{test}(b_i) \wedge \bigcirc[\![S_i]\!]_\Theta) \;\vee \\
& \qquad\qquad\;\; (\exists i\in L.b_i \wedge \mathrm{send}(c_i,e_i) \wedge \bigcirc[\![S_i]\!]_\Theta) \;\vee \\
& \qquad\qquad\;\; (\exists i\in M.b_i \wedge \mathrm{receive}(c_i,u_i) \wedge \bigcirc[\![S_i]\!]_\Theta) \;\vee \\
& \qquad\qquad\;\; ((\forall i\in K.\neg b_i) \wedge \mathrm{fail}(C \cup S))\;)\;)\;)
\end{aligned}
$$

where C is the set $\bigcup_{b_i, i\in L} c_i! \;\cup\; \bigcup_{b_i, i\in M} c_i?$

and finally by changing "just" in the **par** operator to "fair" given below, a **fpar** operator can be obtained.

$$
\mathrm{fair} \;\hat{=}\; \forall c.((\Diamond\square c!\in\omega_i \wedge \square\Diamond c?\in\omega_{\neg i}) \Rightarrow \square\Diamond\exists a.(\lambda_i = c!a \wedge \lambda_{\neg i} = c?a))
$$

for $i=1,2$ (with $\neg i$ as the complement of i).

5. A TEMPORAL PROOF SYSTEM

In this section a compositional temporal proof system for the CSP-like language is given. The soundness and relative completeness with respect to the temporal semantics given in the previous section is easily argued.

Notation

In the following the notation $\vdash (S)_\Theta\; \phi$ is used to denote that the temporal formula ϕ is a specification of the program (statement) S under the assumption of a procedural environment Θ, i.e., in terms of the semantics, all the (λ,ω)-sequences of the statement S are (λ,ω)-models of the specification ϕ. This connection between semantics and the proof system is given by the following rule.

$$
\frac{\vdash (S)_\Theta\; \phi}{\vdash [\![S]\!]_\Theta \Rightarrow \phi}
$$

Rule of Consequence

$$
\frac{\vdash (S)_\Theta\; \phi \qquad \vdash \phi \Rightarrow \psi}{\vdash (S)_\Theta\; \psi}
$$

The soundness of this rule is as follows. The second premise ensures that all models of ϕ are also models of ψ and hence as by the first premise all execution sequences of S are models of ϕ they must also be models of ψ.

Axiom of Assignment

$$\vdash \{u := e\}_\Theta\ (idle(\emptyset)\ \underline{U}\ (\underline{\Pi} \wedge \underline{\emptyset} \wedge \bigcirc \underline{y}=\underline{y}[e/u] \wedge \bigcirc fin))$$

The soundness follows trivially from the assignment semantics.

In fact, as the rules given below, apart from the procedure rule, follow the semantics directly, their soundness is obvious.

Input and Output Axioms

$$\vdash \{c!e\}_\Theta\ (idle(\emptyset)\ \underline{U}\ (send(c,e) \wedge \bigcirc fin \ \vee\ blocked(\{c!\})))$$
$$\vdash \{c?u\}_\Theta\ (idle(\emptyset)\ \underline{U}\ (receive(c,u) \wedge \bigcirc fin \ \vee\ blocked(\{c?\})))$$

Guarded Choice (GC)

$$\frac{\vdash \{S_i\}_\Theta\ \phi_i \quad ,\ i \in K \cup L \cup M}{\begin{array}{l} \vdash \{GC\}_\Theta\ [idle(\emptyset)\ \underline{U} \\ \quad (\ (\exists i \in K.test(b_i) \wedge \bigcirc\phi_i) \ \vee \\ \quad\ \ (\exists i \in L.b_i \wedge send(c_i,e_i) \wedge \bigcirc\phi_i) \ \vee \\ \quad\ \ (\exists i \in M.b_i \wedge receive(c_i,u_i) \wedge \bigcirc\phi_i) \ \vee \\ \quad\ \ (\forall i \in K.\neg b_i) \wedge \exists S.blocked(\bigcup_{b_i, i \in L} c_i! \ \cup\ \bigcup_{b_i, i \in M} c_i? \cup S)\ )] \end{array}}$$

where GC abbreviates the syntactic form of the guarded command given in section 2.

Concatenation

$$\frac{\vdash \{S_i\}_\Theta\ \phi_i \quad ,\ i=1,2}{\vdash \{S_1;S_2\}_\Theta\ (\phi_1\ C\ \phi_2)}$$

While Rule

$$\frac{\vdash \{S\}_\Theta\ \phi}{\begin{array}{l} \vdash \{\textit{while } b \textit{ do } S \textit{ od}\}_\Theta\ [(idle(\emptyset)\ \underline{U}\ (test(b) \wedge \bigcirc\phi))\ C^* \\ \qquad\qquad\qquad\qquad (idle(\emptyset)\ \underline{U}\ (test(\neg b) \wedge fin))] \end{array}}$$

Parallel

$$\frac{\vdash \{S_i\}_\Theta\ \phi_i \quad ,\ i=1,2}{\begin{array}{l} \vdash \{S_1 \| S_2\}_\Theta\ [(\ (\phi_1\ C\ \Box(idle(\emptyset) \wedge \neg fin))\ par \\ \qquad\qquad\quad (\phi_2\ C\ \Box(idle(\emptyset) \wedge \neg fin))\ ) \wedge \\ \qquad\quad \neg\Diamond\Box\underline{E} \wedge (\Diamond fin \rightarrow \Diamond(\underline{\Pi} \wedge \bigcirc fin))\ \ ] \end{array}}$$

Channel and Variable Declarations

$$\frac{\vdash \{S\}_\Theta\ \exists c,u.(\Box PrVar(\underline{E},u) \wedge \Box PrChan(c) \wedge \phi) \qquad c,u \notin free(\phi)}{\vdash \{\textit{begin chan } c;\ \textit{var } u;\ S\ \textit{end}\}_\Theta\ (\phi \wedge \Box PrVar(\neg\underline{E},u))}$$

where $PrChan(c) \triangleq \Box((\neg\exists a.\underline{c!a} \vee \underline{c?a}) \wedge \{c!,c?\} \cap w=\emptyset)$

If ϕ, under the assumption that the environment preserves u and no external

communications are made on the newly declared channel c, then Φ (and the fact that an external variable u is preserved by the component) holds for the declaration.

Procedure Call

$$\vdash \{\ \textit{call}\ P_i(c,u)\ \}_\Theta\ \ \text{idle}(\emptyset)\ \underline{U}\ (\underline{\Pi} \wedge \omega\text{-}\emptyset \wedge \bigcirc\underline{y}\text{-}\underline{y} \wedge \bigcirc\Theta_i(c,u))$$

The procedural environment Θ provides a specification of all the procedures declared in the program. Thus the meaning of an actual call is simply obtained by accessing the appropriate specification.

Program Declaration

$$\frac{[\ \vdash \Xi_i \dashv [\![S_i]\!]_\Xi\ ]_{i\in 1..n}\ \text{implies}\ [\ \vdash \Xi_j \dashv \Theta_j\ ]_{j\in 1..n} \qquad \vdash \{\ S\ \}_\Theta\ \ \Phi}{\vdash \{\ \textit{proc}\ p_i(\ldots)\ \textit{is}\ S_i\ \textit{end};\ S\ \}\ \Phi}$$

The semantics of procedures being defined by maximal fixpoints, this rule states that if every fixpoint is sufficient to prove the environment Θ and if with this environment Φ can be proven of the statement S, then Φ must hold S executed in the associated procedural context.

5.1. Completeness of the System

Completeness of the above system follows by induction on the structure of the statements as for each statement the temporal language can express the semantics in a closed form. The inclusion of the fixpoint operator renders the temporal language expressive enough to provide the closed form for procedure calls.

6. WORKED EXAMPLES

6.1. Proof of a Recursive Factorial

As a first example in the use of the above proof system, a proof of the factorial function given earlier in section 2 is presented. The proof follows traditional approaches to verifying existing programs against specifications.

It is desired to prove the following

$$\vdash \{\ \textit{proc fact}\ \ldots\ \textit{end};\ \ \textit{call fact(in)}\ \}\ \Phi(\text{in})$$

where

$$\begin{aligned}\Phi(\text{in}) \triangleq [\ & \text{idle}(0)\ \underline{U}\ (\exists a.\underline{\text{in?}a} \vee \text{blocked}(\{\text{in?}\}))\ \wedge \\ & \Box\forall a.(\ \underline{\text{in?}a} \wedge a\geq 0 \dashv \\ & \quad (\underline{E} \vee \underline{\Pi})\ \underline{U}\ (\underline{\text{in!}a!} \wedge \bigcirc\text{fin}\ \vee\ \text{blocked}(\{\text{in!}\})))\ ]\end{aligned}$$

Informally this specification states that once fact has been called, it will attempt to input any value a on channel in. If it succeeds, then there will be no external communication (on that channel) until eventually it either manages to output $a!$ on channel in or becomes blocked. The success of that communication is of course dependent on the environment being willing to participate!

The following predicates are defined:-

$$\begin{aligned}\text{input(in,u)}\ \triangleq\ & \text{idle}(\emptyset)\ \underline{U} \\ & ((\exists a.(\underline{\text{in?}a} \wedge \bigcirc y\text{-}y\dagger[a/u] \wedge \bigcirc\text{fin}) \vee \text{blocked}(\{\text{in?}\}))\end{aligned}$$

$$output(in,a) \triangleq idle(\emptyset)\ \underline{U}\ ((\underline{in!a} \wedge \bigcirc fin) \vee blocked(\{in!\}))$$

which correspond to the semantics of receiving and sending. The specification Φ then becomes:–

$$\begin{array}{l} \Phi(in) \triangleq [\ idle(0)\ \underline{U}\ (\exists a.\underline{in?a} \vee blocked(\{in?\})) \ \wedge \\ \quad \Box\forall a.(\ \underline{in?a} \wedge a \geq 0 \Rightarrow \\ \quad\quad (\underline{E} \vee \underline{\Pi})\ \underline{U}\ (output(in,a!)\ )\ ] \end{array}$$

The proof proceeds as follows.

1. $\vdash \xi \Rightarrow [\![B]\!]_{\Xi}$ Assumption from Program Declaration rule
   B is the body of *fact*, and $\Xi \rightarrow \langle\xi\rangle$.

To prove the premise of the Program Declaration rule, it is now shown that $\xi(in)$ implies $\Phi(in)$ by induction on the value of a. Define

$$\begin{array}{l} \Psi(in,k) \triangleq \xi(in) \Rightarrow \\ \quad [\ idle(0)\ \underline{U}\ (\exists a.\underline{in?a} \vee blocked(\{in?\})) \ \wedge \\ \quad \Box\forall a.(\ \underline{in?a} \wedge a{=}k \geq 0 \Rightarrow \\ \quad\quad (\underline{E} \vee \underline{\Pi})\ \underline{U}\ output(a!)\ )\ ] \end{array}$$

The basis for this induction is given by considering the semantics of the body of the factorial function for the case k=0 in 1. This gives step 2 below.

2. $\vdash \xi(in) \Rightarrow [\ idle(0)\ \underline{U}\ (\exists a.\underline{in?a} \vee blocked(\{in?\})) \ \wedge$
   $\Box\forall a.(\ \underline{in?a} \wedge a{=}k{=}0 \Rightarrow$
   $(\underline{E} \vee \underline{\Pi})\ \underline{U}\ output(1)\ )\ ]$

Now, by assuming $\vdash \Psi(in,k)$ (induction hypothesis), that $\vdash \Psi(in,k+1)$ holds. This follows by considering the expansion of the body of the factorial function.

3. $\vdash \{\ in?x\ \}_{\Xi}\ \ idle(\emptyset)\ \underline{U}\ (\exists a.\underline{in?a} \vee blocked(\{in?\}))$
   by Input and Consequence rules

4. $\vdash \{\ t!x{-}1;\ t?y\ \}_{\Xi}\ x{=}k{+}1 \geq 0 \wedge \Box PrVar(E,x,y) \Rightarrow [\ output(t,k)\ C\ input(t,y)\ ]$
   Output, Input and Sequential Composition

5. $\vdash \{\ \mathbf{call}\ fact(t)\ \}_{\Xi}\ \xi(t)$
   Procedure Call rule

6. $\vdash \{\ \mathbf{call}\ fact(t)\ \}_{\Xi}\ idle(\emptyset)\ \underline{U}\ (\exists a.t?a \vee blocked(\{t?\})) \wedge$
   $\Box\forall a.(\underline{t?a} \wedge a{=}k \geq 0 \Rightarrow [(\underline{E} \vee \underline{\Pi})\ \underline{U}\ output(t,a!)])$
   from 5. and the induction hypothesis $\vdash \Psi(t,k)$

7. $\vdash \{\ (t!x{-}1;\ t?y) \| \mathbf{call}\ fact(t)\ \}_{\Xi}$
   $k \geq 0 \wedge x{=}k{+}1 \wedge \Box PrVar(x,y) \Rightarrow [(\underline{E} \vee \underline{\Pi})\ U\ (y{=}k! \wedge fin)]$
   from 4. and 6. with the (*just*) Parallel rule.

8. $\vdash \{\ ((t!x{-}1;\ t?y) \| \mathbf{call}\ fact(t));\ in!x{*}y\ \}_{\Xi}$
   $k \geq 0 \wedge x{=}k{+}1 \wedge \Box PrVar(x,y) \Rightarrow [(\underline{E} \vee \underline{\Pi})\ U\ output(in,(k{+}1)!)]$
   from 7., Output Rule and Sequential Composition

9. $\vdash \{\ B\ \}_{\Xi}\ \ [\ idle(0)\ \underline{U}\ (\exists a.\underline{in?a} \vee blocked(\{in?\})) \ \wedge$
   $\Box\forall a.(\ \underline{in?a} \wedge a{=}k{+}1 \geq 0 \Rightarrow (\underline{E} \vee \underline{\Pi})\ \underline{U}\ output(in,a!)\ )\ ]$
   from 3., 8., Input, Guarded Choice and Sequential Composition and temporal reasoning

10. $\vdash [\![B]\!]_{\Xi} \Rightarrow [$ idle(0) $\underline{U}$ (∃a.$\underline{\text{in?a}}$ ∨ blocked({in?})) ∧
□∀a.($\underline{\text{in?a}}$ ∧ a=k+1≥0 ⇒ ($\underline{E}$ ∨ $\underline{\Pi}$) $\underline{U}$ output(in,a!))]
by Semantics rule

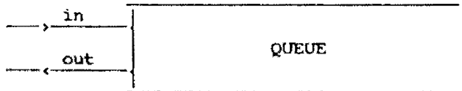
11. $\vdash \Psi(in,k)$
by 1., 10. and Induction.

12. $\vdash$ { **proc** fact(in) .... **end**; **call** fact(in) } Φ(in)
by Program Declaration rule

6.2 Development of a Recursive Queue Network

In this section a general queue specification is developed towards a recursive queue network.

It is desirable to specify systems in a black box fashion. For a queue, this means specifying it solely in terms of its input and output interface; here, it is chosen that this interface is in terms of channels.



Naturally, the specification should not refer to any "innards" of the box. As has been previously shown (Sistla et al. (1982)) pure linear time temporal logic (past or future time) is not expressive enough to be able to capture the behaviour of an unbounded FIFO queue which has non unique messages. There are at least two ways to overcome this problem in expressiveness. It is possible to introduce auxiliary variables into the interface state; such auxiliaries could be used to mimic the behaviour of a particular queue. Alternatively, the logic can be strengthened sufficiently to be able to capture the desired properties.

The approach adopted here may seem a mixture of both the above techniques. Firstly, the temporal language used here includes recursively defined temporal predicates, in a similar manner to Wolper (1981). Secondly, such predicates are allowed to have i) "call be reference" arguments to refer to actual interface state variables, and ii) "call by value/name" arguments to act as auxiliaries for the recursion. Providing such a system is then defined using such predicates over only the externally visible, i.e. interface, parts of the system, specifications are claimed to be fully abstract, i.e. possess no implementation bias.

The first specification of a FIFO queue having input channel *in* and output channel *out* is defined recursively, as below.

queue(in, out, c) ≙
waitfor({in?,out!}) C
∃a.[$\underline{\text{in?a}}$ ∧ ○queue(in, out, c^<a>))
∨
$\underline{\text{out!(hd(c),f)}}$ ∧ c≠<> ∧ ○queue(in, out, tl(c))
∨
$\underline{\text{out!(Ø,t)}}$ ∧ c=<> ∧ ○closedown]

where:-
waitfor(K) ≙ □($\underline{E}$ ∨ $\underline{\Pi}$) ∧ ◇□(ω=K)
closedown ≙ idle(Ø) $\underline{U}$ **fin**

Thus specification of an initially empty queue with channels *in* and *out* is given by the maximal fixpoint solution to the above equation applied with arguments as follows

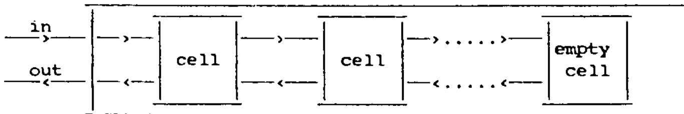
$$[\nu q.\chi_q(q)](in, out, \langle\rangle)$$

where χ_q is the righthand side of the equation. For use in later formulae, the fixpoint solution is abbreviated to F_q.

A queue behaviour is such that either the queue never performs an external communication or immediately after it performs some transput the system behaves like an "updated" queue. Although it appears at first sight that contents is mimicing the real contents, this value is only used to partition behaviour sequences into subsequences from which the appropriate queue behaviour holds.

The fairness behaviour of this queue is such that if no communication occurs it is not the fault of the queue (because it was eventually trying continuously to communicate) but it is because the user of the queue did not try hard enough (i.e. infinitely often).

Rather than implement a queue in the manner directly suggested by the above queue specification, i.e. a recursive queue procedure with an internal variable holding the contents, the specification is developed towards an implementation as a dynamic chain of 1-place cells, following the intuition below.



As values go into the queue so the chain grows,
and as values come out so the chain shrinks.

This system is specified by the solution to two mutually recursive temporal equations. Informally, the first equation describes an empty cell and the second equation describes a one place cell.

```
empty(in,out) ≙
  waitfor({in?,out!}) C
  [ ∃a.(in?a ∧
        ○[∃in1,out1.(cell(in,out,in1,out1,a) fpar empty(in1,out1))] C
        empty(in,out)) ∨
    out!(∅,t) ∧ ○closedown ]
cell(in,out,in1,out1,v) ≙
  waitfor({in?,out!}) C
  [ ∃a.(in?a ∧
        ○[waitfor({in1!}) C (in1!a ∧ ○cell(in,out,in1,out1,v))]) ∨
    out!(v,f) ∧
        ○[waitfor({out1?}) C
          ( out1?(∅,t) ∧ ○closedown ∨
            ∃a.(out1?(a,f) ∧ ○cell(in,out,in1,out1,a))] ]
```

It is asserted that the solution to the equation for empty represents the behaviour of an initially empty queue. As in the case of the queue above, F_e and F_c are used to denote the maximal fixpoint solutions to simultaneous equations. I.e.,

$F_e \triangleq [\nu\langle e,c\rangle.\langle \chi_e,\chi_c\rangle(\langle e,c\rangle)]_1$
and $F_c \triangleq [\nu c.\chi_c(c)]$

Therefore, to establish, the correctness of the above development, it must be proven that

$F_e(in,out) \dashv F_q(in,out,\langle\rangle)$ denoted by $\phi(\langle F_e,F_c\rangle,F_q)$

The proof proceeds by stepwise computational induction, see Manna (1974) for details and restrictions in application.

To make the induction possible, it is necessary to prove a stronger assertion. The following is established simultaneously with the above desired result.

$$\forall a,c.(\ [\exists in1,out1.(F_c(in,out,in1,out1,a)\ \textbf{fpar}\ F_q(in1,out1,c))]\ C\ F_e(in,out)) \dashv F_q(in,out,\langle a\rangle\hat{}c)\ )$$

denoted by $\psi(\langle F_e,F_c\rangle,F_q)$.

Intuitively, this clause establishes that a cell with element a connected to a queue with contents c behaves like a queue with contents $\langle a\rangle\hat{}c$.

Thus $(\phi\wedge\psi)(\langle F_e,F_c\rangle,F_q)$ is the assertion to be proved. It is necessary to establish

(1) (Basis) $(\phi\wedge\psi)(\langle\mathbf{true},\mathbf{true}\rangle,\mathbf{true})$

(2) (Induction step) $\forall f_e,f_c,f_q.((\phi\wedge\psi)(\langle f_e,f_c\rangle,f_q) \dashv (\phi\wedge\psi)(\langle\chi_e,\chi_c\rangle(\langle f_e,f_c\rangle),\chi_q(f_q)))$

to conclude that $(\phi\wedge\psi)(\langle F_e,F_c\rangle,F_c)$ holds.

Clearly (1) holds.

To prove (2) assume $(\phi\wedge\psi)(\langle f_e,f_c\rangle,f_q)$ holds.

$(\phi\wedge\psi)(\langle\chi_e,\chi_c\rangle(\langle f_e,f_c\rangle),\chi_q(f_q))$ expands to

$$\begin{array}{l}
1.\ (\mathrm{waitfor}(\{in?,out!\})\ C \\
\quad [\ \exists a.(\underline{in?a}\ \wedge \\
\qquad \bigcirc[\exists in1,out1.(f_c(in,out,in1,out1,a)\ \textbf{fpar}\ f_e(in1,out1))]\ C \\
\qquad f_e(in,out))\ \vee \\
\quad \underline{out!(\emptyset,t)}\ \wedge \bigcirc closedown\ ]) \\
\dashv \\
\quad (\mathrm{waitfor}(\{in?,out!\})\ C \\
\quad \exists a.[\ \underline{in?a}\ \wedge \bigcirc f_q(in,\ out,\ \langle a\rangle))\ \vee \\
\qquad \underline{out!(hd(\langle\rangle),f)}\ \wedge \langle\rangle\neq\langle\rangle\ \wedge \bigcirc f_q(in,\ out,\ tl(\langle\rangle))\ \vee \\
\qquad \underline{out!(\emptyset,t)}\ \wedge \langle\rangle=\langle\rangle\ \wedge \bigcirc closedown\ ])
\end{array}$$

$\wedge$

$$\begin{array}{l}
\forall a,c.(\ [\exists in1,out1. \\
\qquad (\chi_c(f_c)(in,out,in1,out1,a) \\
\qquad \textbf{fpar} \\
\qquad \chi_q(f_q)(in1,out1,c))]\ C \\
\qquad \chi_e(f_e)(in,out)) \\
\quad \dashv \\
\qquad \chi_q(f_q)(in,out,\langle a\rangle\hat{}c)\ ).
\end{array}$$

Clearly, the first conjunct of 1. holds if the following holds.

2. $[\exists in1,out1.(f_c(in,out,in1,out1,a)$ **fpar** $f_e(in1,out1))] \subset$
$f_e(in,out))$
$\Rightarrow$
$f_q(in,out,\langle a \rangle)$

But 2. holds by the application of the induction hypothesis.

By considering the full expansion of the second conjunct of 2. it also holds through straightforward application of the induction hypothesis and some simple properties of the combine operator and the **fpar** operator.

Thus the induction is completed and hence the weaker assertion ϕ, in particular, is established.

Finally, from the empty and cell specifications it is an easy task to develop the CSP-like code. As the specifications at this level closely characterise the implementation, its verification is a straightforward application of the proof system. An implementation is given below.

```
proc empty (chan in, out) is
begin
  var x;
  [ in?x ->
     begin
       chan in1,out1;
       ( call cell(in,out,in1,out1,x) || call empty(in1,out1) );
       call empty(in,out)
     end
  [] out!(∅,true) -> skip ]
end;

proc cell (chan in, out, in1, out1; var x);
begin
  var y;
  [ in?y ->
     ( in1!y; call cell(in,out,in1,out1,x) )
  [] out!(x,false) ->
     begin
       var end;
       out1?(y,end);
       [ end -> call cell(in,out,in1,out1,y)
       [] ¬end -> skip ]
     end
  ]
end;
```

7. CONCLUSIONS

The semantics and proof system in this paper combine handling of liveness properties and compositionality. The rather low level of description which seems to be enforced by these objectives as well as the desire to have the proof system closely connected to the semantics leads to straightforward although rather complex formulae. However, as the examples show, the very fact that the proof system does reflect the structure of programs in a direct way and provides independent description of components enables proofs to be carried out in a well structured manner at a natural level of abstraction. Usually the structure of the proofs induces recurring subformulae for which natural ad hoc abbreviations can be supplied. The examples also indicate that the proof strategy can be adapted to the various, e.g. recursive, forms in which specifications can be presented. When used as a hierarchical specification method the complexity of the formulae remains more or less in step with the level of detail arrived at in the development.

It is interesting to note that, in contrast to the usual state-based cpo approaches, in the temporal logic based treatment of recursion given here it is not necessary to use transfinite iteration as used in Park (1979) to handle unbounded non-determinism. In the state-based cpo approach recursion steps are represented by iterations, (intermediate) results being sets of states. Partial computations result in a bottom state and all intermediate information is lost. This is quite different from the real computation. Thus, in the case of unbounded non-determinism where there might be no finite bound on the number of iterations necessary to reach a non bottom state, transfinite iteration is required. In the temporal approach here intermediate information is not abandoned. Each level of iteration fixes the prefix representing that intial part of the computation and leaves the rest of the sequence unconstrained thus mimicking the progress of the actual computation. From these observations it is clear that, as all sequences are of countable length, only countably many iterations are needed.

Further research aims are comparing different fairness assumptions in this framework and devising a more abstract semantics, e.g. only describing the observable behaviour of components.

ACKNOWLEDGEMENTS

The authors express their thanks to Willem P. de Roever and Job Zwiers for helpful and stimulating discussions during the summer when this work was being developed. AP thanks Cliff Jones for hospitality in the department of Computer Science at Manchester University and ICL for providing a visiting fellowship at Manchester. HB and RK acknowledge the SERC for their financial support under grant GR/C/05670.

REFERENCES

Apt, K.R., Francez, N. & de Roever, W.P. (1980).
A Proof System for Communicating Sequential Processes,
ACM TOPLAS, Vol. 2, No. 3, pp. 359–385.

de Bakker, J.W. (1980).
The Mathematical Theory of Program Correctness
Prentice Hall International.

Barringer, H., Kuiper, R. & Pnueli, A. (1984).
Now You May Compose Temporal Logic Specifications,
Proc. of the 16th ACM Symposium on Theory of Computing,
Washington.

Gerth, R. & de Roever, W.P. (1984).
A Proof System for Concurrent Ada Programs,
to appear in Science of Computer Programming.

Hoare, C.A.R. (1978).
Communicating Sequential Processes,
CACM, Vol. 21, No. 8, pp. 666–677.

Hoare, C.A.R. (1981).
A Calculus of Total Correctness for Communicating Processes,
Science of Computer Programming, Vol. 1, No. 1, pp. 49–72.

Lamport, L. (1983).
Specifying Concurrent Program Modules,
ACM TOPLAS, Vol. 5, No. 2, pp. 190–222.

Lamport, L. & Schneider, F. (1984).
The "Hoare Logic" of CSP, and All That,
ACM TOPLAS, Vol. 6, No. 2, pp. 281-296

Levin, G.M. & Gries, D. (1981).
A Proof Technique for Communicating Sequential Processes,
Acta Informatica 15, pp.281-302.

Manna, Z. (1974).
The Mathematical Theory of Computation,
McGraw Hill Book Company.

Manna, Z & Pnueli, A. (1983).
How to Cook A Temporal Proof System for your Pet Language,
Proc. of the ACM Symposium on Principles of Programming,
Austin, Texas, 10, pp. 101-124.

Misra, J. & Chandy, K.M. (1981).
Proofs of Networks of Processes,
IEEE TOSE Vol. SE-7, No. 4, pp 417-426.

Misra, J., Chandy, K.M. & Smith, T. (1982).
Proving Safety and Liveness of Communicating Processes with Examples,
Proc. of the ACM Conf. on the Principles of Distributed Computing,
Ottawa.

de Nicola, R. & Hennessy, M.C.B. (1983).
Testing Equivalences for Processes,
Proc. of the 10th ICALP, Barcelona,
Lecture Notes in Computer Science, Vol. 154, Springer-Verlag, pp. 147-159.

Park, D. (1979).
On the Semantics of Fair Parallelism,
Proc. of the 1979 Copenhagen Winter School on "Abstract Software Specifications"
Lecture Notes in Computer Science, Vol. 86, Springer-Verlag, pp. 504-526.

Pnueli, A. & de Roever, W.P. (1982).
Rendezvous with Ada - A Proof Theoretical View,
Proc. of the AdaTEC Conference, Crystal City.

Sistla, A.P., Clarke, E.M., Francez, N. & Gurevich, Y. (1982).
Can Buffers be Specified in Linear Temporal Logic?
Proc. of the ACM Conf. on the Principles of Distributed Computing,
Ottawa.

Zhou Chao Chen & Hoare, C.A.R. (1981).
Partial Correctness of Communicating Sequential Processes
Proc. of 2nd Int. Conf. on Distributed Computing Systems.

Zwiers, J., de Bruin, A. & de Roever, W.P. (1983).
A Proof System for Partial Correctness of Dynamic Networks of Processes,
Lecture Notes in Computer Science, Vol. 164, Springer-Verlag.

A Really Abstract Concurrent Model and its Temporal Logic

Howard Barringer[(1)]
Ruurd Kuiper[(1)]
Amir Pnueli[(2)]

Extended Abstract
July 1985

(1) University of Manchester, Manchester, England

(2) Weizmann Institute of Science, Rehovot, Israel

Abstract. In this paper we advance the radical notion that a computational model based on the *reals* provides a more abstract description of concurrent and reactive systems, than the conventional *integers* based behavioral model of execution *sequences*. The real model is studied in the setting of temporal logic, and we illustrate its advantages by providing a *fully abstract* temporal semantics for a simple concurrent language, and an example of verification of a concurrent program within the real temporal logic defined here. It is shown that, by imposing the crucial condition of *finite variability*, we achieve a balanced formalism that is insensitive to *finite* stuttering, but can recognize *infinite* stuttering, a distinction which is essential for obtaining a fully abstract semantics of nonterminating processes. Among other advantages, going into real-based semantics obviates the need for the controversial representation of concurrency by interleaving, and most of the associated fairness constraints.

The research was supported in part by SERC grant GR/C/05760.

Part of the research of the third author was supported by ONR grant N00014-85-K-0057 while visiting the University of Texas at Austin.

1. Introduction

Temporal logic is, by now, a widely accepted formal tool for the specification and verification of concurrent and reactive systems (see [MP1], [La1], [OL], [HO], [SMS], [CE], [CM] and many others). The underlying time structure upon which those systems are based is *discrete*, and, in the linear temporal logic case, is isomorphic to the nonnegative integers and models the execution sequences that the specified program generates.

An important step in the construction and justification of temporal proof systems is the definition of *temporal semantics*, which constructs for a given program P a characteristic formula ϕ_P, sometimes denoted by $[\![P]\!]$, such that ϕ_P is true precisely over all the admissible executions of P. Such definitions have been given for global systems in [Pn1], [MP2], and in a more syntax directed style, suitable to compositional proof systems in [BKP1], [BKP2].

When comparing the temporal semantics of concurrent programs with other semantic definitions we find that they are deficient in one respect. Namely, they do not achieve *full abstractness.* Full abstractness ([M1]) is a most important criterion which requires that the semantics level of detail should match the desired level of abstractness. In particular it requires that any two programs that we wish to consider equivalent, should be assigned identical semantics. For sequential programs we can easily say that it was the strive towards full abstractness that led from the overly detailed operational semantics into the much more satisfactory denotational domain-based semantics.

Consider the following two program segments that represent modules in a concurrent program:

$$P_1 :: \ x := 1; \ x := x; \ x := 2 \quad , \text{ and}$$
$$P_2 :: \ x := 1; \ x := x; \ x := x; \ x := 2$$

They differ by the number of dummy $x := x$ assignments separating the two externally observable instructions $x := 1$ and $x := 2$. At the qualitative level that we want to analyze such concurrent programs, these two program segments should be considered equivalent.

Let us examine whether their temporal semantics are indeed identical. We consider first the logic $L^{\oplus} = L(\bigcirc, \mathcal{U})$ presented in [MP1] and other related works. This logic uses the basic operators $\bigcirc$ (next time) and $\mathcal{U}$ (until), over an integer-like execution sequence.

Without giving the precise temporal semantics of P_1 and P_2 we can still explain how they differ. The temporal semantics of P_2 (in the $L^{\oplus}$ logic) requires that in any computation sequence of P_2, the $x = 1$ and $x = 2$ are separated by *at least* 3 computation steps (or two intermediate states). In P_1 the lower bound is only two computation steps. Consequently the $L^{\oplus}$ semantics distinguishes between P_1 and P_2, and hence is not fully abstract.

Lamport perceived this lack of abstractness in the $L^{\oplus}$ logic and attributed the problem to the next-time operator. Consequently, the temporal logic that he works with ([La1], [OL]) is $L^{+} = L(\mathcal{U})$, which uses only the until operator (or an appropriate equivalent). He also formulated the requirement that, in order to be abstract, the logic must be insensitive to *stuttering*, which he defined as finite consecutive duplication of some states. Indeed any execution sequence of P_2 may be obtained from some execution of P_1 by duplication of some states, and the semantics that would be assigned to P_1 and P_2 in the L^{+} logic are identical:

$$[\![P_1]\!] = [\![P_2]\!] = \diamond[(x=1)\mathcal{U}^{+}(x=2)]$$

where we use the defined operator $p\mathcal{U}^{+}q = (p \wedge p\mathcal{U}q)$.

Unfortunately, while this next-less logic provides an abstract semantics for finite processes, i.e., having bounded executions, it raises new problems when we go to infinite processes. We may interpret the view represented by Lamport's approach by saying that there is no *absolute* time scale against which executions are measured. Time advances only when there is a (state) change. Clearly, such a view would naturally ignore any *finite* periods of no change. However, by the same token, it would also ignore (or collapse) *infinite* periods of no change, which is unacceptable.

Consider the following recursive procedure

$$P \Leftarrow [x := x;\; P]$$

where it is assumed that the process P *owns* the variable x, in the sense that P is the only process which may modify x.

The common association of semantics to such a procedure is to form a fixpoint equation for a temporal predicate, where the right hand side of the equation is obtained from the semantics of the procedure's body. We therefore look for the *maximal* solution to the equation:

$$X \equiv \exists u.[(x=u)\mathcal{U}^{+}((x=u) \wedge X)]$$

It is not difficult to see that the maximal solution is $X = T$, i.e., all possible behaviors, in particular those that arbitrarily modify x. This can be explained by the fact that the procedure P produces infinite stuttering which the L^{+} semantics consumes in zero time, and leaves the rest of the execution unrestricted.

If in comparison we consider the semantics assigned to this process by the $L^{\oplus}$ logic, we replace $\mathcal{U}^{+}$ by $\mathcal{U}^{\oplus}$, defined as $r\mathcal{U}^{\oplus}q \equiv r \wedge \bigcirc(r\mathcal{U}q)$. Then the solution is

$$X = \exists u.\; \square(x=u)$$

i.e., x is continuously preserved, which is what we intuitively expect.

Thus we find that $L^{\oplus}$ is unsatisfactory because it is sensitive to finite stuttering, while L^{+} is unsatisfactory because it is insensitive to infinite stuttering.

These difficulties are not specific to temporal semantics. To the best of our knowledge, no adequate compositional semantics of concurrent programs which satisfies all of the four following requirements, has yet been proposed.

1. Allows nondeterminism in the processes.
2. Treats fair parallelism.
3. Is fully abstract, in particular is insensitive to finite stuttering.
4. Properly treats divergent processes, in particular infinitely stuttering processes.

Most of the works that did propose semantics of concurrent programs are usually deficient in points 2 or 4 or both. Point 4 is of course highly subjective, and we have our own interpretation of what the "proper" treatment of nonterminating processes should be. By this interpretation nontermination should not be considered catastrophic, and a silently divergent (infinitely stuttering) process should have no effect on any process running in parallel, except when termination of the full system is considered, which we ignore in this treatment. Thus if we define the silently divergent process:

$$1 = [P \ where \ P \Leftarrow [\ skip \ ; P]]$$

we would like to have

$$(1 \parallel Q) \approx Q$$

for any process Q.

Usually, in works such as [HM], [dBMO], [Br], the silently divergent process 1 is treated either as catastrophic (the Smyth view) or as *chaos* (completely unspecified process) which leads to equivalences of the form

$$(1 \parallel Q) \approx 1.$$

In our previous work ([MP2], [BKP1], [BKP2]) using $L^{\oplus}$ we usually achieved requirements 1, 2, and 4 but had to give up on 3. In this paper we suggest that linear temporal logic with the time structure of the (non negative) *real numbers* provides a more abstract logic than that of the non negative integers, and succeeds in meeting all the four criteria above.

2. Temporal Logic of the Reals (TLR)

Let $V = L \cup G$ be a set of variables which is partitioned into $L = \{y_1, \ldots\}$ the *local* variables, and $G = \{u_1, \ldots\}$ the *global* variables. For simplicity we assume that some of the variables range over a *data* domain, and the others, which we call *propositions*, range over the boolean domain $\{F, T\}$.

A *model* over V is an assignment α that assigns to each variable $v \in V$ and each non-negative real number $t \geq 0$, a value $\alpha(v,t)$ from the appropriate domain. The assignment α is required to satisfy:

a) Uniformity of global interpretation –
For each *global* variable $u \in G$, $\alpha(u,t)$ is independent of t.

b) Finite variability –
for each local variable $y \in L$ there exists a denumerable sequence:

$$0 = t_0 < t_1 < t_2 < \ldots \quad \text{with } t_n \to \infty$$

such that the value of $\alpha(y,t)$ is uniform within each open interval (t_i, t_{i+1}), i.e., for every t, t', if $t_i < t < t' < t_{i+1}$ then $\alpha(y,t) = \alpha(y,t')$.

Condition b) guarantees that there could be no infinite variability within a finite interval, and that the interpretation of each variable can be decomposed into countably many open intervals of constant value. Note that we do not restrict the values at the break-points t_i, which could be different from the values of their left or right neighbor intervals.

The temporal logic we consider is based on the operators $\hat{\mathcal{U}}$ (until) and $\hat{S}$ (since) ([LPZ]).

We define the value of terms and state formulae at a nonnegative real instant t of a model α by evaluating them pointwise, i.e., using $\alpha(v_i,t)$ whenever the value of v_i is needed. For a state formula φ, we denote by $\varphi(\alpha,t)$ the value obtained by such pointwise evaluation at point t. then we define satisfiability as follows:

$(\alpha,t) \models \varphi$	*iff*	$\varphi(\alpha,t) = T$ where φ is a state formula
$(\alpha,t) \models \neg\varphi$	*iff*	$(\alpha,t) \not\models \varphi$
$(\alpha,t) \models (\varphi_1 \vee \varphi_2)$	*iff*	$(\alpha,t) \models \varphi_1$ or $(\alpha,t) \models \varphi_2$
$(\alpha,t) \models \varphi\hat{\mathcal{U}}\psi$	*iff*	$\exists t'', t < t''$, such that $(\alpha,t'') \models \psi$ and for every $t', t < t' < t'', (\alpha,t') \models \varphi$
$(\alpha,t) \models \varphi\hat{S}\psi$	*iff*	$\exists t'', 0 \leq t'' < t$, such that $(\alpha,t'') \models \psi$ and for every $t', t'' < t' < t, (\alpha,t') \models \varphi$

Note that differently from the integer-based TL, the basic *until* operator $\varphi\hat{\mathcal{U}}\psi$ is strict and guarantees a non empty φ interval. We may also define some derived operators:

$$\varphi \wedge \psi = \neg(\varphi \vee \psi) \qquad \varphi \to \psi = (\neg\varphi \vee \psi)$$
$$\diamondplus\, \varphi = T\hat{\mathcal{U}}\varphi \qquad \diamondminus\, \varphi = T\hat{S}\varphi$$
$$\boxplus\, \varphi = \neg\diamondplus\, \neg\varphi \qquad \boxminus\varphi = \neg\diamondminus\, \neg\varphi$$
$$\varphi\mathcal{U}^{+}\psi = \varphi \wedge \varphi\hat{\mathcal{U}}\psi$$

The derived temporal operators $\diamondplus$, $\boxplus$ have similar meaning to that of their integer-based counterparts, except that in real temporal logic they are strict, meaning that the present (point t) is not considered as a part of the future.

Two additional logical operators that are needed are *quantification* and *fixpoint.*

The semantics of the existential quantifier is given by:

$(\alpha, t) \models \exists v.\varphi$ iff there exists a model α' differing from α by at most the assignment given to v, such that $(\alpha', t) \models \varphi$

Note that we allow quantification over both global and local variables, in contrast to [MP1] where quantification is allowed only over global variables. When quantifying over a local variable y, the requirement that α' be a model according to the definition given above implies that v satisfies the finite variability condition.

Universal quantification may be introduced as a derived operator:

$$\forall v.\varphi = \neg\exists v.(\neg\varphi)$$

In order to define the fixpoint operator it helps to slightly shift our view of the semantics of temporal formulae and define for each formula φ and a non-negative real number $t \geq 0$, their *extent* (validity-set) given by:

$$E(\varphi, t) = \{\alpha \mid (\alpha, t) \models \varphi\}$$

This definition associates with each formula φ and time instant $t \geq 0$, a set of all the possible models that satisfy φ at t. This leads to a view by which each formula φ defines a function E_φ from R^+ (the non-negative reals) to $\mathcal{M}$, the set of all models (over V). Let $\mathcal{D} = [R^+ \rightarrow \mathcal{M}]$ denote the set of *all* functions from R^+ to $\mathcal{M}$. It is not difficult to see that it is a complete lattice, actually a complete boolean algebra. The ordering on $\mathcal{D}$ is closely connected to implication between formulas. Thus $\varphi \sqsubseteq \psi$ (when interpreted as elements of $\mathcal{D}$) *iff* $\varphi \rightarrow \psi$ is valid. Consequently the minimal element of $\mathcal{D}$ is $F = \lambda t.\emptyset$ and the maximal element of $\mathcal{D}$ is $T = \lambda t.\mathcal{M}$.

The logical operators may now be viewed as functions from $\mathcal{D}$ to $\mathcal{D}$. Thus for everytwo elements $e_1, e_2 \in \mathcal{D}$, we may express the operators of disjunction and *until* by:

$$e_1 \vee e_2 = e_1 \sqcup e_2 = \lambda t.\{\alpha \mid \alpha \in e_1(t) \quad \text{or } \alpha \in e_2(t)\}$$
$$e_1 \hat{\mathcal{U}} e_2 = \lambda t.\{\alpha \mid \exists t''[t < t'', \alpha \in e_2(t'') \text{ and } \forall t', t < t' < t'', \alpha \in e_1(t')]\}$$

We can show that all the operators defined above excluding $\neg$ are monotonic, while $\neg$ is anti-monotonic over $\mathcal{D}$. Consequently, we consider equations of the form:

$$X \equiv \varphi(X)$$

where X is a local proposition variable, and φ is a temporal formula in which all instances of X are positive, i.e., encompassed by an *even* number of negations. In

such a case this equation is known to have both a *minimal* and a *maximal* solution. We denote them respectively by $\mu X.\varphi$ and $\nu X.\varphi$.

The fixpoint theorem that ensures the existence of the minimal and maximal fixpoints also gives an alternative characterization of these fixpoints as the limit of successive approximations. We define approximations of two kinds; $\vee$-approximations that approximate the minimal fixpoint and $\wedge$-approximations that approximate the maximal fixpoint.

The starting point for both is:

$$\varphi_\vee^0(X) = \varphi_\wedge^0(X) = X.$$

In general, the approximations have to be carried to infinite ordinal orders. For non-limit ordinals, which also covers the finite case we define:

$$\varphi_\vee^{\alpha+1}(X) = \varphi(\varphi_\vee^\alpha(X)) \;, \quad \text{and} \quad \varphi_\wedge^{\alpha+1}(X) = \varphi(\varphi_\wedge^\alpha(X))$$

For a limit ordinal β we define:

$$\varphi_\vee^\beta(X) = \bigvee_{\alpha<\beta} \varphi_\vee^\alpha(X) \quad \text{and} \quad \varphi_\wedge^\beta(X) = \bigwedge_{\alpha<\beta} \varphi_\wedge^\alpha(X)$$

The characterization by limits of approximations states that for each *monotonic* operator φ, there exist two ordinals α and β, such that

$$\mu X.\varphi = \varphi_\vee^\alpha(F)$$
$$\nu X.\varphi = \varphi_\wedge^\beta(T)$$

An interesting special case is when the operator φ is *continuous*. Since we are interested in both minimal and maximal fixpoints, there are two different types of relevant continuity. An operator φ is defined to be $\vee$-*continuous* if it satisfied:

$$\varphi(\bigvee_{i<\omega} p_i) \equiv \bigvee_{i<\omega} \varphi(p_i)$$

for any sequence of temporal predicates p_i, $i = 0, 1, \ldots$. It is defined to be $\wedge$-*continuous* if it satisfies:

$$\varphi(\bigwedge_{i<\omega} p_i) \equiv \bigwedge_{i<\omega} \varphi(p_i).$$

Consider the equation:

$$X \equiv \varphi(X).$$

If φ is $\vee$-continuous then $\mu X.\varphi = \varphi_\vee^\omega(F)$, and if it is $\wedge$-continuous then $\nu X.\varphi = \varphi_\wedge^\omega(T)$. This means that when continuity of the right type is ensured then it is

sufficient to carry the approximations only up to $\alpha = \beta = \omega$, the first infinite ordinal.

It can be established that the operators $\diamondminus$, $\diamondplus$ and $X_1 \hat{\mathcal{U}} X_2$, $X_1 \hat{S} X_2$ with respect to X_2 are $\vee$-continuous, while $\boxminus$, $\boxplus$ and $X_1 \hat{\mathcal{U}} X_2$, $X_1 \hat{S} X_2$ with respect to X_1 are $\wedge$-continuous. The boolean operators $\vee$, $\wedge$ are both $\vee$-continuous and $\wedge$-continuous. Combination of the operators of the two kinds may lead to operators which are monotonic but not continuous.

One result of this characterization is that even if φ is monotonic but not $\vee$-continuous ($\wedge$-continuous), but $X = \varphi^{\omega}_{\vee}(F)$ ($\varphi^{\omega}_{\wedge}(T)$) satisfies the equation $X = \varphi(X)$, then it is a minimal (maximal) fixpoint solution to the equation.

As a simple example consider the equation

$$X \equiv \diamondplus (p \wedge X)$$

Its maximal fixpoint can be obtained by approximations. Denoting $\varphi(X) = \diamondplus (p \wedge X)$, we can show that $\varphi^i(T)$ holds at t *iff* there are i distinct time instants $t < t_1 < \ldots < t_i$ such that p holds at each of the $t_1, \ldots, t_i$. Consequently $\varphi^{\omega}_{\wedge}(T)$ holds at t *iff* there are infinitely many points ahead of t at which p is true. In real temporal logic this leads to:

$$\varphi^{\omega}_{\wedge}(T) = (\boxplus \diamondplus p \vee \diamondplus (p \hat{\mathcal{U}} T))$$

It is not difficult to see that:

$$\varphi^{\omega}_{\wedge}(T) \equiv \diamondplus (p \wedge \varphi^{\omega}_{\wedge}(T))$$

We conclude that:

$$\nu X. \diamondplus (p \wedge X) = (\boxplus \diamondplus p \vee \diamondplus (p \hat{\mathcal{U}} T))$$

On the other hand, the minimal fixpoint of this equation is F. This is due to the fact that F satisfies the equation and is also the minimal element of $\mathcal{D}$. We thus have:

$$\mu X. \diamondplus (p \wedge X) \equiv F$$

An important observation is that all the operators introduced respect the finite variability restriction. This means that the finite variability restriction holds not only for the propositions and variables, but also for any temporal formula defined over them.

3. Axiomatic Characterization of the Real Temporal Logic

Whenever a logic is introduced and recommended as a tool for formal reasoning about programs, an essential part of this recommendation should be a deductive system that supports sound reasoning within the logic itself. Since the full logic, including data variables and predicates is clearly not finitely axiomatizable, we will introduce the deductive system we propose in steps, indicating the step at which we lose completeness and decidability.

The Propositional Fragment

The propositional fragment is obtained by requiring that all the variables in V are propositions, i.e., range over $\{F, T\}$. In this case *global* quantification can be eliminated. This is because for a global proposition u:

$$\exists u.\varphi(u) \equiv \varphi(T) \vee \varphi(F).$$

Consider first the language without (local) quantification or fixpoint operators. We propose the following axiomatization:

F0. All substitution instances of propositional tautologies.

F1. $\boxplus(\varphi \rightarrow \psi) \rightarrow \{[\varphi \hat{\mathcal{U}} \theta \rightarrow \psi \hat{\mathcal{U}} \theta] \wedge [\theta \hat{\mathcal{U}} \varphi \rightarrow \theta \hat{\mathcal{U}} \psi]\}$

F2. $\varphi \wedge \theta \hat{\mathcal{U}} \psi \rightarrow \theta \hat{\mathcal{U}}[\psi \wedge \theta \hat{\mathcal{S}} \varphi]$

F3. $\varphi \hat{\mathcal{U}} \psi \equiv [\varphi \wedge \varphi \hat{\mathcal{U}} \psi] \hat{\mathcal{U}} \psi$

F4. $\varphi \hat{\mathcal{U}} \psi \equiv \varphi \hat{\mathcal{U}} [\varphi \wedge \varphi \hat{\mathcal{U}} \psi]$

F5. $(\varphi \hat{\mathcal{U}} \psi) \wedge \neg(\theta \hat{\mathcal{U}} \psi) \rightarrow (\varphi \wedge \neg\theta) \hat{\mathcal{U}} \psi$

F6. $\varphi \hat{\mathcal{U}} \psi \wedge \theta \hat{\mathcal{U}} \rho \rightarrow [(\varphi \wedge \theta) \hat{\mathcal{U}} (\psi \wedge \rho) \ \vee \ (\varphi \wedge \theta) \hat{\mathcal{U}} (\psi \wedge \theta) \ \vee \ (\varphi \wedge \theta) \hat{\mathcal{U}} (\varphi \wedge \rho)]$

Six additional axioms P1–P6 are obtained as the mirror images of F1–F6, that is, by interchanging in each axiom $\boxplus$ with $\boxminus$ and $\hat{\mathcal{U}}$ with $\hat{\mathcal{S}}$.

Axioms F1 and P1 state that the $\hat{\mathcal{U}}$ and $\hat{\mathcal{S}}$ operators are monotonic in both arguments.

Axioms F2 and P2 specify the relation of reflection holding between past and future. Axioms F3, F4 and their past counterparts characterize the time structure as being *dense*, i.e., between every two instants there exists an additional instant distinct from both. To see this, consider a simpler version $\diamondplus \varphi \rightarrow \diamondplus \diamondplus \varphi$ which also characterizes density. It certainly does not hold in integer-based TL, when we interpret $\diamondplus \varphi$ as $\bigcirc \Diamond \varphi$. Axioms F6 and P6 state that the time structure is *linear*. Essentially it says that if both ψ and ρ are bound to happen, then they will either happen simultaneously or one will precede the other.

F7. $\boxplus \varphi \rightarrow \diamondplus \varphi$

P7. $\boxminus F \vee \diamondminus \boxminus F$

Axiom F7 states that the future is unbounded while P7 asymmetrically states that the past does have a definite starting point.

the proposed system includes the following inference rules:

R1. Modus Ponens: $\vdash \varphi, \vdash (\varphi \rightarrow \psi) \Rightarrow \vdash \psi$.

R2. Generalization: $\vdash \varphi \Rightarrow \vdash \boxplus \varphi, \vdash \boxminus \varphi$.

the system consisting of axioms F0–F7, P1–P7 and rules R1, R2 is taken almost verbatim from [Bu], where it is stated that it forms a sound and complete axiomatic system for the considered fragment of propositional TL over the *rational* half line $Q^+ = \{r \in Q \mid r \geq 0\}$.

In order to characterize the real half line $\mathbb{R}^+$ we usually add a *completeness* axiom. This axiom states that any (Dedekind) cut defined by a change of a proposition, say from T to F, identifies an instant *belonging* to the structure which marks the transition point. In our case, the requirement of finite variability already ensures that any change in value of a variable y must be associated with some node t_i that marks the transition point. Consequently completeness is superceded by the finite variability requirement represented by the axioms:

F8. $\varphi \hat{\mathcal{U}} T \vee (\neg\varphi)\hat{\mathcal{U}} T$

P8. $(\diamondminus T) \rightarrow \varphi \hat{S} T \vee (\neg\varphi)\hat{S} T$

These axioms state that for every formula φ and instant $t \geq 0$, there is always an open interval to the right of t ($\{t' \mid t < t' < t''\}$ for some t'', $t < t''$) in which the value of φ is uniform, and if $t > 0$, also an open interval to the left of t in which φ is uniform.

A consequence of the fact that finite variability implies completeness is that, relative to the language TLR, the class of models based on the reals is equivalent to the class of models based on the rationals. This means that a TLR formula is satisfiable by a real model iff it is satisfiable by a rational model. Consequently we may interpret the R of TLR as standing for either the Reals or the Rationals.

Consider next the introduction of the fixpoint operators to our system. Since the minimal and maximal fixpoint operators are interdefinable, we choose as basic the maximal fixpoint operator. It is controlled by the following axiom:

X1. $\nu X.\varphi(X) \equiv \varphi(\nu X.\varphi(X))$

i.e., the maximal solution to the equation $X \equiv \varphi(X)$ satisfies the equation.

A rule associated with the fixpoint operator is:

R3. $\vdash \theta \rightarrow \varphi(\theta) \Rightarrow \vdash \theta \rightarrow \nu X.\varphi(X)$

This rule states that $\nu X.\varphi(X)$ is the maximal solution to the equation $X \rightarrow \varphi(X)$, and hence every other solution, such as θ above, is smaller than $\nu X.\varphi(X)$.

The minimal fixpoint can be defined by:

$$\mu X.\varphi(X) = \neg\nu X.\neg\varphi(\neg X)$$

The completeness of the system up to this point is discussed in [LP].

The most complex operators in the language are the quantifiers. Actually, the fixpoint operators can be defined by means of quantifiers. Introducing the abbreviation:

$$\Box\varphi = \boxminus\varphi \wedge \varphi \wedge \boxplus\varphi$$

we can express $\nu X.\varphi(X)$ by the following formula:

$$\exists q.[q \wedge \Box\varepsilon(q) \wedge \forall p.\ \Box(\Box\varepsilon(p) \rightarrow \Box(p \rightarrow q))]$$

where $\varepsilon(r)$ is given by $r \equiv \varphi(r)$.

This formula explicitly states that q holds now, q satisfies the equation ε at all points, and any other p satisfying ε at all points is necessarily smaller or equal to q.

The axioms controlling the quantifiers are similar to those presented in [MP3]:

QF1. $\exists p.[\varphi\hat{\mathcal{U}}\psi] \equiv \varphi\hat{\mathcal{U}}(\exists p.\psi)$

where p is not free in φ.

The additional axiom QP1, is the past counterpart of QF1.

Q2. $\forall p.\varphi(p) \rightarrow \varphi(\theta)$

where θ is any formula free for p in φ.

We also have the following rule:

R4. $\vdash \varphi \rightarrow \psi \ \Rightarrow \vdash \varphi \rightarrow \forall p.\psi$

where p is not free in φ.

For the proper definition of the semantics of programs we should be able to establish the existence of propositions that have an infinite variation over the full half line.

For a formula φ, we define the following abbreviations:

$$\text{Rise}(\varphi) = [(\neg\varphi)\hat{\mathcal{S}}(T) \vee (\neg\varphi)] \wedge [\varphi \vee \varphi\hat{\mathcal{U}}T]$$

Rise(φ)is true at $t \geq 0$ *iff* t is a transition point at which φ changes from F to T.

$$\text{Fall}(\varphi) = \text{Rise}(\neg\varphi).$$
$$\text{Ch}(\varphi) = \text{Rise}(\varphi) \vee \text{Fall}(\varphi)$$

Thus Ch(φ) is true at $t \geq 0$ *iff* φ changes its value at t.

$$\text{Clock}(q) = [\ \boxplus \diamondplus\, q \wedge \Box(q \rightarrow (\neg q)\hat{\mathcal{U}}T)]$$

A proposition is called a *clock* if it is true at infinitely many points, and whenever it is true it is immediately false at a right neighboring interval. This implies that q is true at countably many isolated points (never at an interval) and false elsewhere.

We add the following axiom:

C. $\exists q.\{\text{Clock}(q) \wedge \Box(\text{Ch}(\varphi) \rightarrow (\neg\text{Ch}(\varphi))\hat{\mathcal{U}}(q \wedge \neg\text{Ch}(\varphi)))\}$

This axiom states that for any formula φ there exists a clock q that becomes true (ticks) at least once between every two consecutive changes in φ.

The questions of decidability and completeness of this axiomatic system for the propositional fragment of TLR will be discussed in [LP], hoping to establish positive answers for both.

A trivial extension of the propositional fragment, which is still decidable, is obtained by allowing a single fixed data domain D of finite cardinality.

The General Logic

As soon as we allow data domains of unbounded cardinality, the logic becomes highly undecidable and not finitely axiomatizable. In that case we have also to consider quantification over global variables. This quantification obeys axioms QF1, QP1, Q2 and rule R4 as well.

A formula φ is called *global* if it depends only on global variables and propositions. For global formulas φ we have the following axiom:

G. $\varphi \equiv \Box\varphi$

4. A Programming Language and Its Operational Semantics

We introduce a simple programming language of processes which communicate by shared variables. since we want to emphasize their continuous behavior rather than the result they yield on termination, we will not allow them to terminate.

Assuming that the syntax for terms and conditions is understood, the following recursive definition describes the syntax of processes:

Idle: *rest* is a process that performs no further action.

Call: *call P* represent a recursive call to a process P within its body.

Skip: If π is a process then so is *skip*; π

Assignment:

If y is a *data variable*, e a term and π a process, then $y := e; \pi$ is a process that first assigns e to y and then proceed to perform π.

Conditional:

If $\pi_1, \ldots \pi_k$ are processes and $c_1, \ldots, c_k$ are conditions, then $[\overset{k}{\underset{i=1}{\Box}} c_i \rightarrow \pi_i]$ is a process that non-deterministically chooses a direction i such that c_i is true and then proceeds to perform π_i, for some i, $i = 1, \ldots, k$.

Parallel:

If π_1, π_2 are two processes and $\overline{y^1}, \overline{y^2}$ two disjoint sets of data variables, then
$[own\ \overline{y^1}; \pi_1 \| own\ \overline{y^2}; \pi_2]$ is a process that performs π_1 and π_2 in parallel. The *own* declarations partition the available variables into two sets associating with each process the set of variables it is allowed to modify.

Data Variables Declaration:

If π is a process then so is $new\ \overline{y}; \pi$, declaring a set of new variables $\overline{y}$ and then proceeding to perform π.

Process Declaration and Activation:

If P is a process variable and B a process (body) then $[P\ where\ P \Leftarrow B]$ is a process that begins to perform B and recursively reactivate B whenever it meets a call to P. Note that our recursive processes do not admit parameters, and also never return from a call.

A *complete* process will have the general form $own\ \overline{y}; \pi$, where the preceding *own* declaration identifies the variables, not locally declared in π, which π may modify.

Given a complete process we define for each constituent subprocess ρ, the set mod(ρ) which is the set of variables that ρ actually modifies or declares owning.

This is defined by the following equations:

mod $(rest) = \emptyset$
mod $(call\ P)$ = mod(B) when the *call* P is
contained within a $P \Leftarrow B$ declaration.
mod $(skip;\pi)$ = mod(π)
mod $(y := e; \pi)$ = mod$(\pi) \cup \{y\}$
mod $([\overset{k}{\underset{i=1}{\Box}} c_i \rightarrow \pi_i]) = \bigcup_{i=1}^{k} \text{mod}(\pi_i)$
mod $(own\ \overline{y}; \pi)$ = mod$(\pi) \cup \{\overline{y}\}$
mod $(\pi_1 \| \pi_2)$ = mod$(\pi_1) \cup$ mod(π_2)
mod $(new\ \overline{y}; \pi)$ = mod$(\pi) - \{\overline{y}\}$
mod $(P\ where\ P \Leftarrow B)$ = mod(B)

These equations are recursive, so we look for their *minimal* solution.

We may now define for each subprocess ρ the set owns(ρ), which is the set of variables that the context in which ρ occurs has declared as owned by ρ. The

computation of these sets proceeds in a top-down fashion.

If $\rho = skip; \pi$ or $\rho = [y := e; \pi]$, then
owns(π) =owns(ρ)

If $\rho = [\mathop{\Box}_{i=1}^{k} c_i \rightarrow \pi_i]$, then
owns(π_i) =owns(ρ) for each $i = 1, \ldots, k$.

If $\rho = [own\ \overline{y}^1; \pi_1 \parallel own\ \overline{y}^2; \pi_2]$, then we require that owns(ρ) $= \overline{y}^1 \cup \overline{y}^2$ and define

owns(π_i) $= \overline{y}^i$, for $i = 1, 2$

If $\rho = new\ \overline{y}; \pi$, then
owns(π) =owns(ρ) $\cup \{\overline{y}\}$

If $\rho = [P\ where\ P \Leftarrow B]$,
then owns(P) = owns(B) = owns(ρ) $\cup \bigcup_{callP \in B}$ owns($callP$)

This definition is again recursive and we look for the minimal solution.

A complete process $own\ \overline{x}; \rho$ is well formed if:

a) No declaration of the form $new\ \overline{y}$ falls under the scope of another declaration for some variable in $\overline{y}$. Violations of this condition can always be corrected by renaming.

b) Every *call P* process occurs within the body of a declaration for P.

c) For every subprocess π in ρ mod(π) $\subseteq owns(\pi)$.

d) All the free variables in ρ are contained in $\overline{x}$.

We next define operational semantics for this language. We assume that each subprocess within the complete process $own\ \overline{x}; \rho$ is uniquely identifiable. We define a labelled transition relation representing the possible transformations that can be effected in one computation step. Assume a set of states S, each of which is a mapping from the currently declared variables to their values. A *configuration* is a pair $<\pi, \sigma>$ consisting of a process π and a state $\sigma \in S$.

For $\pi = [skip\ ; \rho]$ or $\pi = [own\ \overline{y}\ ; \rho]$, $<\pi, \sigma> \xrightarrow{\pi} <\rho, \sigma>$

For $\pi = [y := e\ ;\ \rho]$, $<\pi, \sigma> \xrightarrow{\pi} <\rho, (\sigma; y{:}\sigma(e))>$

where $(\sigma; y{:}\sigma(e))$ denotes the state obtained from σ by assigning the value of e evaluated at σ to y.

For $\pi = [\mathop{\Box}_{i=1}^{k} c_i \rightarrow \rho_i]$, $<\pi, \sigma> \xrightarrow{\pi} <\rho_i, \sigma>$

for each $i = 1, \ldots, k$ such that $\sigma(c_i) = T$.

For $\pi = [new\ \overline{y}\ ;\ \rho]$, $<\pi, \sigma> \xrightarrow{\pi} <\rho', (\sigma; \overline{y}'{:}\perp)>$

where ρ' and $\overline{y}'$ are obtained from ρ and $\overline{y}$ by systematically renaming all the variables in $\overline{y}$ that are in conflict with the currently declared variables, i.e. the current domain of σ. Again $(\sigma;\overline{y}':\bot)$ denotes the state obtained from σ by augmenting the domain of σ by $\overline{y}'$ and assigning to them the undefined value $\bot$.

For $\pi = [\rho_1 \parallel \rho_2]$, we have

$<\pi,\sigma> \xrightarrow{\lambda} <\rho_1' \parallel \rho_2 \; , \; \sigma'>$

for each transition $<\rho_1,\sigma> \xrightarrow{\lambda} <\rho_1',\sigma'>$, and

$<\pi,\sigma> \xrightarrow{\lambda} <\rho_1 \parallel \rho_2' \; , \; \sigma'>$

for each transition $<\rho_2,\sigma> \xrightarrow{\lambda} <\rho_2',\sigma'>$

For $\pi = [P \; where \; P \Leftarrow B]$, $<\pi,\sigma> \xrightarrow{\pi} <B,\sigma>$

For $\pi =$ *call* P, appearing within the body B of a declaration $P \Leftarrow B$, $<\pi,\sigma> \xrightarrow{\pi} <B,\sigma>$.

If $<\pi,\sigma> \xrightarrow{\lambda} <\pi',\sigma'>$ for some π',σ', then we say that the label (process) λ is *enabled* in the configuration $<\pi,\sigma>$.

An execution sequence corresponding to the initial configuration $<\pi_0,\sigma_0>$ is a labelled transition sequence:

$$S: <\pi_0,\sigma_0> \xrightarrow{\lambda_0} <\pi_1,\sigma_1> \xrightarrow{\lambda_1}, \ldots$$

such that:

a) Every transition appearing in S is justified by the definition above.

b) The sequence S is *maximal*, i.e., it is either infinite or terminates in a configuration $<\pi_k,\sigma_k>$ on which no subprocess of π_k is enabled.

c) The sequence S is weakly fair. This means that we exclude infinite sequences in which for some λ and $i \geq 0$, λ is continuously enabled beyond $<\pi_i,\sigma_i>$ but never taken, i.e., λ is enabled in each $<\pi_j,\sigma_j>$, $j > i$, but for all $j > i$, $\lambda_j \neq \lambda$.

We define $S_{\overline{x}}$, the set of $\overline{x}$*-states*, as the set of all states whose domain is $\overline{x}$. Let $\pi = [own \; \overline{x}; \rho]$ be a complete process, and $s_0 \in S_{\overline{x}}$. A *behavior* of π on s_0 is a finite or infinite sequence of $\overline{x}$-states:

$$B: s_0, s_1, \ldots$$

such that there exists an execution sequence:

$$S: <\pi_0,\sigma_0> \xrightarrow{\lambda_0} <\pi_1,\sigma_1> \xrightarrow{\lambda_1}, \ldots$$

with $\pi_0 = \pi$ and $s_i = \sigma_i |_{\bar{x}}$, i.e., σ_i restricted to the domain $\bar{x}$, for each $i = 0, 1, \ldots$.

This definition of behavior is still too detailed and may contain redundant details such as stuttering. Consequently, we define the notion of a *reduced behavior* which eliminates stuttering altogether. A reduced behavior corresponding to a complete process π and an initial state s_0, is a finite or infinite sequence of x-states which is obtainable from a behavior of π on s_0 by deleting all consecutive duplicates. Obviously such a deletion may transform infinite behaviors into finite reduced behaviors. Let $B(\pi, s_0)$ be the set of all reduced behaviors of π on s_0. Then the operational semantics we assign to a complete process π is a mapping from initial states to reduced behaviors given by:

$$O \llbracket \pi \rrbracket = \lambda s_0 . B(\pi, s_0)$$

This definition leads directly to a definition of an induced observational congruence given by:

The processes π and ρ are *operationally congruent*, $\pi \sim \rho$ iff for every context $C(\cdot)$

(1) $C(\pi)$ is a well formed complete process iff $C(\rho)$ is.

(2) In the case that both $C(\pi)$ and $C(\rho)$ are well formed complete processes, $O \llbracket C(\pi) \rrbracket = O \llbracket C(\rho) \rrbracket$.

As an example of this congruence we observe that

$$(rest) \sim [\,\Box F \rightarrow skip] \sim (P \; where \; P \Leftarrow P)$$

We may now reformulate the challenge we posed in the introduction as: Find a compositional semantics which is fully abstract relative to the operational congruence defined above. We claim that the real temporal semantics that we introduce in the next section answers this challenge.

5. A Real Temporal Semantics

Let $own\ \bar{x}; \pi_0$ be a well formed complete process. Let us associate a temporal proposition variable X_i with each process variable P_i, $i = 1, \ldots k$ defined in π_0. Also assume that we have computed for each subprocess ρ appearing in π_0, its ownership set $owns(\rho)$ determined by its context.

In the section dealing with temporal logic, we have defined the formula $\text{Ch}(\varphi)$ that marks the transition point at which a formula φ changes its truth value. We extend this formula to mark a change in a data variable y by:

$$\text{Ch}(y) = \exists u . \text{Rise}(y = u)$$

This marks the point of a change from $y \neq u$ to $y = u$. We also define the *idling formula* for y:

$$\imath(y) = \neg \mathrm{Ch}(y).$$

The temporal semantics of a process π, denoted by $[\![\pi]\!]$, is a temporal formula that characterizes its behavior in an abstract way. In the following definitions we use the abbreviation $\imath = \imath(owns(\pi))$ to denote that all the variables owned by π are not presently changed. We provide one clause of the definition for each type of subprocess:

- $[\![rest]\!] = \imath \wedge \boxplus \imath$
  This implies that the main effect of the process *rest* is to preserve forever the values of variables it owns.
- $[\![call P_i]\!] = \imath \mathcal{U}^+ X_i$
  where X_i is the proposition variable we have associated with the process variable P_i.
- $[\![skip; \rho]\!] = \imath \mathcal{U}^+ [\![\rho]\!]$
- $[\![y := e; \rho]\!] =$
  $\imath \wedge \exists u[\imath \hat{\mathcal{U}}(\imath \wedge (u = e) \wedge \imath \hat{\mathcal{U}}\{(y = u) \wedge \imath(\mathrm{owns}(\pi) - y) \wedge \imath \hat{\mathcal{U}}\ [\![\rho]\!]\}]$

  This formula identifies a first point in which e is evaluated, and then a second point at which y is assigned the obtained value while all the other variables owned by π are still preserved.
- $[\![\mathop{\Box}_{j=1}^{k} c_j \rightarrow \rho_j]\!] =$

  $$\imath \wedge \{[\boxplus \imath \wedge \bigwedge_{j=1}^{k} \boxplus \diamondplus \neg c_j] \vee \bigvee_{j=1}^{k} \imath \hat{\mathcal{U}}[c_j \wedge [\![\rho_j]\!]]\}$$

  This definition considers the possibility of deadlock at π if each condition is infinitely many times false. the other possibility is the identification of a true c_j followed by the execution of ρ_j.
- $[\![\rho_1 \parallel \rho_2]\!] = [\![\rho_1]\!] \wedge [\![\rho_2]\!]$.

  We consider the simplicity of this clause an important feature that may well justify the complexity of the other clauses.
- $[\![new\ \overline{x}; \rho]\!] = \imath \wedge \left(\bigwedge_{y \in \mathrm{owns}(\pi) \cap \overline{x}} \boxplus \imath(y) \right) \wedge \exists \overline{x}(\imath \hat{\mathcal{U}}\ [\![\rho]\!])$

  The main effect of the declaration of new variables is expressed by the existential quantification over the newly introduced variables. A secondary effect is that all the variables that π owns but have been *covered* or redeclared in $\overline{x}$, i.e., variables in $\mathrm{owns}(\pi) \cap \overline{x}$, will never be modified again. This is because any reference made by ρ to one of these variables is interpreted as addressing the newly declared variable of that name.

- $[\![P_i \; where \; P_i \Leftarrow B_i]\!] = \exists q.\nu X_i[\imath\mathcal{U}^+(\mathrm{Ch}(q) \wedge \imath\mathcal{U}^+ [\![B]\!])]$

  In principle, the natural definition we would expect for process recursion is:

$$\nu X.[\imath\mathcal{U}^+ [\![B]\!]].$$

However, as we explained in the introduction, if B contains an *unguarded* path, i.e., a path with no change in the values of variables, from P to *call* P, the maximal fixpoint of the naive equation will include undesirable behaviors. To ensure that all paths to X_i in $[\![B]\!]$ will contain a change, we impose an external clock q which is required to change at least once on each recursion. By existentially quantifying over it, we abstract away any particular features that may be associated with a specific clock.

Because of space limitations we present the main theorem of this paper without a proof. A detailed proof will be contained in a technical report presenting a fuller version of the paper.

Theorem:

The real temporal semantics presented in this section is fully abstract with respect to the relation of operational congruence.

6. TLR As a Working Tool

The complex formulae appearing in the definition of the temporal semantics of processes may have created the impression that TLR is a complicated formalism to work with. This impression is unjustified, and the apparent complexity should be attributed to the efforts of constructing a compositional semantics of concurrent processes. In fact, for actual reasoning about programs, TLR is quite comparable to integer-based temporal logic, and the added feature of full abstractness makes it an attractive alternative.

Consider for example the following process:

$$\pi\colon \; own \; x; P \; where \; P \Leftarrow [x := x + 1; P]$$

An obvious property of this process is expressed by the formula $(x \geq 0) \rightarrow \boxplus (x \geq 0)$. Let φ denote $x \geq 0$. In the integer-based TL we establish the conditional invariance of φ, i.e., that once it holds it is preserved forever, by showing that all the atomic actions of π preserve φ. Here we do something similar. First, we observe that after some simplifications $[\![\pi]\!] = \Theta$ where

$$\Theta\colon \; \nu X.\exists u[(x = u)\mathcal{U}^+(x = u + 1)\mathcal{U}^+[(x = u + 1) \wedge X]]$$

We have eliminated in this expression the external clock q, since the process itself guarantees a change on each iteration. This elimination can be formally justified. Obviously Θ satisfies its equation:

1. $\Theta \equiv \exists u[(x = u)\mathcal{U}^+(x = u+1)\mathcal{U}^+[(x = u+1) \wedge \Theta]]$
   From which it is not difficult to establish:
2. $\Theta \wedge \varphi \rightarrow \{\varphi\mathcal{U}^+[\mathrm{Ch}(x) \wedge \varphi\mathcal{U}^+(\Theta \wedge \varphi)]\}$
   This can be interpreted as showing that $\Theta \wedge \varphi$ satisfies the equation

   $$Y \rightarrow \varphi\mathcal{U}^+[\mathrm{Ch}(x) \wedge \varphi\mathcal{U}^+ Y]$$

   Consequently, using rule R3 and the existential version of R4 we obtain
3. $\Theta \wedge \varphi \rightarrow \exists x.\nu Y[\varphi\mathcal{U}^+[\mathrm{Ch}(x) \wedge \varphi\mathcal{U}^+ Y]]$
   An important theorem of TLR is:
4. $\{\exists x.\nu Y[\varphi\mathcal{U}^+[\mathrm{Ch}(x) \wedge \varphi\mathcal{U}^+ Y]]\} \equiv (\varphi \wedge \boxplus\, \varphi)$
   We thus obtain
5. $\Theta \wedge \varphi \rightarrow \boxplus\, \varphi$
   Or equivalently
6. $[\![\pi]\!] \rightarrow [\varphi \rightarrow \boxplus\, \varphi]$
   Using the notation of [BKP1] this is representable as
7. $[\pi]\{\varphi \rightarrow \boxplus\, \varphi\}$
   which means that all executions of π satisfy the temporal property $\varphi \rightarrow \boxplus\, \varphi$.

Since the only step in this proof that depends on the specific π and φ considered, was the derivation of 2 from 1, we can condense all the others into a derived proof principle.

Let π be a process of the form:

$$\pi\text{: } \mathit{own}\ \overline{y};\ P\ \mathit{where}\ P \Leftarrow B$$

Denote by $[\![B]\!]\,(X)$ the temporal semantics of B, where dependence on the propositional variable X has been made explicit. Then we have the following rule:

$$\frac{\Theta \equiv [\![B]\!]\,(\Theta) \quad \vdash \ \Theta \wedge \varphi \rightarrow \{\varphi\mathcal{U}^+[\mathrm{Ch}(q) \wedge \varphi\mathcal{U}^+(\Theta \wedge \varphi)]\}}{[\pi]\{\varphi \rightarrow \boxplus\, \varphi\}}$$

A slightly more general rule is needed for the case that B is not guarded.

Inspecting the passage from 1 to 2 above, we see that what is needed is establishing that φ is preserved along any computation path in B leading to any *call* P appearing within B. We also observe that it is very similar to the rule PROC handling recursion in [BKP1].

It is clear that many more derived rules of this kind should be developed before we can use TLR with the same ease and convenience now attained in the integer-based TL. However, we do feel confident that such high-level rules can and will be developed.

7. An Example of Specification and Verification

For a more comprehensive example we consider Peterson's algorithm for mutual exclusion ([Pe]).

In a slightly extended version of our programming language, the algorithm can be presented as:

P: $own\ y_1, y_2, t, in_1, in_2;$

$(y_1, y_2, t, in_1, in_2) := (F, F, F, F, F); [\rho_1 \parallel \rho_2]$

where

ρ_1: $own\ y_1, in_1, \downarrow t\ ;\ [P_1\ where\ P_1 \Leftarrow B_1]$

ρ_2: $own\ y_2, in_2, \uparrow t\ ;\ [P_2\ where\ P_2 \Leftarrow B_2]$

B_1: $[(T \rightarrow\ call\ P_1\ )$
$\square$
$(T \rightarrow y_1 := T;\ t := F\ ;$
$[Q_1\ where\ Q_1 \Leftarrow C_1])]$

B_2: $[(T \rightarrow\ call\ P_2)$
$\square$
$(T \rightarrow y_2 := T;\ t := T\ ;$
$[Q_2\ where\ Q_2 \Leftarrow C_2])]$

C_1: $[(y_2 \wedge \neg t \rightarrow\ call\ Q_1)$
$\square$
$(\neg y_2 \vee t \rightarrow in_1 := T\ ;\ in_1 := F\ ;$
$y_1 := F\ ;\ call\ P_1)]$

C_2: $[(y_1 \wedge \neg t \rightarrow\ call\ Q_2)$
$\square$
$(\neg y_1 \vee \neg t \rightarrow in_2 := T\ ;\ in_2 := F\ ;$
$y_2 := F\ ;\ call\ P_2)]$

The extension we introduced to our programming language is that both ρ_1 and ρ_2 are allowed to modify t, but each in its own way. The notation $\downarrow t$ means that ρ_1 and its subprocesses are only allowed to set t to F, while ρ_2 is only allowed to set t to T. The variable in_1 represents the entry and exit of P_1 in and out of its critical section. Similarly, in_2 represents the criticality of P_2.

As a result the $\imath$ formula for ρ_1 and ρ_2 should read respectively:

$\iota_1 = \neg\mathrm{Ch}(y_1) \wedge \neg\mathrm{Ch}(in_1) \wedge \neg\ Fall\ (t)$

$\iota_2 = \neg\mathrm{Ch}(y_2) \wedge \neg\mathrm{Ch}(in_2) \wedge \neg\ Rise\ (t)$

Writing the semantics of the two processes, it is possible to infer from them the following modular specifications:

$[\rho_1]\{\ \Box(in_1 \rightarrow \Theta_1) \wedge\ \Box(in_1 \wedge \Theta_2 \rightarrow t)\}$

$[\rho_2]\{\ \Box(in_2 \rightarrow \Theta_2) \wedge\ \Box(in_2 \wedge \Theta_2 \rightarrow \neg t)\}$

where Θ_1 and Θ_2 characterize the history of a point in which ρ_1 and ρ_2 are ready to enter their critical section (signified by setting in_1 and in_2 to T).

$\Theta_1 : \iota_1 \wedge \iota_1 \hat{S}(\neg t \wedge y_1)$

$\Theta_2 : \iota_2 \wedge \iota_2 \hat{S}(t \wedge y_2)$

It is easy to see that when we combine these specifications we can obtain (by contradiction):

$[\rho_1 \parallel \rho_2]\ \{\ \Box\neg(in_1 \wedge in_2)\}$

which establishes mutual exclusion.

8. Conclusions

The real-numbers based model and its associated real temporal logic, seem to achieve a higher degree of abstractness than the one provided by the integers-based model. The price does not appear to be excessive since the basic structure of temporal formulae, specifications and proofs is not significantly altered. The gain is obvious since it provides a much cleaner and more natural semantics. This becomes even more apparent when illustrated on a communication based process language such as CCS. It can be shown that the real temporal semantics of CCS attains the same standard of abstractness set up in the algebraic treatment of CCS and its derivatives ([M2], [HM], [dNH]).

Acknowledgements

We would like to gratefully acknowledge the support given by the Weizmann Institute to the visit of the first two authors. Many thanks are due to L. Lamport, M. Chandi and J. Misra for most illuminating discussions, to A. Emerson and L. Zuck for friendly help and advice, to the participants of E.W. Dijkstra's Tuesday afternoon club for many helpful comments, and last but not least to C. Weintraub for her most speedy and efficient typing.

References

[BKP1] Barringer, H., Kuiper, R., Pnueli, A. — Now You May Compose Temporal Logic Specifications, 16th STOC (1984) 51–63.

[BKP2] Barringer, H., Kuiper, R., Pnueli, A. — A Compositional Temporal Approach to a CSP-like Language, Proc. of IFIP Conference: The Role of Abstract Models in Information Processing, Vienna (1985).

[dBMO] de Bakker, J.W., Meyer, J.-J.Ch., Olderog, E.-R. — Infinite Streams and Finite Observations in the Semantics of Uniform Concurrency, 12th ICALP (1985) 149–157.

[Br] Brookes, S.D. — A Semantics and Proof System for Communicating Processes, 2nd Workshop on Logics of Programs, *LNCS* **164** (1983) 68–85.

[Bu] Burgess, J.P. — Basic Tense Logic, in D. Gabbay and F. Guenthner (eds.) Handbook of Philosophical Logic, Vol II, D. Reidel Publishers (1984) 89–133.

[CE] Clarke, E.M., Emerson, E.A. — Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic, 1st Workshop on Logic of Programs, *LNCS* **131** (1981) 52–71.

[CM] Clarke, E.M., Mishra, B. — Automatic Verification of Asynchronous Circuits, 2nd Workshop on Logics of Programs, *LNCS* **164** (1983) 101-115.

[HM] Hennesy, M.C.B., Milner, R. — Algebraic laws for Nondeterminism and Concurrency, *JACM 32*, 1 (1985) 137–161.

[HO] Hailpern, B., Owicki, S. — Modular Verification of Computer Communication Protocols, *IEEE Trans. on Communications*, COM-31, 1 (1983) 56–68.

[HP] Hennesy, M.C.B., Plotkin, G.D. — Full Abstraction for a Simple Parallel Programming Language, Mathematical Foundations of Computer Science, *LNCS*, **74**, Springer Verlag (1979) 108–120.

[J] Jones, C.B. — Software Development: A Rigorous Approach, Prentice Hall International Series in Computer Science.

[La1] Lamport, L. — What Good is Temporal Logic?, Proc. IFIP Congress, Paris (1983) 657–668.

[La2] Lamport, L. — Specifying Concurrent Program Modules, *ACM TOPLAS 5*, 2 (1983) 190–222.

[LP] Lichtenstein, O., Pnueli, A. — A Deductive System for the Temporal Logic of the Reals, Technical Report, Weizmann Institute of Science, in preparation.

[LPZ] Lichtenstein, O., Pnueli, A., Zuck, L. — The Glory of the Past, Logics of Programs, *LNCS*, **193**, Springer Verlag (1985) 196–218.

[M1] Milner, R. — Fully Abstract Models of Typed γ-Calculi, Theoretic Computer Science (1977).

[M2] Milner, R. — A Calculus of Communicating Systems, *LNCS* **92** (1980).

[MP1] Manna, Z., Pnueli, A. — Verification of Concurrent Programs: The Temporal Framework, in Correctness Problem in Computer Science, R.S. Boyer, J.S. Moore (eds.) Academic Press (1982) 215–273.

[MP2] Manna, Z., Pnueli, A. — How to Cook a Temporal Proof System for Your Pet Language, 10th POPL (1983) 141–154.

[MP3] Manna, Z., Pnueli, A. — Verification of Concurrent Programs: A Temporal Proof System, Foundations of Computer Science IV, Distributed Systems, *Mathematical Centre Tracts*, **159**, Amsterdam (1983) 163–255.

[dNH] de Nicola, R., Hennesy, M.C.B. — Testing Equivalence for Processes, 10th ICALP, *LNCS* **154** (1983).

[NGO] Nguyen, V., Gries, D., Owicki, S. — A Model and Temporal Proof System for Networks of Processes, 12th POPL (1985).

[OL] Owicki, S., Lamport, L. — Proving Liveness Properties of Concurrent Programs, *ACM TOPLAS 4*, **3** (1982) 455–495.

[Pe] Peterson G.L. — Myths about the Mutual Exclusion Problem, *Information Processing Letters* 12,3(1981) 115–116.

[Pn1] Pnueli, A. — The Temporal Semantics of Concurrent Programs, *Theoretical Computer Science* **13** (1981) 45–60.

[Pn2] Pnueli, A. — In Transition from Global to Modular Temporal Reasoning About Programs, Proc of NATO School on Logic and Models for Verification and Specification of Concurrent Systems, La Colle-Sur-Loup (1984).

[SMS] Schwartz, R.L., Melliar-Smith, P.M. — Temporal Logic Specifications of Distributed Systems, 2nd International Conference on Distributed Computing Systems, Paris (1981).

Enforcing Nondeterminism via Linear Time Temporal Logic Specifications using Hiding

Ruurd Kuiper

Department of Mathematics and Computing Science,
Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

It is shown how some amount of nondeterminism can be enforced when using linear time temporal logic. This is achieved through extending the notion of satisfaction rather than changing the logic, i.e., no recourse is taken to branching time. The treatment is compared, both in intent and with respect to realisation, to a similar approach using predicate transformers.

1. Introduction

A specification describes requirements which further developments or implementations must fulfill in order to satisfy it. Usually, many decisions are deliberately left open to be filled in at later stages. Consequently, specifications usually contain nondeterminism which will, perhaps only in part, be resolved later.

For example, if production of either of the actions a,b,c or d will satisfy the user, a component S might, using without further explanation an intuitively obvious notation, be specified by

$$S \; sat \; a \vee b \vee c \vee d.$$

The customary interpretation of such a specification is to allow S to be implemented by any process of which the output is in the set $\{a, b, c, d\}$. For instance, by a process $\underline{a}$, which always produces a when activated, but also by $\underline{a \vee c}$, which produces either an a or a c upon different activations.

This kind of nondeterminism, say *allowed* nondeterminism, is not required of the implementation at all and only leaves some freedom to the implementor due to, deliberate, vagueness in the specification.

A completely different kind of nondeterminism, say *required* nondeterminism, is nondeterminism which the implementation should possess.

The author is currently working in and partially supported by ESPRIT project P937: "Debugging and Specification of Ada Real-Time Embedded Systems (DESCARTES)".

For example, a random number generator should not always generate the same number when activated. Yet a specification like

$$S \ sat \ x \in \mathbb{N},$$

interpreted similarly as above as containing allowed nondeterminism, does not guarantee this. Although an implementation of required nondeterminism like $x := x' (x' \in \mathbb{N})$ is intended (cf. [Bo78]), an implementation which always assigns, say, 5 to x would perfectly satisfy this specification.

Often, specification methods make use of the first kind of nondeterminism to allow general specifications, but cannot handle the second kind. Branching time temporal logic (cf.[EH86]), which describes behaviour as sets of trees, is one of the few exceptions. Linear time temporal logic, describing behaviour as sets of sequences, does, in its usual form, not have this expressive ability. There are, however, many different considerations which at present leave the debate as to which of the two is the most suitable, wide open.

Another framework in which both kinds of nondeterminism can be distinguished is that of partial orders (cf.[Pr86]). However, the level of abstraction obtained there appears to be lower than in the temporal logic case and more appropriate in a context where refinement is part of the development process.

We will present and discuss a way to enable in the context of linear time temporal logic, specification of a modest amount of required nondeterminism. The idea is to limit the extent to which the allowed nondeterminism may be resolved by additionally specifying a lower bound. This enforces implementationsto possess a degree of nondeterminism between the bounds set by the required and the allowed nondeterminism.

For the above examples such lower bounds might be, respectively, $a \vee c$ and $x \in \{1,...,100\}$

The present paper is an extension of [Ku87] in that a solution is proposed to a problem concerning development that was not solved is a satisfactory way in the previous paper. That problem and its solution, by means of hiding, are described in a now extended section 4.
In section 2 we briefly discuss the (only) approach similar to ours we know of, namely [Fr77]. This is carried out in the context of predicate transformers and safety properties, but it will be seen that a more general idea underlies this approach. In section 3 we show how this can be used for linear time temporal logic specifications. The interaction with development is discussed in the next section. In section 5 a brief look is taken at the situation for branching time temporal logic. The last section contains some discussion.

2. A precursor: required nondeterminism and predicate transformers

In [Fr77], Francez addresses specifying allowed and required nondeterminism using predicate transformers. We look at the example given above, $S \ sat \ a \vee b \vee c \vee d$, with the extra aim to specify some required nondeterminism.

Let the specification of S be given as $\{\phi\}S\ \{\psi\}$
In the usual weakest precondition approach, only considering allowed nondeterminism, this means that S has to satisfy

(i) $\phi \Rightarrow wp(S,\psi)$ where, in this example,

$$\phi = \text{true}$$

$$\psi = a \vee b \vee c \vee d.$$

This only gives an upper bound to the allowed nondeterministic behaviour of S and allows implementations like, e.g., $S = \underline{b}$.

The idea in [Fr77] now is, to enforce ψ as a lower bound on required nondeterminism as well, again using weakest preconditions. The extra part of the satisfaction notion then is, that S should also satisfy

(ii) $\forall\psi^* \neq \psi[(\psi^* \Rightarrow \psi) \Rightarrow \neg(\phi \Rightarrow wp(S,\psi^*))]$,

where in this example again

$$\phi = \text{true}$$

$$\psi = a \vee b \vee c \vee d.$$

It can be easily seen, that together these requirements limit the implementations to $\underline{a \vee b \vee c \vee d}$ only.

In this example, lower and upper bound coincide. The words lower and upper suggest, although [Fr77] does not claim this, noncoinciding bounds, allowing a range of implementations in between them. This might, for instance, be denoted by

$\{\phi\}S\ \{\underline{\psi},\overline{\psi}\}r$, where $\underline{\psi}$ is the lower

and $\overline{\psi}$ the upper bound.

Intuitively, expressed in terms of an obvious semantics of i/o pairs, the lower /upper bound approach, in our view, aims at achieving the following kind of constraints.
Let $<i,a>$ denote: on any input, produce a. Take as lower and upper bound requirements respectively

$$\underline{\psi} = a \vee c$$

and

$$\overline{\psi} = a \vee b \vee c \vee d.$$

Then, denoting the i/o semantics of S by $[\![S]\!]$, the desired constraint on S would be

$$\{<i,a>,<i,c>\} \subset [\![S]\!] \subseteq \{<i,a>,<i,b>,<i,c>,<i,d>\},$$

i.e., allowing the implementations $\underline{a \vee c}$, $\underline{a \vee b \vee c}$, $\underline{a \vee c \vee d}$ and $\underline{a \vee b \vee c \vee d}$.
Unfortunately, using (ii) with ψ as $\underline{\psi}$ does not give the desired result. Namely (ii) now is of the form

$$\forall\underline{\psi}^* \neq a \vee c[(\underline{\psi}^* \Rightarrow a \vee c) \Rightarrow \neg(\text{true} \Rightarrow wp(S,\underline{\psi}^*))].$$

Consider the implementation $S = \underline{b}$. As S produces only b, S does not satisfy $wp(S,a \vee c)$, which

will remain the case if $a \vee c$ is strengthened. So S is, contrary to the intuition, allowed as an implementation of $\{\phi\}S\ \{\underline{\psi}\}$. Hence, the approach in [Fr77] seems limited to coinciding lower and upper bounds.

In the next section, the lower/upper bound approach will be adapted to linear time temporal logic specifications and extended to enable the use of lower and upper bounds that do not coincide.

3. Enforcing required nondeterminism using linear time temporal logic

In linear time temporal logic (LTL) we take both the specification, ψ, and the semantics, $[\![S]\!]$, of an implementation S to be an LTL formula. Such a formula in turn can be interpreted as characterizing a set of (state) sequences, namely those for which it is true.

The customary satisfaction relation when considering only allowed nondeterminism is then straightforward:

$$S\ sat\ \psi \triangleq [\![S]\!] \Rightarrow \psi.$$

Intuitively this means that the set of sequences that can be generated by S is included in the set allowed by ϕ. It is clear that any less nondeterministic implementation S', meaning that the set of sequences it can generate is smaller, which in turn means that $[\![S']\!] \Rightarrow [\![S]\!]$, satisfies ψ as well. So the implication makes it impossible to specify required nondeterminism. Establishing a lower bound is a solution and, in the LTL framework, can be easily incorporated in a manner reflecting the intuitive set inclusion as mentioned in the previous section.

Namely, by redefining the notion of satisfaction as follows:

$$S\ sat\ <\underline{\psi}, \overline{\psi}> \triangleq \underline{\psi} \Rightarrow [\![S]\!] \Rightarrow \overline{\psi}.$$

The specification of, for instance, the first example, in the formal notation as used in [BKP84], i.e. assuming sequences to have labels indicating environment (E) steps and component (Π) steps, then becomes:

$$S\ sat\ <\underline{\psi}, \overline{\psi}>,$$

where

$\underline{\psi} = E\ U\,(\Pi \wedge (a \vee c))\ \mathrm{C}\,fin,$

(Which informally states:
starting with environment steps E,
eventually a component step occurs which produces a or c,
after which the component stops.)

and

$\overline{\psi} = E\ U\,(\Pi \wedge (a \vee b \vee c \vee d))\ \mathrm{C}\,fin.$

(Similar meaning, but describing production of a, b, c *or* d.)

Remarks

(i) An alternative way to enable specifying required nondeterminism may seem to change the implication to equivalence :

$$P \ sat \ \psi \triangleq [\![P]\!] = \psi.$$

This indeed fulfills the aim, but does not possess the lower and upper bound flexibility. Consequently, extra allowed nondeterminism can now only be obtained by explicitly listing the allowed alternatives, for example, via exclusive or notation:

$$S \ sat \ \psi_1 \oplus \psi_2 \oplus \cdots \oplus \psi_n \triangleq$$

$$S \ sat \ \psi_1 \oplus S \ sat \ \psi_2 \oplus \cdots \oplus S \ sat \ \psi_n.$$

This is undesirable, as usually when giving a specification one only has a rough idea about what one wants to allow, but certainly not a full grasp of all possible alternatives. Furthermore, if infinitely many alternatives for implementation exist, as in the case of the random number generator example, it is not possible to list all of these unless infinite $\oplus$ is allowed.

(ii) In, e.g., [Pn85] a strong notion of expressivity is defined for specification methods: A method is expressive $\triangleq$ for all S there is a characteristic specification, $spec_c$ such that:

(i) For all S', $S' \ sat \ spec_c \Leftrightarrow ([\![S]\!] = [\![S']\!])$

(ii) For all $spec$, $S \ sat \ spec \Leftrightarrow (spec_c \Rightarrow spec)$

This property usually does not hold; it is obtained for the system described in [BKP84] when it is extended as above.

4. Development

One part of development is concerned with decomposition into subspecifications. The extension of the notion of specification is such, that adapting of this part of existing methods is straightforward.

For instance, a compositional specification method dealing with required nondeterminism can be obtained by using an existing one like described in [BKP84] and just redefining the notion of specification as above and adapting the proof rules as follows.

For the decomposition part, the essential rules are those concerned with syntactical combinators, e.g., sequential and parallel composition, enabling to derive properties of components from properties of their syntactic subcomponents. These rules reflect the semantics of such operators and are of the form

$$\frac{\begin{array}{c} S_1 \ sat \ \psi_1 \\ S_2 \ sat \ \psi_2 \\ C'(\psi, \psi_1, \psi_2) \end{array}}{C(S_1, S_2) \ sat \ \psi}$$

where C is a syntactical combinator on the components and C' the corresponding syntactical

condition on the specifications.

The translation then is

$$\frac{S_1 \; sat \; <\underline{\psi}_1, \overline{\psi}_1> \qquad S_2 \; sat \; <\underline{\psi}_2, \overline{\psi}_2> \qquad C'(\underline{\psi}, \overline{\psi}, \underline{\psi}_1, \overline{\psi}_1, \underline{\psi}_2, \overline{\psi}_2)}{C(S_1, S_2) \; sat \; <\underline{\psi}, \overline{\psi}>}$$

A concrete example, for sequential composition, uses the temporal logic operator **C** (chop). The idea is, that $\phi \, \mathbf{C} \, \psi$ is true for a sequence if and only if this sequence can be chopped into two consecutive sequences for which ϕ respectively ψ is true.

$$\frac{S_1 \; sat \; <\underline{\psi}_1, \overline{\psi}_1> \qquad S_2 \; sat \; <\underline{\psi}_2, \overline{\psi}_2> \qquad (\underline{\psi} \Rightarrow \underline{\psi}_1 \, \mathbf{C} \, \underline{\psi}_2) \wedge (\overline{\psi}_1 \, \mathbf{C} \, \overline{\psi}_2 \Rightarrow \overline{\psi})}{S_1 ; S_2 \; sat \; <\underline{\psi}, \overline{\psi}>}$$

Another part of development is concerned with extending the requirements on the behaviour. In the context of LTL this intuitively means further narrowing down the sets of sequences allowed by the specification. In the $\underline{\psi} \Rightarrow [\![S]\!] \Rightarrow \overline{\psi}$ framework, this amounts to weakening (!) $\underline{\psi}$ and strengthening $\overline{\psi}$. This gives rise to the following rule.

$$\frac{S \; sat \; <\underline{\phi}, \overline{\phi}> \qquad \underline{\psi} \Rightarrow \underline{\phi} \qquad \overline{\phi} \Rightarrow \overline{\psi}}{S \; sat \; <\underline{\psi}, \overline{\psi}>}$$

Again turning to the previously used example, this means that it can be derived that from

$$S \; sat \; <a \vee c \vee d, \; a \vee c \vee d>$$

it follows that

$$S \; sat \; <a \vee c, \; a \vee b \vee c \vee d>$$

This corresponds to the intuition, as the first specification only allows the implementation $S = a \vee c \vee d$. This is, as has been seen previously, one of the various implementations allowed by the second specification.

Remark

There is a rather subtle problem in the treatment of required nondeterminism in development. Assume specifications to be given in terms of interface variables. Of variables about which at a certain stage in the development nothing has yet been decided, usually nothing is required, i.e., all sequences are allowed as regards their values.

However, if nothing is required in ψ about such a variable, this should remain so during further development, because, as seen from the rules, ψ may only be weakened. Intuitively, as seen from the example, if straightforward strengthening of already mentioned variables is involved, there is no problem, because required nondeterminism for this variable was explicitly stated.

A possible solution in this case is to argue that a development step causes a lower level of abstraction to be used. This may be refelected by the addition of new variables to the interface.
Requirements, especially required nondeterminism, pertaining to such variables can then also be seen as limited to this level only.

The problem then disappears, as ψ on a higher level of specification cannot impose requirements on these variables. This approach may be formalized by introducing an explicit interface for each level of specification. (See, e.g., [BK83].)

The above solution can be incorporated more directly in the system via the use of hiding. To focus thought, reconsider the example used previously.

$$S\ sat(x_1 \vee x_2 \vee x_3, x_1 \vee x_2 \vee s_3 \vee x_4),$$

denoting the values some variable x might obtain.
What about variables that were not mentioned, say $y \neq x$? Assume a lower bound $y_1 \vee y_2$ and an upper bound $y_1 \vee y_2 \vee y_3$ is required for $y \neq x$.

$$(x_1 \vee x_2) \cdots ? \cdots \Rightarrow [\![S]\!] \Rightarrow (x_1 \vee x_2 \vee x_3) \wedge (y_1 \vee y_2 \vee y_3)$$

should be the form of the requirement.
However, as $\psi = x_1 \vee x_2$ did not mention y, *all* values of y were implicitly required as possible.

We now introduce hiding.
Let $\phi(x,y)$ restrict x and y.
Then $\phi(x,y)/y \stackrel{def}{=} \exists y \cdot \phi(x,y)$, ϕ with y hidden, restricts only x.

Using hiding, satisfaction can be redefined so as to represent the intuitively desirable feature that variables which have not been mentioned are unrestricted. Let $var(\phi)$ be the set of free variables in ϕ.

$$S\ sat^* <\underline{\psi}, \overline{\psi}> \triangleq (\underline{\psi} \Rightarrow [\![S]\!] / var([\![S]\!] \setminus var(\underline{\psi})))$$

$$\wedge$$

$$([\![S]\!] \Rightarrow \overline{\psi})$$

This produces the desired result for the example:

$$S\ sat^* <x_1 \vee x_2, x_1 \vee x_2 \vee x_3 \vee x_4>$$

allows an implementation $S = (x_1 \vee x_2) \wedge y_1$, as $x_1 \vee x_2 \Rightarrow \exists y \cdot (x_1 \vee x_2) \wedge y_1$.

In a further development step, nondeterminism for y can be enforced through

$$S\ sat^{*} <(x_1 \vee x_2) \wedge (y_1 \vee y_2), x_1 \vee x_2 \vee x_3 \vee x_4>$$

which is, e.g., not satisfied by $\underline{S = (x_1 \vee x_2) \wedge y_1}$, as $(x_1 \vee x_2) \wedge (y_1 \vee y_2) \not\Rightarrow (x_1 \vee x_2) \wedge y_1$.

The new definition of satisfaction requires a change to be made to the development rule concerning extending requirements.
As can be observed from the last example, the clause $\underline{\psi} \Rightarrow \underline{\phi}$ in the old rule is not satisfied:

$$x_1 \vee x_2 \not\Rightarrow (x_1 \vee x_2) \wedge (y_1 \vee y_2)$$

It should be clear from the discussion so far that the required change is:

$$\frac{\begin{array}{c} S\ sat^{*} <\underline{\phi}, \overline{\phi}> \\ \underline{\psi} \Rightarrow \underline{\phi}/var(\overline{\phi}) \setminus var(\underline{\psi}) \\ \overline{\phi} \Rightarrow \overline{\psi} \end{array}}{S\ sat^{*}, <\underline{\psi}, \overline{\psi}>}$$

If in the above example the development is further restricting the required nondeterminism of x to $x_1 \vee x_2 \vee x_3$ and the allowed nondeterminism of x to $x_1 \vee x_2 \vee x_3$ then this development can be proved to be correct using the rule, as:

(i) $x_1 \vee x_2 \Rightarrow ((x_1 \vee x_2 \vee x_3) \wedge (y_1 \vee y_2))/y$
because
$x_1 \vee x_2 \Rightarrow x_1 \vee x_2 \vee x_3$
and

(ii) $(x_1 \vee x_2 \vee x_3) \wedge (y_1 \vee y_2 \vee y_3) \Rightarrow x_1 \vee x_2 \vee x_3 \vee x_4$

Correct implementations similarly can be seen to be
$\underline{(x_1 \vee x_2 \vee x_3) \wedge (y_1 \vee y_2)}$ and $\underline{(x_1 \vee x_2 \vee x_3) \wedge (y_1 \vee y_2 \vee y_3)}$.

Expressiveness is maintained.
S is specified up to observational equivalence by

$$S\ sat <[\![S]\!]/inv([\![S]\!]), [\![S]\!]/inv([\![S]\!])>$$

where $inv([\![S]\!])$ denote the set of invisible variables of S.
Let $O[\![S]\!]$ be the restriction of the description of $[\![S]\!]$ to visible variables, then clearly

$$O[\![S']\!] = O[\![S]\!] \text{ iff}$$

$$[\![S]\!]/inv([\![S]\!]) \Rightarrow [\![S']\!]/var([\![S']\!]) \setminus var([\![S]\!]/inv([\![S]\!]))$$

$$\wedge$$

$$[\![S']\!] \Rightarrow [\![S]\!] \setminus inv([\![S]\!]).$$

5. Branching time temporal logic

In branching time temporal logic (BTL), formulae are interpreted not as characterizing sets of sequences, but sets of trees.
It is possible to state that a branch possessing certain properties is present in such trees. Hence, required nondeterminism can be expressed directly.

One consideration when choosing between LTL and BTL is, that in the LTL application we proposed, behaviour of a process is still viewed as a set of state sequences. Generally, BTL allows to express branching at every arbitrary state in the tree. This could be regarded as providing too much structure. Of course, one may put some restriction on the use of BTL formulae (more specifically, on the use of existential quantification over branches) to yield trees that only branch at the root. This could be seen as the transition point from linear to branching time views. Also see in this respect [GS86].
Another consideration might be, that decision procedures for branching time logics are in general more complex than those for comparable linear time ones.

As yet, there seems to be no concensus about which of the views is the most suitable (or when).

For more information on BTL see, e.g., [EL85, EH86].

6. Discussion

We presented a way to enforce some amount of required nondeterminism via LTL specifications. It is sometimes argued that specifying required nondeterminism is meaningless, as no test will be able to falsify a claim like, e.g., $\psi = a \vee b$. The idea is, that even after repeated testing with consistently result a, b might still occur at some future test.
One remark here is, that exactly the same argumentation applies to fairness requirements like: eventually b will occur. This concept however now seems quite well accepted.

More direct counter arguments are the following:

(i) When designing a system, it is natural that initially some properties are underdefined. During development these may be strengthened to falsifiable ones, which is certainly the only way in which they can be implemented.

(ii) An implementation will come together with a proof that its specification is met, so testing is not required.

A fortunate consequence of the fact that the extension made to the notion of specification retains the interpretation as a pure LTL formula and does not alter the logic is, that existing decision procedures (see, e.g., [Go83]) can still be used. There seems to be a problem with the use of hiding to enable development in that it interferes with compositionality. We indicate where (though not how!) the rule for sequential composition should be adapted. We use the rule in a strong form, writing $\overline{\psi}_1$ C $\overline{\psi}_2$ for $\overline{\psi}$) and unsuccessfully attempting the same for $\underline{\psi}$.

$$\frac{\begin{array}{c} S_1\ sat^* <\underline{\psi}_1, \overline{\psi}_1>, \text{i.e.} [\![\underline{\psi}_1]\!] \Rightarrow [\![S_1]\!]/var([\![S_1]\!]) \setminus var([\![S_2]\!]) \\ \wedge [\![S_1]\!] \Rightarrow \overline{\psi}_1 \\ S_2\ sat^* <\underline{\psi}_2, \overline{\psi}_2>, \text{i.e.} [\![\underline{\psi}_2]\!] \Rightarrow [\![S_2]\!]/var([\![S_2]\!]) \setminus var([\![S_2]\!]) \\ \wedge [\![S_2]\!] \Rightarrow \overline{\psi}_2 \end{array}}{S_1;S_2\ sat^* <?, \overline{\psi}_1\ C\ \overline{\psi}_2>}$$

An open problem is, whether existing devices that contain nondeterminism like the one given above, like random number generators, will in general satisfy intuitively correct abstract specifications of this property. Furthermore, if this is the case, how can this be proven? The link between the formulation of the practical and the theoretical properties seems not obvious.

Acknowledgements
Many thanks go to Ron Koymans and Rob Gerth for comments and help at various stages and especially to Willem-Paul de Roever, who provided the link to reference [Fr77].
I am very grateful to Edmé van Thiel for Elastic Time Typing.

References

[BK83] Barringer, H., Kuiper, R., *Towards the Hierarchical, Temporal Logic, Specification of Concurrent Systems,* LNCS 207.

[BKP84] Barringer, H., Kuiper, R., Pnueli, A., *Now You May Compose Temporal Logic Specifications,* ACM-STOC 1984.

[Bo78] Boom, H.J. : *A weaker precondition for loops,* Math. Centrum Report, Amsterdam, 1978.

[EL85] Emerson, E.A., Chin-Laung Lei, *Modalities for Model Checking: Branching Time Logic Strikes Back,* POPL 1985.

[EH86] Emerson, E.A., Halpern, Y.N., "Sometimes" and "Not Never" Revisited: *On Branching versus Linear Time Temporal Logic,* JACM, Vol. 33, No. 1, 1986.

[Fr77] Francez, N., *A Case for a Forward Predicate Transformer,* Inf. Proc. Letters IEEE 6:6, 1977.

[GS86] Graf, S., Sifakis, J., *A Logic for the Description of Non-deterministic Programs and their Properties,* Inf. and Control, Vol. 68, Nos. 1-3, pp. 254-270 (1986).

[Go83] Gough, G.D., M.Sc. Thesis, *Decision Procedures for Temporal Logic,* Univ. of Manchester, 1983.

[Ku87] Kuiper, R., *Enforcing Nondeterminism via Linear Time Temporal Logic Specifications,* Proc. of the SION Conference on Computing Science in the Netherlands, 1987.

[Pn85] Pnueli, A., *Linear and Branching Structures in the Semantics and Logics of Reactive Systems,* LNCS 194.

[Pr86] Pratt, V., *Modelling Concurrency with Partial Orders,* International Journal of Parallel Programming 15, 1986.

SAMENVATTING

Nog steeds vormt het vervaardigen van programma's die aan de verwachtingen van hun gebruikers voldoen een probleem. Dit is in het bijzonder het geval bij concurrente programma's waarbij diverse berekeningen tegelijk worden uitgevoerd. Toch zijn zulke programma's gewenst, bijvoorbeeld uit efficiëntie oogpunt of vanwege de inherent concurrente aard van sommige talen.

Het probleem kan gesplitst worden in twee deelproblemen.

1. Het beschrijven van de eisen waaraan een programma moet voldoen: specificatie.
2. Het komen tot een programma dat aan deze eisen voldoet: ontwikkeling.

Een onderdeel van de oplossing van het eerste deelprobleem is het kiezen van een geschikt niveau van detail voor de beschrijving. Uiteraard zal een specificatie minder precies zijn dan het uiteindelijke programma, dit betreft derhalve abstractie.

Volgens een dikwijls toegepaste methode geven wij de waarden van in een berekening voorkomende variabelen weer via het begrip toestand: een functie die aan variabelen waarden toekent. Op verschillende momenten geven bijbehorende toestanden dan de waarden van de variabelen weer.

Een geschikt abstractie niveau voor het beschrijven van een grote klasse concurrente berekeningen is dat, waarop de onderlinge volgorde van individuele toestanden kan worden beschreven. De abstractie is, dat geen preciese tijdstippen aan toestanden worden verbonden.

Een in het afgelopen decennium bestudeerd formalisme voor het uitdrukken van dergelijke specificaties is temporele logica. Speciale symbolen voor het weergeven van de orderelatie tussen toestanden maken het mogelijk het expliciet noemen van tijdsmomenten te vermijden.

Voor de in dit proefschrift beschouwde vragen blijkt de eenvoudige variant lineaire tijd temporele logica toereikend. Hierin wordt tijd beschouwd als een lineair geordende stroom momenten in plaats van, bijvoorbeeld, een boomstructuur van alternatieven of een slechts partiëel geordende verzameling gebeurtenissen.

Een onderdeel van de oplossing voor het tweede deelprobleem is het verder ontwikkelen van een specificatie voor een programma (ook wel component genoemd) door middel van opsplitsing in deelspecificaties van de deelcomponenten van dat programma. Dit noemt men ontwikkeling via decompositie.

Component gedrag, de betekenis van component specificaties en het samenvoegen van deelcomponenten worden alle gemodelleerd via temporele logica. Een bewijssysteem maakt het dan mogelijk te bewijzen of deelcomponenten die aan zekere deelspecificaties voldoen na samenvoeging de specificatie vervullen van een eerder gespecificeerde component. Dit noemt men verificatie.

Een bewijssysteem moet voldoen aan de volgende voorwaarden.

(i) Geldigheid: alleen wat waar is in het temporele logica model kan bewezen worden.

(ii) Volledigheid: alles wat waar is in het model kan ook bewezen worden.

In dit proefschrift worden modelleringen en bewijssystemen voor verschillende oplossingen voor de eerder genoemde twee deelproblemen gegeven.

Het proefschrift bestaat uit vijf hoofdstukken.

Het eerste hoofdstuk (dat overigens het laatst geschreven is) is een poging wat orde te scheppen in de vele, vaak slechts gebrekkig geformuleerde, begrippen op het gebied van specificatie.

In het bijzonder compositionaliteit en modulariteit worden nader beschouwd, en wel wat betreft hun betekenis in modellen. Het eerste begrip zegt iets over de mogelijkheden voor gegeven deelcomponenten deelspecificaties te vinden die te combineren zijn, het tweede iets over de mogelijkheden gegeven deelspecificaties te combineren zonder over de deelcomponenten te beschikken.

In de volgende twee hoofdstukken wordt aandacht besteed aan een aanpak van het tweede probleem: compositionele ontwikkeling.

Het blijkt van belang in de specificatie van een component informatie op te nemen over de omgeving (andere componenten!) waarin deze functioneert. Deze aanpak bouwt voort op ideeën van Lamport.

Een natuurlijk niveau van abstractie wordt bereikt door voor elke component de omgeving ervan te beschrijven als één geheel, dus zonder de deelcomponenten waaruit deze bestaat apart te onderscheiden. Dit bouwt voort op een idee van Jones, de "component/environment aanpak", en een door Aczel hiervoor ontwikkeld model.

In het tweede hoofdstuk wordt een taal beschouwd waarin communicatie tussen deelcomponenten plaatsvindt door middel van gemeenschappelijke variabelen. Dit wil zeggen dat verschillende componenten dezelfde variabelen kunnen veranderen of lezen.

De belangrijkste bijdrage in dit hoofdstuk is de formalisering van de component/environment aanpak binnen lineaire tijd temporele logica.

In het derde hoofdstuk wordt een soortgelijke aanpak geïntroduceerd voor een op Hoare's taal, Communicating Sequential Processes, gebaseerde taal. De interactie tussen deelcomponenten is hier gecompliceerder, namelijk via synchrone communicatie waarbij deelcomponenten op elkaar moeten wachten en bovendien uit verscheidene communicatie mogelijkheden kunnen kiezen.

Een bekende modellering is, in essentie, dat aan de toestanden een keuzeverzameling

van mogelijke communicaties wordt toegevoegd. Dit volgt ideeën van Hennessy/De Nicola en Brookes/Hoare/Roscoe.

Gedemonstreerd wordt hoe een dergelijke aanpak kan worden ingepast in het model, dat in het vorige hoofdstuk is gepresenteerd.

Het vierde hoofdstuk betreft het eerste deelprobleem: abstractie.

Toepassingen van temporele logica in de informatica gaan meestal uit van een discreet tijdsdomein, waarbij tijdsmomenten bijvoorbeeld worden weergegeven door de natuurlijke getallen.

Hierdoor blijken sommige intuïtief gelijke programma's ongewild verschillende betekenis te krijgen. In essentie is het probleem als volgt.

Een programma dat een willekeurige tijd, maar ongelijk nul, actief is, wordt in discrete tijd gemodelleerd als

"een of meer tijdsmomenten actief".

De sequentiële compositie van twee van zulke programma's zou intuïtief dezelfde betekenis moeten krijgen, maar leidt, op voor de hand liggende wijze, tot modellering als

"twee of meer tijdsmomenten actief".

De in dit hoofdstuk gepresenteerde oplossing is het gebruik van een dicht tijdsdomein, bijvoorbeeld de reële getallen.

De betekenis van het boven gegeven programma alswel van de sequentiële compositie van twee ervan wordt nu

"gedurende een willekeurig eindig lang niet-leeg interval actief".

De prijs die hiervoor wordt betaald is overigens, dat een extra conditie moet worden opgenomen die uitsluit dat oneindig veel gebeurtenissen plaatsvinden in eindig veel tijd. Voor zover nu bekend is een dergelijke conditie onvermijdelijk bij het gebruik van dichte tijdsdomeinen.

Tenslotte wordt in het laatste hoofdstuk niet-determinisme nader beschouwd.

In het algemeen worden in specificaties diverse mogelijkheden toegelaten als component gedrag—de specificaties abstraheren immers. Een specificatie kan dus worden beschouwd als bovengrens aan de mogelijkheden van component gedrag.

Voor sommige toepassingen is ook een ondergrens gewenst. Bijvoorbeeld bij een programma dat willekeurige getallen moet genereren zou zeker moeten worden geëist dat

het zich niet beperkt tot slechts één mogelijkheid.

In dit laatste hoofdstuk laten wij zien hoe een idee van Francez voor het toevoegen van een ondergrens aan specificaties, ontwikkeld vóór temporele beschrijvingen bekend werden, in de context van lineaire tijd temporele logica specificaties kan worden gebruikt.

Deze modificatie toont tevens de grenzen aan van lineaire tijd temporele logica voor de directe uitdrukking van keuzes (i.t.t. de toevoeging van keuzeverzamelingen als in het derde hoofdstuk).

Voor meer uitdrukkingskracht in dit opzicht is men aangewezen op, bijvoorbeeld, vertakkende tijd temporele logica. Hierbij wordt tijd niet meer als lineaire stroom momenten maar als een zich vertakkende stroom beschouwd.

CURRICULUM VITAE

Op 26 juni 1953 werd ik te Amsterdam geboren.

Na het eindexamen Gymnasium β aan het Hervormd Lyceum West te Amsterdam begon ik in 1971 de studie aan het Wiskundig Seminarium van de Vrije Universiteit te Amsterdam.

Na het kandidaatsexamen Wiskunde met bijvakken Natuur- en Sterrenkunde in 1974 (cum laude) trok vooral topologie mijn aandacht.

Het doctoraal examen in 1979 met uitgebreid hoofdvak wiskunde en bijvak psychologie (cum laude) omvatte dan ook een afstudeerscriptie in deze richting, over de Absoluut, bij Prof. P.C. Baayen. Hij suggereerde mij Informatica als onderzoeksterrein.

Vervolgens trad ik als assistent (aanvankelijk voor halve werktijd) in dienst bij de Afdeling Informatica van het Mathematisch Centrum, bij Prof. J.W. de Bakker. Hij wekte mijn belangstelling voor de semantiek van programmeertalen, in het bijzonder wat betreft niet-determinisme. Dit leidde tot een publicatie in die richting [R. Kuiper, An Operational Semantics for Bounded Nondeterminism Equivalent to a Denotational One, Algorithmic Languages (J.W. de Bakker, J.C. van Vliet eds.), North-Holland, pp. 373-398, 1981].

Tijdens deze periode kwan ik, op zoek naar een promotieonderwerp, in contact met Willem-Paul de Roever, die onderzoek naar de toepassing van temporele logica in de informatica suggereerde als mogelijkheid. De hieruit volgende samenwerking leidde in 1982 tot een artikel over fairness principes voor Hoare's taal Communicating Sequential Processes [R. Kuiper, W.P. de Roever, Fairness Assumptions for CSP in a Temporal Logic Framework, Formal Description of Programming Concepts-II (D. Bjørner ed.), North-Holland, pp. 159-170, 1983].

De conferentielezing hierover leidde ertoe dat Howard Barringer mij uitnodigde als research associate te komen werken aan de Universiteit van Manchester— eerst bij de groep van Prof. C.B. Jones, later bij een project van hemzelf.

Samenwerking met Barringer leidde in 1983 tot een publicatie en tot een lezing op het STL/SERC symposium te Cambridge [H. Barringer, R. Kuiper, Towards the Hierarchical Temporal Logic Specification of Concurrent Systems, LNCS 207, pp. 157-184, 1984] waaruit het contact met Prof. A. Pnueli voortkwam.

Uit de hierop volgende samenwerking ontstonden de artikelen waarop dit proefschrift voornamelijk is gebaseerd.

Sinds 1986 ben ik als universitair docent in dienst bij de Technische Universiteit Eindhoven, bij de Sectie Theoretische Informatica, geleid door Willem-Paul de Roever. Hier kwam het laatste gedeelte van dit proefschrift tot stand en werd tevens een beter inzicht gekregen in de onderliggende problematiek van de erin behandelde onderwerpen.

Current address:

Department of Computing Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven,
The Netherlands
Email:. mcvax!eutrc3!wsinruur.uucp.

Promotiereglement artikel 15.3b

The EUT "promotieregelement" requires that if a thesis contains co-authored papers it should be indicated which parts are based on active contributions of the author of the thesis.

In the case of chapter 1, co-authored by Koymans and Zijlstra, the material presented there forms part of a more extensive attempt by the three authors to investigate which requirements a real time specification method should satisfy; this research is carried out in the context of the ESPRIT project DESCARTES. As regards the part presented in this thesis, the author of the thesis has been actively involved in all parts of the research and has contributed at least three quarters of the material. The investigations were initiated during his stay at the University of Manchester and carried through and completed at the EUT.

In the case of chapters 2, 3 and 4, with co-authors Barringer and Pnueli, after an initial phase during which Barringer and Kuiper developed a precursor, [BK83], of chapter 2, all three authors have been actively and essentially involved at all stages and in all parts of the research reported on in chapter 2, 3 and 4. It is not feasible to provide any further quantitative or qualitative division of the effort; both Barringer and Pnueli are directly involved in the evaluation of this thesis as, repectively, one of the promotores and member of the "kleine commissie".

Copyright

The copyright of the papers reproduced in this thesis (with kind permission of the copyright holders) remains with the journals or proceedings in which they were published originally - as claimed in these journals or proceedings, listed in the table of contents of this thesis.

STELLINGEN

bij het proefschrift Combining Linear Time Temporal Logic Descriptions of Concurrent Computations, door Ruurd Kuiper

1. Expliciete identificatie van de actieve component voor elke transitie in een concurrente berekening is niet noodzakelijk voor het verkrijgen van een compositioneel temporele logica formalisme.

2. Indien in de componentspecificaties voor elke transitie in een concurrente berekening expliciet wordt aangegeven of dit een component- of omgevingstransitie betreft, kunnen de compositieoperatoren in een compositioneel temporele logica formalisme eenvoudiger zijn dan wanneer expliciete identificatie niet plaatsvindt.
[Hoofdstuk 2 van dit proefschrift.]

3. De opmerking van Lamport en Zwiers dat in het artikel dat het tweede hoofdstuk van dit proefschrift vormt voor elke operator uit de programmeertaal een nieuwe temporele operator wordt geïntroduceerd is misleidend in de zin dat deze nieuwe operatoren slechts worden gegeven als alternatief voor in de standaard logica plus dekpunten gedefinieerde operatoren.
[Hoofdstuk 2 van dit proefschrift,
Lamport, L., An Axiomatic Semantics of Concurrent Programming Languages, NATO ASI Series, Vol. F13, Logics and Models of Concurrent Systems, (K.R. Apt ed.), Springer, pp. 77–122, 1985,
Zwiers, J., Compositionality, Concurrency and Partial Correctness: Proof theories for networks of processes, and their connection, Ph.D. Thesis, Eindhoven University of Technology, 1988; also as LNCS 321, 1989].

4. Compositionaliteit en modulariteit voor de semantiek van specificatieformalismen geven het verband aan tussen compositionaliteit voor de semantiek van programmeertalen enerzijds en compositionaliteit en modulariteit van bewijssystemen voor specificatieformalismen anderzijds.
[Hoofdstuk 1 van dit proefschrift.]

5. Het criterium van Stirling voor de vergelijking van invarianten is onjuist als dit wordt toegepast onder de geïntroduceerde conventie dat $\neg\emptyset = ff$. Het is juist als $\neg\emptyset = tt$ wordt gebruikt.
[Stirling, C., A generalization of Owicki-Gries's Hoare logic for a concurrent while language, Theoretical Computer Science 58, p. 351, 1988.]

6. Het gebruik van verzamelingen bomen als semantiek voor concurrente processen met externe en interne keuzes maakt het mogelijk zowel concurrente executie als deze beide soorten keuzes via modelstructuur afzonderlijk weer te geven. Op natuurlijke wijze correspondeert concurrente executie met "interleaving", externe keuze met vertakking en interne keuze met verschillende bomen.

7. Bij gebruik van verzamelingen partiëel geordende verzamelingen als semantiek voor concurrente processen wordt partiële ordening aangewend om concurrente executie te modelleren. Interne keuze correspondeert op natuurlijke wijze met verschillende partiëel geordende verzamelingen. Er is dan geen modelstructuur meer over om externe keuze te modelleren. Kennelijk moet men hiervoor weer zijn toevlucht nemen tot modellering door middel van "failure sets" of "ready sets".
[Hoare, C.A.R., Communicating Sequential Processes, Prentice-Hall, 1985, Hennessy, M.S., Algebraic Theory of Processes, MIT Press, 1988.]

8. Het verdient aanbeveling het gebruik van de termen "liveness" en "fairness" gescheiden te houden en te beperken tot, respectievelijk, de categorie van progressie eigenschappen en die van keuze eigenschappen.

9. Een voldoende expressief domein voor de semantiek van natuurlijke talen is dat der functies van hersentoestanden naar hersentoestanden.
Kunstmatige intelligentie onderzoek toont aan dat het abstraheren hiervan veel moeilijker is dan het abstraheren van machinetoestanden in het geval van programmeertalen.

10. De "Kunst der Fuge" van J.S. Bach is een vroeg voorbeeld van machine-onafhankelijke specificatie van concurrente processen.