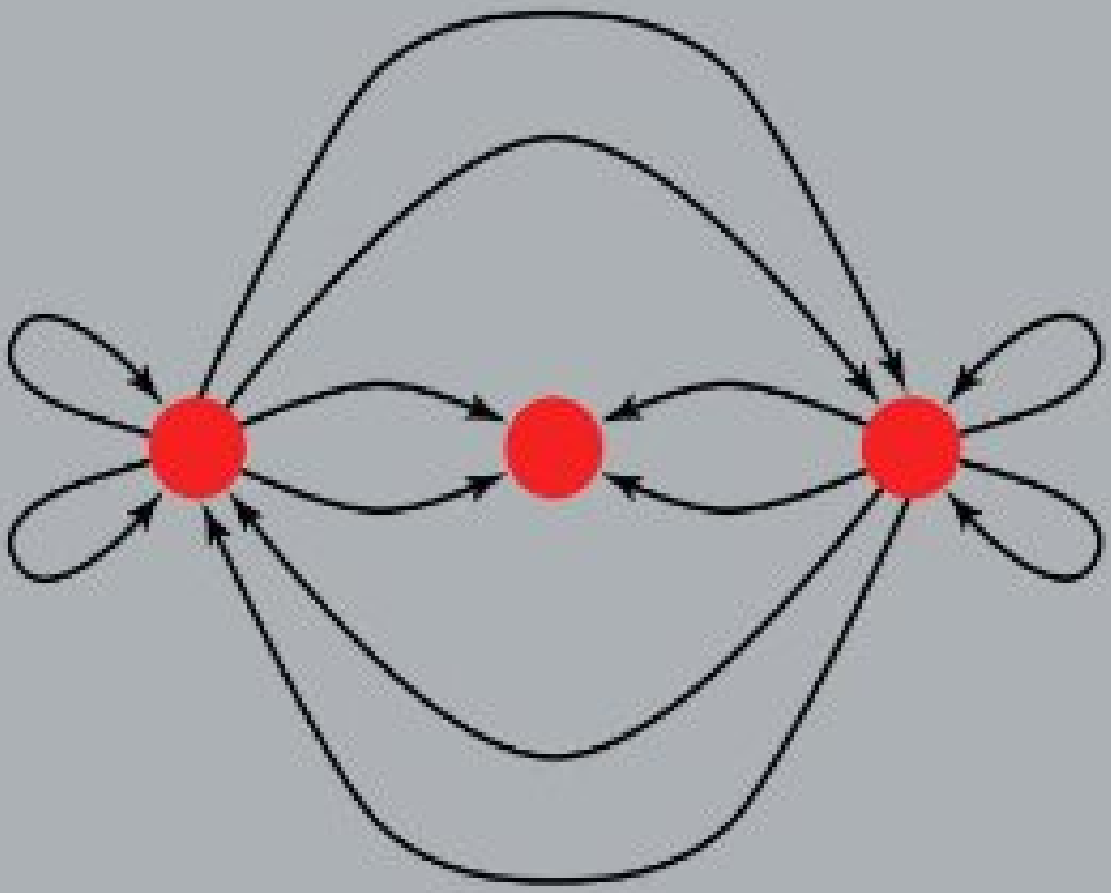
MONOGRAPHS IN COMPUTER SCIENCE

PROGRAMMING METHODOLOGY

Annabelle McIver
Carroll Morgan
Editors



IFIP International Federation for Information Processing

Monographs in Computer Science

Editors

David Gries
Fred B. Schneider

Springer Science+Business Media, LLC

Monographs in Computer Science

Abadi and Cardelli, **A Theory of Objects**

Benosman and Kang [editors], **Panoramic Vision: Sensors, Theory, and Applications**

Broy and Stølen, **Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement**

Brzozowski and Seger, **Asynchronous Circuits**

Cantone, Omodeo, and Policriti, **Set Theory for Computing: From Decision Procedures to Declarative Programming with Sets**

Castillo, Gutiérrez, and Hadi, **Expert Systems and Probabilistic Network Models**

Downey and Fellows, **Parameterized Complexity**

Feijen and van Gasteren, **On a Method of Multiprogramming**

Leiss, **Language Equations**

McIver and Morgan [editors], **Programming Methodology**

Misra, **A Discipline of Multiprogramming: Programming Theory for Distributed Applications**

Nielson [editor], **ML with Concurrency**

Paton [editor], **Active Rules in Database Systems**

Selig, **Geometrical Methods in Robotics**

Annabelle McIver
Carroll Morgan
Editors

Programming Methodology

With 68 Figures

Springer

Annabelle McIver
Department of Computing
Macquarie University
Sydney 2109, Australia
anabel@ics.mq.edu.au

Carroll Morgan
Department of Computer Science
and Engineering
The University of New South Wales
Sydney 2052, Australia
carrollm@cse.unsw.edu.au

Library of Congress Cataloging-in-Publication Data
McIver, Annabelle 1964–
Programming methodology/Annabelle McIver, Carroll Morgan.
p. cm.—(Monographs in computer science)
Includes bibliographical references and index.

1. Computer programming. I. Morgan, Carroll, 1952–. II. Title. III. Series.
QA76.6 .M3235 2002
005.1—dc21 2002017377

ISBN 978-1-4419-2964-8 ISBN 978-0-387-21798-7 (eBook)
DOI 10.1007/978-0-387-21798-7

Originally published by Springer-Verlag New York. Inc in 2003
Softcover reprint of the hardcover 1st edition 2003

www.springer-ny.com

Preface

The second half of the twentieth century saw an astonishing increase in computing power; today computers are unbelievably faster than they used to be, they have more memory, they can communicate routinely with remote machines all over the world — and they can fit on a desktop. But, despite this remarkable progress, the voracity of modern applications and user expectations still pushes technology right to the limit. As hardware engineers build ever-more-powerful machines, so too must software become more sophisticated to keep up.

Medium- to large-scale programming projects need teams of people to pull everything together in an acceptable timescale. The question of how programmers understand their own tasks, and how they fit together with those of their colleagues to achieve the overall goal, is a major concern. Without that understanding it would be practically impossible to realise the commercial potential of our present-day computing hardware.

That programming has been able to keep pace with the formidable advances in hardware is due to the similarly formidable advances in the principles for design, construction and organisation of programs. The efficacy of these methods and principles speaks for itself — computer technology is all-pervasive — but even more telling is that they are beginning to feed back and influence hardware design as well. The study of such methods is called *programming methodology*, whose topics range over system- and domain-modelling, concurrency, object orientation, program specification and validation.

That is the theme of this collection.

Programming Methodology

Most systems today aim to be secure, robust, easy-to-use and timely. To achieve these aims the programmer needs the right tools, which in this context are "intellectually-based", and comprise techniques to help organise complex problems and express them in a way that can be both understood by developers and interpreted by machines.

The desire to reduce complexity (or at least to hide it where possible) has been the driving force behind the invention of design methods and principles, many of which are now built in to popular programming languages and (automatic) program-development tools. Typed languages for instance help with error de-

tection, and the object-oriented programming method and data abstraction (both present for example in Java) support program modification, programming at the interface-level and readability. Meanwhile concurrency has flourished with the introduction of concurrent languages together with formal tools, including the model-checkers and proof assistants which are used in validation.

Many of these tools have at their heart impressive theoretical credentials — "assertions" and "program invariants" rely on a theory of programming logics; and specification and refinement techniques have program semantics at their basis. The essays in this collection concentrate on new and emerging techniques for constructing modern applications; they deal with the problems that software designers face and propose practical solutions together with their theoretical foundations.

The idea of assembling papers on this theme to form a book arose in the technical meetings of the members of the Working Group 2.3 of the International Federation for Information Processing (IFIP).

Working Group 2.3

The working groups of IFIP meet regularly to discuss new ideas — their own, and others' — and to evaluate and promote trends in many aspects of computing systems. Their official output varies widely between individual groups, and depends largely on the traditions and style of the current membership, though they frequently promote special courses and host conferences.

The term "programming methodology" was coined by one of the members of WG2.3, and over the group's nearly thirty years of existence, its members have contributed to many of the topics mentioned above; and indeed many flourishing areas of research in programming methodology today are based on ideas which were once discussed and developed in WG2.3 meetings.

This Collection

The present volume represents the second official publication by the group. Our aim was to gather material which would attract both students and professionals working either in an academic or industrial environment. Indeed we hope that this collection will form a reference and guide to the front line of research activity in programming methodology.

The range of subjects reflects the interests of the current membership and addresses in particular the problems associated with contemporary demands for highly complex applications that actually work. Many of the essays contain new material, highlighting specific theoretical advances, whilst others aim to review or evaluate a particular area, or to outline suggestive problems for further investigation.

Structure

The book comprises three parts, each one devoted to a major theme in programming methodology. The parts are further divided into subsections where essays focussing on a particular topic lying within the scope of its overall section are gathered together. The short introductions at the beginning of each subsection serve to set the scene for the detailed articles to follow.

Systems may be complex because they are distributed over a network, or because they are time-critical or concurrent — the first part deals with the business of describing, modelling and analysing such systems. The second part concentrates on specific programming techniques, the "programmer's toolkit", whilst the final part elaborates on some topical applications including security and telephony.

Acknowledgments

It goes without saying that this book would have been impossible to put together without the creative work of the authors of the articles. We thank especially Natarajan Shankar (chairman of WG2.3) for the initial motivation for this project and David Gries for help in its realisation.

Annabelle McIver
Carroll Morgan
Sydney, Australia, 2002

IFIP WG2.3 dedicates this book to the fond memory of two of its founding members:

Ole-Johan Dahl (1931–2002)
and
Edsger Wybe Dijkstra (1930–2002)

Contents

Contributors

Members of the WG2.3 working group:

Professor Dr. R.-J. Back
Dr. R.M. Balzer
Dines Bjørner
Professor Dr. M. Broy
Dr. Ernie Cohen
Dr. P. Cousot
Professor Dr. E.W. Dijkstra
Professor D. Gries
Professor I.J. Hayes
Professor E.C.R. Hehner
Professor P. Henderson
Professor C.A.R. Hoare
Dr. J.J. Horning
Professor Daniel N. Jackson
Mr. M.A. Jackson
Professor C.B. Jones
Dr. B.W. Lampson
Dr. K. Rustan M. Leino
Dr. M.D. McIlroy
Dr. Annabelle McIver
Dr. W.M. McKeeman
Dr. K. McMillan
Professor J. Misra
Dr. Carroll Morgan (vice-chairman)
Dr. Greg Nelson
Professor Benjamin Pierce
Dr. J.R. Rao
Professor J.C. Reynolds
Dr. D.T. Ross
Professor F.B. Schneider
Dr. N. Shankar (chairman)
Professor M. Sintzoff
Dr. Joakim von Wright
Dr. Pamela Zave

Members who contributed are:

Dines Bjørner
Department of Information
Technology
Technical University of Denmark
DTU-Building 344
DK-2800 Lingby
Denmark

Ralph Back
Abo Akademi University
Department of Computer Science
Lemminkainenkatu 14
SF-20520 Turku
Finland

Manfred Broy
Institut für Informatik
Technische Universität München
D-80290 München
Germany

Ernie Cohen
ernie.cohen@home.com

Ian Hayes
School of Information Technology and Electrical Engineering
The University of Queensland
Queensland, 4072
Australia

Eric C.R. Hehner
Pratt Building, Room PT398
University of Toronto
6 King's College Road
Toronto, Ontario M5S 3H5
Canada

Peter Henderson
Department of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ
United Kingdom

Tony Hoare
Microsoft Research
Cambridge, CB3 0FB
United Kingdom

Daniel Jackson
Lab. for Computer Science
200 Technology Square
Cambridge, Massachusetts 02139
USA

Michael Jackson
jacksonma@acm.org

Benjamin Pierce
University of Pennsylvania
Department of Computer and Information Science
200 South 33rd Street
Philadelphia, Pennsylvania
19104-6389
USA

Rustan Leino
Compaq SRC
130 Lytton Avenue
Palo Alto, California 94301
USA

Annabelle McIver
The Department of Computing
Macquarie University
Sydney, 2109
Australia

Jayadev Misra
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188
USA

Carroll Morgan
Department of Computer Science and Engineering
The University of New South Wales
Sydney, 2052
Australia

Greg Nelson
Compaq SRC
130 Lytton Avenue
Palo Alto, California 94301
USA

C.B. Jones
Department of Computing Science
University of Newcastle
Newcastle-upon-Tyne, NE1 7RU
United Kingdom

John C. Reynolds
Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15123-3890
USA

Josyula Rao
IBM Research
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598
USA

Natarajan Shankar
SRI International
MS EL256
333 Ravenswood Avenue
Menlo Park, California 94025-3493
USA

Joakim von Wright
Abo Akademi University
Department of Computer Science
Lemminkainenkatu 14
SF-20520 Turku
Finland

Pamela Zave
180 Park Avenue
P.O. Box 971
Florham Park, New Jersey 07932-2971
USA

The following people also contributed:

Suresh Chari
IBM Research
T.J. Watson Research Center
P.O. Box 714
Yorktown Heights, New York 10598
USA

Charanjit S. Jutla
IBM Research
T.J. Watson Research Center
P.O. Box 714
Yorktown Heights, New York 10598
USA

Pankaj Rohatji
IBM Research
T. J. Watson Research Center
P.O. Box 714
Yorktown Heights, New York 10598
USA

He Jifeng
International Institute of
Software Technology
United Nations University
P.O. Box 3058
Macau

Theodore Norvell
Electrical and Computer Engineering
Faculty of Engineering
Memorial University of Newfoundland
St. John's, NF A1B 3X5
Canada

Richard F. Paige
Department of Computer Science
York University
4700 Keele Street
Toronto, Ontario M5J 1P3
Canada

Part I

Models and correctness

Section A

Concurrency and interaction

The practical application of a formal method to the correct design of industrial-strength programs is impeded if the method does not scale. But scalability (equivalently efficiency for large problems) can be tricky. In particular a problem must be reduced to smaller subproblems, analysed in some way, and the results recomposed to produce a solution to the original problem. The litmus test for scalability in such a procedure is that the analysis of the subproblems must be both cost-effective and composable — have one without the other and the enterprise flounders. Unfortunately having both is not as easy as it might seem, because a cost-cutting analysis implies 'measuring' only the absolutely essential details, and has a tendency to encroach on composability. Methods that do compose are called compositional.

A simple chemistry example illustates the point. An analysis of chemicals might be based on colour which, for the sake of argument, is a more obvious candidate for observation than weight. In reactions however the colour of the product cannot be deduced from the colours of the reagents, so a weight-based analysis is compositional but a colour-based analysis is not.

This paper explores this crucial idea of compositionality, focussing on its application to concurrent programs, which present special challenges to the designer.

Interactive systems are a generalisation of the traditional notion of concurrency in that they allow different kinds of scheduling — typically demonic and angelic — during computation. The important mathematical idea underlying both concurrency and interaction is that of multi-user games, where subsets of users can form coalitions to comply with some particular contract. The analysis of temporal properties for these systems is relatively tricky, and the aim of research on this

topic is to simplify analysis, either by discovering straightforward proof rules or by simplifying the systems themselves.

This paper can be seen as contributing to both those areas. Using an operational description of the kinds of contracts and interactions involved in game playing, this work demonstrates how to develop simple verification rules in the well-known action-system framework.

Action systems enjoy impressive credentials as a formal method because of their descriptive clarity and expressivity for concurrent programs. Indeed they are a natural choice for this application, for their predicate-transformer semantics extends easily to cope with both angelic and demonic scheduling. Moreover other typical features of contract-games, such as various kinds of contract breaking, are modelled by termination, abortion and miracles in action systems. Other specialised treatments of games are unable to deal with these concepts.

1

Wanted: a compositional approach to concurrency

C. B. Jones

Abstract

A key property for a development method is *compositionality*, because it ensures that a method can scale up to cope with large applications. Unfortunately, the inherent *interference* makes it difficult to devise development methods for concurrent programs (or systems). There are a number of proposals such as rely/guarantee conditions but the overall search for a satisfactory *compositional approach to concurrency* is an open problem. This paper identifies some issues including granularity and the problems associated with ghost variables; it also discusses using atomicity as a design abstraction.

1.1 Compositionality

Formal specification languages and associated rules for proving that designs satisfy specifications are often called *formal methods*. As well as providing completely formal criteria, it is argued in [Jon00] that formal methods offer thinking tools –such as invariants– which become an integral part of professional practice. The main interest in this paper is on the contribution that formal methods can make to the design process for concurrent systems. Just as with Hoare's axioms for sequential programs, the sought after gains should come both from (the reference point of) formal rules and from the intuitions they offer to less formal developments.

The development of any large system must be decomposed into manageable steps. This is true both for the construction phase and for subsequent attempts to comprehend a design. For software, understanding after construction is important because of the inevitable maintenance and modification work. But, for the current purposes, it is sufficient to concentrate the argument on the design process.

A. McIver et al. (eds.), *Programming Methodology*

It is easy to see that it is the design process of large systems which requires support. Regardless of the extent to which techniques for error detection of finished code can be made automatic, there is still the inherent cost of reworking the design when errors are detected and little greater certainty of correctness after modification. The only way to achieve high productivity and correctness is to aim to make designs correct by construction.

What is required therefore is to be able to make and justify one design decision before moving on to further steps of design. To take the design of sequential programs as a reference point, specification by pre- and post-conditions offers a natural way of recording what is required of any level of component. So, in facing the task of developing some C specified by its pre- and post-conditions, one might decide that a series of sub-components sc_i are required and record expectations about them by writing their pre- and post-conditions. The design step must also provide a proposed way of combining the eventual sc_i and this should be one of the constructs of the (sequential) programming language. Each such construct should have an associated proof rule like the Hoare axiom for **while** which can be used to show that any implementations satisfying the specifications of the sc_i will combine with the stated construct into an implementation satisfying the specification of C. This idealised top-down picture requires some qualification below but the essential point remains: pre- and post-conditions provide an adequate description of the functionality of a system to facilitate the separation of a multi-level design into separate steps. A method which supports such development is classed as *compositional*; one that requires details of the implementations of the sc_i to justify the decomposition is non-compositional.

The above ideal is rarely achieved. The first difficulty is that there is no guarantee against making bad design decisions which result in the need to backtrack in the design process. What a compositional method offers is a way of justifying a design step — not an automatic way of choosing good design decisions. Secondly, there was above a careful restriction to functional properties and performance considerations, in particular, are commonly excluded. There are also a number of technical points: the case for separating pre- from post-conditions and the arguments for employing post-conditions of two states (plus the consequent search for apposite proof rules) are explored in [Jon99]. It will also come as no surprise to anyone who has read this author's books on VDM that the method of data reification is considered an essential tool for program design; fortunately there is also a transitivity notion for reification which again facilitates compositional design (see [dRE99] for an excellent survey of data refinement research).

Nothing which has been written above should be seen as assuming that all design has to be undertaken in a top-down order: separate justification of design steps is necessary in whatever order they are made; a top-down structure of the final documentation might well enhance comprehensibility; and arguments based on specifications rather than on the details of the code have much to commend them however these arguments are discovered.

The key argument of this section is that compositionality is a desirable property of a development method if it is to scale up to large tasks. Subsequent sections explore the difficulties in achieving this property in the presence of concurrency.

1.2 The essence of concurrency is interference

The easiest way to illustrate interference is with parallel processes which can read and write variables in the same state space. Simple examples can be constructed with parallel execution of assignment statements; but to avoid an obvious riposte it is necessary to resolve an issue about granularity. Some development methods assume that assignment statements are executed atomically in the sense that no parallel process can interfere with the state from the beginning of evaluation of the right hand side of the assignment until the variable on the left hand side has been updated. The rule is reciprocal in the sense that the assignment in question must not interfere with the atomic execution of one in any other process. Essentially, assignments in all processes are non-deterministically merged in all processes but never allowed to overlap. A few moments' thought makes it clear that such a notion of granularity would be extremely expensive to implement because of the setting and testing of something equivalent to semaphores. There is a suggestion to remove the need for semaphores: sometimes referred to as "Reynold's rule", the idea is to require no more than one reference (on the left or right hand sides) in any assignment to potentially shared variables. Section 1.6 argues that not even variable access or change are necessarily atomic; but even without opening this facet to investigation, one can observe that Reynold's rule is also arbitrary and prohibits many completely safe programs.

Thus, for the purposes of this section, assignment statements are not assumed to be executed in an atomic step. If then a variable x has the value 0 before two assignment statements

$$x \leftarrow x + 1 \,\|\, x \leftarrow x + 2$$

are executed in parallel, what can be said of the final value of x? In the simplest case, where one parallel assignment happens to complete before the other begins, the result is $x = 3$; but if both parallel assignments have their right hand sides evaluated in the same state ($x = 0$) then the resulting value of x could be 1 or 2 depending on the order of the state changes.[1]

Some computer scientists recoiled at the difficulty of such shared state concurrency and their idea of stateless communicating processes might appear to finesse the problem illustrated above. Unfortunately, escaping the general notion of interference is not so easy. In fact, since processes can be used to model variables, it is obvious that interference is still an issue. The shared variable problem above can be precisely mirrored in, for example, the π-calculus [MPW92] as follows

[1] Atomicity of update of scalar values is assumed – for now!

$$(\overline{x}0 \mid !x(v).(\overline{r_x}v.\overline{x}v + w_x(n).\overline{x}n)) \mid r_x(v).\overline{w_x}v + 1 \mid r_x(v).\overline{w_x}v + 2$$

One might argue that assertions over communication histories are easier to write and reason about than those over state evolutions but the issue of interference has clearly not been avoided. Furthermore, interference affects liveness arguments as well as safety reasoning.

1.3 Reasoning about interference

Before coming to explicit reasoning about interference, it is instructive to review some of the early attempts to prove that shared-variable concurrent programs satisfy specifications. One way of proving that two concurrent programs are correct with respect to an overall specification is to consider their respective flow diagrams and to associate an assertion with every pair of arcs (i.e. quiescent points). So with sc_1 having n steps and sc_2 having m, it is necessary to consider $n \times m$ steps of proof. This is clearly wasteful and does not scale at all to cases where there are more than two processes. There is also here an assumption about granularity which is dangerous: are the steps in the flow diagram to be whole assignments? For the current purposes, however, the more fundamental objection is that the approach is non-compositional: proofs about the two processes can only be initiated once their final code is present; nothing can be proved at the point in time where the developer chooses to split the overall task into two parallel processes; there is no separate and complete statement of what is required of each of the sc_i.

Susan Owicki's thesis [Owi75] proposes a method which offers some progress. Normally referred to as the Owicki-Gries method because of the paper she wrote [OG76] with her supervisor David Gries, the idea is to write normal pre/post condition specifications of each of the sc_i and develop their implementations separately with normal sequential proof rules. Essentially, this first step can be thought of as considering the implementation as though it is a non-deterministic choice between one of two sequential implementations: sc_1; sc_2 or sc_2; sc_1. Having completed the top level decomposition, developments of the separate sc_i can be undertaken to obtain code which satisfies their specifications. So far, so good — but then the Owicki-Gries method requires that each program step in sc_i must be shown not to interfere with any proof step in sc_j. With careful design, many of these checks will be trivial so the worrying product of $n \times m$ checks is not as daunting. It is however again clear that this method is non-compositional in that a problem located in the final proof of "interference freedom" could force a development of sc_i to be discarded because of a decision in the design of sc_j. In other words, the specification of sc_i was incomplete in that a development which satisfied its pre- and post-condition has to be reworked at the end because it fails some criteria not present in its specification.

Several authors took up the challenge of recording assumptions and commitments which include a characterisation of interference. In [FP78], an interference constraint has to be found which is common to all processes. In [Jon81], pre/post

conditions specifications for such processes are extended with rely and guarantee conditions. The subsequent description here is in terms of the rely/guarantee proposal.

The basic idea is very simple. Just as a pre-condition records assumptions the developer can make about the initial state when designing an implementation, a rely condition records assumptions that can be made about interference from other processes: few programs can work in an arbitrary initial condition; only vacuous specifications can be met in the presence of arbitrary interference. Thus pre- and rely conditions record assumptions that the developer can make.

Just as post-conditions document commitments which must be (shown to be) fulfilled by the implementation, the interference which can be generated by the implementation is captured by writing a guarantee condition.

A specification of a component C then is written $\{p, r\}\ C\ \{g, q\}$ for a pre-condition p, a rely condition r, a guarantee condition g, and a post-condition q. It has always been the case in VDM that post-conditions were predicates of the initial and final states [2]:

$$q : \Sigma \times \Sigma \to \mathbb{B}$$

Since they record (potential) state changes, it is natural that rely and guarantee conditions are both relations:

$$r : \Sigma \times \Sigma \to \mathbb{B}$$

$$g : \Sigma \times \Sigma \to \mathbb{B}$$

Pre-conditions indicate whether an initial state is acceptable and are thus predicates of a single state:

$$p : \Sigma \to \mathbb{B}$$

The compositional proof rule for decomposing a component into two parallel components is presented in Fig. 1.1. It is more complicated than rules for sequential constructs but is not difficult to understand. If $S_1 \mid\mid S_2$ has to tolerate interference r, the component S_1 can only assume the bound on interference to be $r \vee g_2$ because steps of S_2 also interfere with S_1. The guarantee condition g of the parallel construct cannot be stronger than the disjunction of the guarantee conditions of the components. Finally, the post-condition of the overall construct can be derived from the conjunction of the individual post-conditions, conjoined with the transitive closure of the rely and guarantee conditions, and further conjoined with any information that can be brought forward from the pre-condition $\overleftarrow{p}$.

There are more degrees of freedom in the presentation of such a complex rule than those for sequential constructs, and papers listed below experiment with various presentations. It was however recognised early that there were useful generic thinking tools for reasoning about concurrent systems. "Dynamic invariants" are

[2] See [Jon99] for discussion.

$$\boxed{||}\ \frac{\begin{array}{l}\{p, r \vee g_2\}\ S_1\ \{g_1, q_1\} \\ \{p, r \vee g_1\}\ S_2\ \{g_2, q_2\} \\ g_1 \vee g_2 \Rightarrow g \\ \overleftarrow{p} \wedge q_1 \wedge q_2 \wedge (r \vee g_1 \vee g_2)^* \Rightarrow q\end{array}}{\{p, r\}\ (S_1 \parallel S_2)\ \{g, q\}}$$

Figure 1.1. A proof rule for rely/guarantee conditions

the best example of a concept which is useful in formal and informal developments alike. A dynamic invariant is a relation which holds between the initial state and any which can arise. It is thus reflexive and composes with the guarantee conditions of all processes. It is accepted by many who have adopted methods like VDM that standard data type invariants are a valuable design aid and their discussion even in informal reviews often uncovers design errors. There is some initial evidence that similar design pay off comes from dynamic invariants. In fact, they have even been seen as beneficial in the design of sequential systems (e.g. [FJ98]).

There have been many excellent contributions to the rely/guarantee idea in the twenty years since it was first published ([Jon83] is a more accessible source than [Jon81]). Ketil Stølen tackled the problem of progress arguments in his thesis [Stø90]. Xu Quiwen [Xu92] in his Oxford thesis covers some of the same ground but also looks at the use of equivalence proofs. Pierre Collette's thesis was done under the supervision of Michel Sintzoff: [Col94] makes the crucial link to Misra and Chandy's Unity language (see [CM88]). Colin Stirling tackles the issue of Cook completeness in [Sti88], and in [Sti86] shows that the same broad form of thinking can be applied to process algebras. Recent contributions include [Din00][3] and [BB99].

Returning to the fact that there have been other assumption-commitment approaches which record interference in ways different from the specific rely-guarantee conditions used here, the reader is referred to the forthcoming book from de Roever and colleagues for a review of many approaches. As far as this author is aware, none of the recorded approaches avoids the difficulties discussed in the following sections.

1.4 Some problems with assumption/commitment reasoning

In spite of the progress with rely-guarantee specifications and development, much remains to be done. It should not be surprising that reasoning about intimate in-

[3]Note [Sti88, Din00] employ unary predicates and experience the problems that are familiar from unary post-conditions when wanting to state requirements such as variables not changing.

terference between two processes can be tricky. An illustration of the delicacy of placing clauses in assumptions and commitments is given in [CJ00]. Perhaps much of what is required here is experience and the classification of types of interference.

One obvious conclusion is to limit interference in a way that makes it possible to undertake much program development with sequential rules. This echoes the message that Dijkstra et al. were giving over the whole early period of writing concurrent programs. One avenue of research in this direction has been to deploy object-based techniques to provide a way of controlling interference; this work is outlined –and additional references are given– in [Jon96].

Turning to the rules for the parallel constructs, that given in Figure 1.1 is only one with which various authors who are cited above have experimented. There more degrees of freedom than with rules for sequential constructs.[4] Again, experiments should indicate the most usable rules.

There are some general developments to be looked at in combination with any form of assumption-commitment approach. One is the need to look at their use in real-time programs. Intuitively, the same idea should work, but determining the most convenient logic in which to record assumptions and commitments might take considerable experimentation. Another extension which would require careful integration is that to handle probabilistic issues. This is of particular interest to the current author because –as described in [Jon00]– of the desire to cover "faults as interference".

1.5 The role of ghost variables

A specific problem which arises in several approaches to proofs about concurrency is finding some way of referring to points in a computation. A frustratingly simple example is the parallel execution of two assignment statements which are, for this section, assumed to be atomic.

$$\langle x \leftarrow x + 1 \rangle \,||\, \langle x \leftarrow x + 2 \rangle$$

The subtlety here is that because both increments are by the same amount one cannot use the value to determine which arm has been executed. A common solution to such issues is to introduce some form of "ghost variable" which can be modified so as to track execution. There are a number of unresolved questions around ghost variables including exactly when they are required; what is the increase in expressivity and their relationship to compositionality.

For the specific example above, this author has suggested that it might be better to avoid state predicates altogether and recognise that the important fact is that the assignments commute. Of course, if one branch incremented x and the other

[4]There is of course some flexibility with sequential constructs such as whether to fold the consequence rule into those for each programming construct.

multiplied it by some value, then they would not commute; but it would also be difficult to envisage what useful purpose such a program would have. So the proposal is that reasoning about concurrency should not rely solely on assertions about states; other –perhaps more algebraic techniques– can also be used to reason about the joint effect of actions in parallel processes.

1.6 Granularity concerns

Issues relating to granularity have figured in the discussion above and they would appear to pose serious difficulties for many methods. The problems with assuming that assignment statements can be executed atomically are reviewed in Section 1.2 but the general issue is much more difficult. For example, it is not necessarily true that variables can be read and changed without interference. This should be obvious in the case of say arrays but it is also unlikely that hardware will guarantee that long strings are accessed in an uninterrupted way. There is even the danger that scalar arithmetic value access can be interrupted.

Having ramified the problem, what can be done about it? Any approach which requires recognising the complete proof of one process to see whether another process can interfere with proof steps appears to be committed to low level details of the implementation language. To some extent, a rely-guarantee approach puts the decision about granularity in the hands of the designer. In particular, assertions carried down as relations between states can be reified later in design. This works well in the object-based approach described in [Jon96]. But granularity is a topic which deserves more research rather than the regrettable tendency to ignore the issue.

1.7 Atomicity as an abstraction, and its refinement

As well as rely and guarantee conditions, the object-based design approach put forward in [Jon96] employs equivalence transformations. The idea is that a relatively simple process could be used to develop a sequential program which can be transformed into an equivalent concurrent program. The task of providing a semantic underpinning in terms of which the claimed equivalences could be proved to preserve observational equivalence proved difficult (see for example [PW98, San99]).

The key to the equivalences is to observe that under strict conditions, islands of computation exist and interference never crosses the perimeter of the island. One of the reasons that these equivalences are interesting is that their essence – which is the decomposition of things which it is easy to see posses some property when executed atomically– occurs in several other areas. In particular, "atomicity" is a useful design abstraction in discussing database transactions and cache coherence: showing how these "atoms" can overlap is an essential part of justi-

fying a useful implementation. There are other approaches to this problem such as [JPZ91, Coh00]; but the ubiquity of atomicity refinement as a way of reasoning about some concurrency problems suggests that there is a rich idea lurking here.

1.8 Conclusion

The general idea behind assumption/commitment specifications and proof rules would appear to be a useful way of designing concurrent systems. Much detailed research and experimentation on practical problems is still required to come up with some sort of agreed approach. Even as a proponent of one of the assumption (rely) commitment (guarantee) approaches, the current author recognises that there are also quite general problems to be faced before a satisfactory compositional approach to the development of concurrent programs can be claimed. One area of extension is to look for more expressiveness whether to merge with real-time logics or to cope with probabilities. Another issue is that of arguments which do not appear to be dealt with well by assertions about states. In all of this search for formal rules, one should continue to strive for things which can be adopted also informally as thinking tools by engineers.

Acknowledgements

I gratefully acknowledge the financial support of the UK EPSRC for the Interdisciplinary Research Collaboration "Dependability of Computer-Based Systems". Most of my research is influenced by, and has been discussed with, members of IFIP's Working Group 2.3.

References

[BB99] Martin Buechi and Ralph Back. Compositional symmetric sharing in B. In *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 431–451. Springer-Verlag, 1999.

[BG91] J. C. M. Baeten and J. F. Groote, editors. *CONCUR'91 – Proceedings of the 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction*, chapter 10, pages 275–305. MIT Press, 2000.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Coh00] Ernie Cohen. Separation and reduction. In *Mathematics of Program Construction, 5th International Conference, Portugal, July 2000. Science of Computer Programming*, pages 45–59. Springer-Verlag, 2000.

[Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY*. PhD Thesis, Louvain-la-Neuve, June 1994.

[Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD Thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.

[dRE99] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, 1999.

[FJ98] John Fitzgerald and Cliff Jones. A tracking system. In J. C. Bicarregui, editor, *Proof in VDM: Case Studies*, FACIT, pages 1–30. Springer-Verlag, 1998.

[FP78] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD Thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP '83*, pages 321–332. North-Holland, 1983.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon99] C. B. Jones. Scientific decisions which characterize VDM. In *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 28–47. Springer-Verlag, 1999.

[Jon00] C. B. Jones. Thinking tools for the future of computing science. In *Informatics — 10 Years Back, 10 Years Forward*, volume 2000 of *Lecture Notes in Computer Science*, pages 112–130. Springer-Verlag, 2000.

[JPZ91] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In *[BG91]*, pages 298–316, 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD Thesis, Department of Computer Science, Cornell University, 1975. 75-251.

[PW98] Anna Philippou and David Walker. On transformations of concurrent-object programs. *Theoretical Computer Science*, 195:259–289, 1998.

[San99] Davide Sangiorgi. Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. *Theory and Practice of Object Systems*, 5(1):25–34, 1999.

[Sti86] C. Stirling. A compositional reformulation of Owicki-Gries' partial correctness logic for a concurrent while language. In *ICALP '86*. Springer-Verlag, 1986. LNCS 226.

[Sti88] C. Stirling. A generalisation of Owicki-Gries's Hoare logic for a concurrent while language. *TCS*, 58:347–359, 1988.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD Thesis, Manchester University, 1990. available as UMCS-91-1-1.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming*. PhD Thesis, Oxford University, 1992.

2

Enforcing behavior with contracts

Ralph-Johan Back and Joakim von Wright

Abstract

Contracts have been introduced earlier as a way of modeling a collection of agents that work within the limits set by the contract. We have analyzed the question of when an agent or a coalition of agents can reach a stated goal, despite potentially hostile behavior by the other agents. In this paper, we extend the model so that we can also study whether a coalition of agents can enforce a certain temporal behavior when executing a contract. We show how to reduce this question to the question of whether a given goal can be achieved. We introduce a generalization of the action system notation that allows both angelic and demonic scheduling of actions. This allows us to model concurrent systems and interactive systems in the same framework, and show that one can be seen as the dual of the other. We analyze enforcement of temporal behavior in the case of action systems, and show that these provide for simpler proof obligations than what we get in the general case. Finally, we give three illustrative examples of how to model and analyze interactive and concurrent systems with this approach.

2.1 Introduction

A computation can generally be seen as involving a number of agents (programs, modules, systems, users, etc.) who carry out actions according to a document (specification, program) that has been laid out in advance. When reasoning about a computation, we can view this document as a contract between the agents. We have earlier described a general notation for contracts, and have given these a formal meaning using an operational semantics [6]. Given a contract, we can analyze what goals a specific agent or coalition of agents can achieve with the contract. This will essentially amount to checking whether an agent or a coalition of agents have a winning strategy to reach the given goal.

In this paper, we consider the question of whether an agent or a coalition of agents can *enforce* a certain temporal behavior on the execution of the contract.

A. McIver et al. (eds.), *Programming Methodology*

This means that there is a way for these agents to co-ordinate their decisions, so that the temporal property will hold for the whole execution of the contract. We show how to model temporal properties with an operational semantics for contracts, and then study how to prove that a certain temporal property can be enforced. We show that enforcement of a temporal property can be reduced to the question of achieving a goal, which in turn can be established with standard techniques that we have developed in earlier work.

We then introduce a generalization of the *action system* notation which unifies the notion of a concurrent and an interactive system. Both kinds of systems are essentially initialized loops, but the difference comes from whether the scheduling of the loop is demonic (in concurrent systems) or angelic (in interactive systems). We show how to analyze temporal properties of the special kinds of contract that action systems provide. It turns out that we get considerable simplification in the proof obligations by using action systems rather than general contracts.

Finally, we illustrate the approach by considering three examples. The first example is the game of Nim, which illustrates the interaction of two agents in a game-playing situation. The second example is the familiar puzzle of the wolf, the goat and the cabbages, which have to be transported across a river. The last example illustrates how to apply the approach described here to a resource allocation situation, here exemplified by an imaginary Chinese Dim Sun restaurant.

Our notion of contracts is based on the refinement calculus [3, 6, 18]. We have earlier extended the original notion of contracts to consider coalitions of agents [8]. Here we combine contracts and the idea of considering a system as a game between two players [1, 20, 5, 21] with the idea of temporal properties in a predicate transformer setting [14, 19].

The paper first introduces the notion of contracts, both informally and with a precise operational semantics, in Section 2. *Action systems* are described as a special kind of contract, and we give three examples of action systems, which we will analyze in more detail later on. Section 3 shows how to analyze what kind of goals can be *achieved* with contracts, introducing a weakest precondition semantics for contracts for this purpose. In Section 4 we develop the main theme of this paper: how to show that temporal properties can be *enforced* during execution of a contract. Section 5 looks at enforcement in the special case when the contracts are action systems, showing that we can get simplified proof conditions in this case. Section 6 looks at the practice of verifying enforcement properties, and illustrates the basic proof methods by showing specific enforcement properties for the example action systems introduced earlier. We conclude with some general remarks in Section 7.

We use *simply typed higher-order logic* as the logical framework in the paper. The type of functions from a type Σ to a type Γ is denoted by $\Sigma \to \Gamma$. Functions can be described using λ-abstraction and we write $f.x$ for the application of function f to argument x.

2.2 Contracts

In this section we give an overview of contracts and their operational semantics, following [7] (with some notational changes) and introduce action systems as a special kind of contract.

2.2.1 States and state changes

We assume that the world that contracts talk about is described as a *state* σ. The *state space* Σ is the set (type) of all possible states. An agent changes the state by applying a function f to the present state, yielding a new state $f.\sigma$. We think of the state as having a number of *attributes* $x_1, \ldots, x_n$, each of which can be observed and changed independently of the others. Such attributes are usually called *program variables*. An attribute x of type Γ is really a pair of two functions, the *value function* $val_x : \Sigma \to \Gamma$ and the *update function* $set_x : \Gamma \to \Sigma \to \Sigma$. The function val_x returns the value of the attribute x in a given state, while the function set_x returns a new state where x has a specific value, with the values of all other attributes left unchanged. Given a state σ, $val_x.\sigma$ is thus the value of x in this state, while $\sigma' = set_x.\gamma.\sigma$ is the new state that we get by setting the value of x to γ.

An *expression* like $x + y$ is a function on states, described by $(x + y).\sigma = val_x.\sigma + val_y.\sigma$. We use expressions to observe properties of the state. They are also used in *assignments* like $x := x+y$. This assignment denotes a state-changing function that updates the value of x to the value of the expression $x + y$. Thus

$$(x := x + y).\sigma \;=\; set_x.(val_x.\sigma + val_y.\sigma).\sigma$$

A function $f : \Sigma \to \Sigma$ that maps states to states is called a *state transformer*. We also make use of predicates and relations over states. A *state predicate* is a boolean function $p : \Sigma \to \mathsf{Bool}$ on the state (in set notation we write $\sigma \in p$ for $p.\sigma$). Predicates are ordered by inclusion, which is the pointwise extension of implication on the booleans.

A *boolean expression* is an expression that ranges over truth values. It gives us a convenient way of describing predicates. For instance, $x \leq y$ is a boolean expression that has value $val_x.\sigma \leq val_y.\sigma$ in a given state σ.

A *state relation* $R : \Sigma \to \Sigma \to \mathsf{Bool}$ relates a state σ to a state σ' whenever $R.\sigma.\sigma'$ holds. Relations are ordered by pointwise extension from predicates. Thus, $R \subseteq R'$ holds iff $R.\sigma \subseteq R'.\sigma$ for all states σ.

We permit a generalized assignment notation for relations. For example,

$$(x := x' \mid x' > x + y)$$

relates state σ to state σ' if the value of x in σ' is greater than the sum of the values of x and y in σ and all other attributes are unchanged. More precisely, we have that

$$(x := x' \mid x' > x + y).\sigma.\sigma' \;\equiv$$

$$(\exists x' \bullet \sigma' = set_x.x'.\sigma \ \wedge \ x' > val_x.\sigma + val_y.\sigma)$$

This notation generalizes the ordinary assignment; we have that $\sigma' = (x := e).\sigma$ iff $(x := x' \mid x' = e).\sigma.\sigma'$.

2.2.2 *Contracts*

Consider a collection of *agents*, each with the capability to change the state by choosing between different *actions*. The behavior of agents is regulated by *contracts*.

Assume that there is a fixed collection Ω of agents, which are considered to be atomic (we assume that we can test for equality between agents). We let A range over sets of agents and a, b, c over individual agents.

We describe contracts using a notation for *contract statements*. The syntax for these is as follows:

$$S ::= \langle f \rangle \mid \mathsf{if}\ p\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid S_1 ; S_2 \mid \langle R \rangle_a \mid S_1 \,[]_a\, S_2 \mid (\mathsf{rec}_a\, X \bullet S) \mid X$$

Here a stands for an agent while f stands for a state transformer, p for a state predicate, and R for a state relation, all expressed using higher-order logic. X is a variable that ranges over (the meaning of) contracts.

Intuitively, a contract statement is carried out ("executed") as follows. The *functional update* $\langle f \rangle$ changes the state according to the state transformer f, i.e., if the initial state is σ_0 then the final state is $f.\sigma_0$. An *assignment statement* is a special kind of update where the state transformer is expressed as an assignment. For example, the assignment statement $\langle x := x + y \rangle$ (or just $x := x + y$ — for simplicity, we may drop the angle brackets from assignment statements) requires the agent to set the value of attribute x to the sum of the values of attributes x and y. We use the name skip for the identity update $\langle \mathsf{id} \rangle$, where $\mathsf{id}.\sigma = \sigma$ for all states σ.

In the *conditional composition* if p then S_1 else S_2 fi, S_1 is carried out if p holds in the initial state, and S_2 otherwise.

Relational update and choice introduce nondeterminism into the language of contracts. Both are indexed by an agent which is responsible for deciding how the nondeterminism is resolved.

The *relational update* $\langle R \rangle_a$ requires the agent a to choose a final state σ' so that $R.\sigma.\sigma'$ is satisfied, where σ is the initial state. In practice, the relation is expressed as a relational assignment. For example, $\langle x := x' \mid x' < x \rangle_a$ expresses that the agent a is required to decrease the value of the program variable x without changing the values of the other program variables. If it is impossible for the agent to satisfy this, then the agent has *breached* the contract. In this example, agent a must breach the contract if x ranges over the natural numbers and its initial value is 0.

An important special case of relational update occurs when the relation R is of the form $(\lambda\, \sigma\ \sigma' \bullet \sigma' = \sigma \wedge p.\sigma)$ for some state predicate p. In this case, $\langle R \rangle_a$ is called an *assertion* and we write it simply as $\langle p \rangle_a$. For example, $\langle x + y = 0 \rangle_a$ expresses that the sum of (the values of) x and y in the state must be zero. If the

assertion holds at the indicated place when the agent a carries out the contract, then the state is unchanged, and the rest of the contract is carried out. If, on the other hand, the assertion does not hold, then the agent has breached the contract. The assertion $\langle \mathsf{true} \rangle_a$ is always satisfied, so adding this assertion anywhere in a contract has no effect. Dually, $\langle \mathsf{false} \rangle_a$ is an impossible assertion; it is never satisfied and always results in the agent breaching the contract.

A *choice* $S_1 \,[]_a\, S_2$ allows agent a to choose which is to be carried out, S_1 or S_2. To simplify notation, we assume that sequential composition binds stronger than choice in contracts.

In the *sequential composition* $S_1 ; S_2$, contract S_1 is first carried out, followed by S_2, provided that there is no breach of contract when executing S_1. We also permit *recursive contract statements*. A recursive contract is essentially an equation of the form

$$X =_a S$$

where S may contain occurrences of the contract variable X. With this definition, the contract X is intuitively interpreted as the contract statement S, but with each occurrence of statement variable X in S treated as a recursive invocation of the whole contract S. For simplicity, we use the syntax $(\mathsf{rec}_a\, X \bullet S)$ for the contract X defined by the equation $X = S$.

The index a for the recursion construct indicates that agent a is responsible for termination of the recursion. If the recursion does not terminate, then a will be considered as having breached the contract. In general, agents have two roles in contracts: (i) they choose between different alternatives that are offered to them, and (ii) they take the blame when things go wrong. These two roles are interlinked, in the sense that things go wrong when an agent has to make a choice, and there is no acceptable choice available.

An important special case of recursion is the *while-loop* which is defined in the usual way:

$$\mathsf{while}_a\ p\ \mathsf{do}\ S\ \mathsf{od} \triangleq (\mathsf{rec}_a\, X \bullet \mathsf{if}\ p\ \mathsf{then}\ S ; X\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi})$$

Note the occurrence of agent a in the loop syntax; this agent is responsible for termination (so nontermination of the loop is not necessarily a bad thing).

2.2.3 *Operational semantics*

We give a formal meaning to contract statements in the form of a structured operational semantics. This semantics describes step-by-step how a contract is carried out, starting from a given initial state.

The rules of the operational semantics are given in terms of a transition relation between configurations. A *configuration* is a pair (S, σ), where

- S is either an ordinary contract statement or the empty statement symbol Λ, and

- σ is either an ordinary state, or the symbol $\bot_a$ (indicating that agent a has breached the contract).

The transition relation $\rightarrow$ (which shows what moves are permitted) is inductively defined by a collection of axioms and inference rules. It is the smallest relation which satisfies the following (where we assume that σ stands for a proper state while γ stands for either a state or the symbol $\bot_x$ for some agent x):

- *Functional update*

$$\frac{}{(\langle f\rangle, \sigma) \rightarrow (\Lambda, f.\sigma)} \qquad \frac{}{(\langle f\rangle, \bot_a) \rightarrow (\Lambda, \bot_a)}$$

- *Conditional composition*

$$\frac{p.\sigma}{(\mathsf{if}\ p\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, \sigma) \rightarrow (S_1, \sigma)} \qquad \frac{\neg p.\sigma}{(\mathsf{if}\ p\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, \sigma) \rightarrow (S_2, \sigma)}$$

$$\frac{}{(\mathsf{if}\ p\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, \bot_a) \rightarrow (\Lambda, \bot_a)}$$

- *Sequential composition*

$$\frac{(S_1, \gamma) \rightarrow (S_1', \gamma'), \quad S_1' \neq \Lambda}{(S_1\ ;\ S_2, \gamma) \rightarrow (S_1'\ ;\ S_2, \gamma')} \qquad \frac{(S_1, \gamma) \rightarrow (\Lambda, \gamma')}{(S_1\ ;\ S_2, \gamma) \rightarrow (S_2, \gamma')}$$

- *Relational update*

$$\frac{R.\sigma.\sigma'}{(\langle R\rangle_a, \sigma) \rightarrow (\Lambda, \sigma')} \qquad \frac{R.\sigma = \varnothing}{(\langle R\rangle_a, \sigma) \rightarrow (\Lambda, \bot_a)} \qquad \frac{}{(\langle R\rangle_a, \bot_b) \rightarrow (\Lambda, \bot_b)}$$

- *Choice*

$$\frac{}{(S_1\ []_a\ S_2, \gamma) \rightarrow (S_1, \gamma)} \qquad \frac{}{(S_1\ []_a\ S_2, \gamma) \rightarrow (S_2, \gamma)}$$

- *Recursion*

$$\frac{}{((\mathsf{rec}_a\ X \bullet S), \gamma) \rightarrow (S[X := (\mathsf{rec}_a\ X \bullet S)], \gamma)}$$

A *scenario* for the contract S in initial state σ is a sequence of configurations

$$C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \cdots$$

where

1. $C_0 = (S, \sigma)$,
2. each transition $C_i \rightarrow C_{i+1}$ is permitted by the axiomatization above, and
3. if the sequence is finite with last configuration C_n, then $C_n = (\Lambda, \gamma)$, for some γ.

Intuitively, a scenario shows us, step by step, what choices the different agents have made and how the state is changed when the contract is being carried out. A finite scenario cannot be extended, since no transitions are possible from an empty configuration.

2.2.4 *Examples of contracts*

Programs can be seen as special cases of contracts, where two agents are involved, the *user* and the *computer*. In simple batch-oriented programs, choices are only made by the computer, which resolves any internal choices (nondeterminism) in a manner that is unknown to the user. Our notation for contracts already includes assignment statements and sequential composition. The *abort* statement of Dijkstra's guarded commands language [11] can be expressed as $\mathsf{abort} = \{\mathsf{false}\}_{user}$. If executed, it signifies that there has been a breach of contract by the user. This will release the computer from any obligations to carry out the rest of the contract, i.e., the computer is free to do whatever it wants. The abort statement thus signifies misuse of the computer by the user.

A batch-oriented program does not allow for any user interaction during execution. Once started, execution proceeds to the end if possible, or it fails because the contract is breached (allowing the computer system to do anything, including going into an infinite loop).

An interactive program allows the user to make choices during the execution. The user chooses between alternatives in order to steer the computation in a desired direction. The computer system can also make choices during execution, based on some internal decision mechanism that is unknown the user, so that she cannot predict the outcome.

As an example, consider the contract

$$\begin{aligned} S &= S_1 \,;\, S_2, \text{ where} \\ S_1 &= (x := x + 1 \;[]_a\; x := x + 2) \\ S_2 &= (x := x - 1 \;[]_b\; x := x - 2) \end{aligned}$$

Computing the operational semantics for S results in the tree shown in Fig. 2.1. After initialization, the user a chooses to increase the value of x by either one or two. After this, the computer b decides to decrease x by either one or two. The choice of the user depends on what she wants to achieve. If, e.g., she is determined that x should not become negative, she should choose the second alternative. If, again, she is determined that x should not become positive, she should choose the first alternative. We can imagine this user interaction as a *menu choice* that is presented to the user after the initialization, where the user is requested to choose one of the two alternatives.

We could also consider b to be the user and a to be the computer. In this case, the system starts by either setting x to one or to two. The user can then inspect the new value of x and choose to reduce it by either 1 or 0, depending on what she tries to achieve.

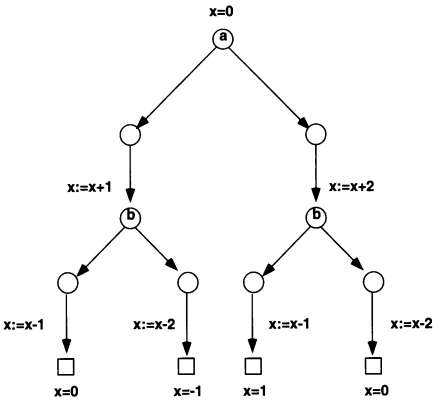


Figure 2.1. Executing contract S

A more general way for the user to influence the computation is to give input to the program during its execution. This can be achieved by a relational assignment. The following contract describes a typical interaction:

$$\langle x, e := x', e' \mid x' \geq 0 \wedge e > 0 \rangle_a \,;$$
$$\langle x := x' \mid -e < x'^2 - x < e \rangle_b$$

The user a gives as input a value x whose square root is to be computed, as well as the precision e with which the computer is to compute this square root. The computer b then computes an approximation to the square root with precision e. The computer may choose any new value for x that satisfies the required precision.

This simple contract thus *specifies* the interaction between the user and the computer. The first statement specifies the user's responsibility (to give an input value that satisfies the given conditions) and the second statement specifies the computer's responsibility (to compute a new value for x that satisfies the given condition).

The use of contracts allows user and computer choices to be intermixed in any way. In particular, the user choices can depend on previous choices by the computer and vice versa, and the choices can be made repeatedly within a loop, as we will show later.

Of course, there is nothing in this formalism that requires one of the agents to be a computer and the other to be a user. Both agents could be humans, and both agents could be computers. There may also be any number of agents involved in a contract, and it is also possible to have contracts with no agents.

2.2.5 *Action systems*

An *action system* is a contract of the form

$$\mathcal{A} = (\mathsf{rec}_c\, X \bullet S \,;\, X \,[]_a \, \langle g \rangle_a)$$

The contract S inside $\mathcal{A}$ is iterated as long as agent a wants. Termination is normal if the *exit condition* g holds when a decides to stop the iteration, otherwise a will fail, i.e. breach the contract (however, we assume that an agent a will never make choices that lead to a breaching a contract, if she can avoid it; for a justification of this assumption we refer to [8]). Agent c gets the blame if the execution does not terminate.

In general, we also allow an action system to have an *initialization (begin)* statement B and a *finalization (end)* statement E, in addition to the *action* statement S. The initialization statement would typically introduce some local variables for the action system, and initialize these. The finalization statement would usually remove these local variables. The action statement S can in turn be a choice statement,

$$S = S_1 \,[]_b\, S_2 \,[]_b\, \ldots \,[]_b\, S_n$$

We refer to each S_i here as an *action* of the system. As a notational convenience, we write

$$g_1 \rightarrow S_1 \,[]_b\, \ldots \,[]_b\, g_m \rightarrow S_m$$

for

$$\langle g_1 \rangle_b \,;\, S_1 \,[]_b\, \ldots \,[]_b\, \langle g_m \rangle_b \,;\, S_m$$

Thus, an action system is in general of the form

$$\mathcal{A}(a,b,c) = B \,;\, (\mathsf{rec}_c\, X \bullet (S_1 \,[]_b\, S_2 \,[]_b\, \ldots \,[]_b\, S_n) \,;\, X \,[]_a\, \langle g \rangle_a) \,;\, E$$

2.2.6 *Examples of action systems*

We present here three examples of action systems. The first example, Nim, illustrates a game, where the question to be decided is whether and when a player has a winning strategy. The second example is a classical puzzle, the Wolf, Goat and Cabbages, and illustrates a purely interactive system. The last example illustrates an imaginary Chinese Dim Sun restaurant.

The game of Nim

In the game of Nim, two players take turns removing sticks from a pile. A player can remove one or two sticks at a time, and the player who removes the last stick loses the game.

We model the players as two agents a and b. Agent c is the scheduler, who decides which player makes the first move and agent z is responsible for termination. The game is then described by the following contract:

$$\begin{aligned}
\mathit{Nim} \;&=\; (f := \mathsf{T} \,[]_c\, f := \mathsf{F}) \,;\, \mathit{Play} \\
\mathit{Play} \;&=_z\; f \wedge x > 0 \rightarrow \langle x := x' \mid x' < x \leq x' + 2 \rangle_a \,;\, f := \neg f \,;\, \mathit{Play} \\
&\quad\; []_c\, \neg f \wedge x > 0 \rightarrow \langle x := x' \mid x' < x \leq x' + 2 \rangle_b \,;\, f := \neg f \,;\, \mathit{Play} \\
&\quad\; []_c\, x = 0 \rightarrow \mathsf{skip}
\end{aligned}$$

Note that this only describes the moves and the scheduling. Notions like winning and losing are modeled using properties to be established and are part of the analysis of the system. Also note that the initial number of sticks is left unspecified (x is not initialized).

The wolf, the goat and the cabbages

The classical puzzle of the wolf, the goat and the cabbages goes as follows: A man comes to a river with a wolf, a goat and a sack of cabbages. He wants to get to the other side, using a boat that can only fit one of the three items (in addition to the man himself). He cannot leave the wolf and the goat on the same side (because the wolf eats the goat) and similarly, he cannot leave the goat and the cabbages on the same side. The question is: can the man get the wolf, the goat, and the cabbages safely to the other side?.

We model the situation with one boolean variable for each participant, indicating whether they are on the right side of the river or not: m for the man, w for the wolf, g for the goat, and c for the cabbages. The boat does not need a separate variable, because it is always on the same side as the man. There is only one agent involved (the scheduler a, who is in practice the man). The contract that describes the situation is the following:

$$
\begin{aligned}
\mathit{CrossRiver} \;&=\; m, w, g, c := \mathsf{F}, \mathsf{F}, \mathsf{F}, \mathsf{F} \,;\, \mathit{Transport} \\
\mathit{Transport} \;&=_a\; m = w \rightarrow m, w := \neg m, \neg w \,;\, \mathit{Transport} \\
&\quad\; []_a\, m = g \rightarrow m, g := \neg m, \neg g \,;\, \mathit{Transport} \\
&\quad\; []_a\, m = c \rightarrow m, c := \neg m, \neg c \,;\, \mathit{Transport} \\
&\quad\; []_a\, m := \neg m \,;\, \mathit{Transport} \\
&\quad\; []_a\, \mathsf{skip}
\end{aligned}
$$

The initialization says that all four are on the wrong side, and each action corresponds to the man moving from one side of the river to the other, either alone or together with an item that was on the same side.

Let us have a quick look at what the man is trying to achieve. He wants to reach a situation where all four items are on the right side of the river, i.e., we want $m \wedge w \wedge g \wedge c$ to be true at some point in the execution. Furthermore, if the wolf and the goat are on the same side of the river, then the man must also be on that side. Thus, he wants the property

$$(w = g \Rightarrow m = w) \wedge (g = c \Rightarrow m = g)$$

to be true at every intermediate point of the execution.

Note that termination is nondeterministic; the man can decide to stop at any time. The fact that we want to achieve a situation where $m \wedge w \wedge g \wedge c$ holds will be part of an analysis of the system, rather than the description.

The Dim Sun restaurant

In a Dim Sun restaurant, a waiter continuously offers customers items from a tray. We assume that there are three customers a, b, and c, and that x_0, x_1 and x_2 is the number of items that they have taken, respectively (initially set to 0). The waiter d decides in what order to offer customers items, but he may not offer items to the same customer twice in a row (we let f indicate who got the last offer). The remaining number of items is r. The manager e can decide to close the restaurant at any time (he must close it when there are no items left). This gives us the following system:

$$
\begin{aligned}
\mathit{Dim\ Sun} \;=\; & x_0, x_1, x_2, f := 0, 0, 0, 3 \,;\, \mathit{Serve} \\
\mathit{Serve} \;=_z\; & \langle r > 0 \rangle_e \,; \\
& \big(\quad \langle f \neq 0 \rangle_d \,;\, (\langle x_0, r := x_0 + 1, r - 1 \rangle []_a \mathsf{skip}) \,;\, \langle f := 0 \rangle \,;\, \mathit{Serve} \\
& \;\; []_d \, \langle f \neq 1 \rangle_d \,;\, (\langle x_1, r := x_1 + 1, r - 1 \rangle []_b \mathsf{skip}) \,;\, \langle f := 1 \rangle \,;\, \mathit{Serve} \\
& \;\; []_d \, \langle f \neq 2 \rangle_d \,;\, (\langle x_2, r := x_2 + 1, r - 1 \rangle []_c \mathsf{skip}) \,;\, \langle f := 2 \rangle \,;\, \mathit{Serve} \;\big) \\
& []_e \; \mathsf{skip}
\end{aligned}
$$

2.3 Achieving goals with contracts

The operational semantics describes all possible ways of carrying out a contract. By looking at the state component of final configurations we can see what outcomes (final states) are possible, if all agents cooperate. However, in reality the different agents are unlikely to have the same goals, and the way one agent makes its choices need not be suitable for another agent. From the point of view of a specific agent or a group of agents, it is therefore interesting to know what outcomes are possible regardless of how the other agents resolve their choices.

Consider the situation where the initial state σ is given and a group of agents A agree that their common goal is to use contract S to reach a final state in some set q of desired final states. It is also acceptable that the coalition is released from the contract, because some other agent breaches the contract. This means that the agents should strive to make their choices in such a way that the scenario starting from (S, σ) ends in a configuration (Λ, γ) where γ is either an element in q, or $\bot_b$ for some $b \notin A$ (the latter indicating that some other agent has breached the contract). A third possibility is to prevent the execution from terminating, if an agent that does not belong to A is responsible for termination.

For the purpose of analysis we can think of the agents in A as being one single agent and dually, the remaining agents as also being one single agent that tries to prevent the former from reaching its goal. In [7, 8] we show how an execution of the contract can then be viewed as a two-person game and how this intuition can be formalized by interpreting contracts with two agents as predicate transformers. This section gives an overview of how this is done.

2.3.1 *Weakest preconditions*

A *predicate transformer* is a function that maps predicates to predicates. We order predicate transformers by pointwise extension of the ordering on predicates, so $F \sqsubseteq F'$ for predicate transformers holds if and only if $F.q \subseteq F'.q$ for all predicates q. The predicate transformers form a complete lattice with this ordering.

Assume that S is a contract statement and A a *coalition*, i.e., a set of agents. We want the predicate transformer $\mathsf{wp}.S.A$ to map postcondition q to the set of all initial states σ from which the agents in A have a winning strategy to reach the goal q if they co-operate. Thus, $\mathsf{wp}.S.A.q$ is the *weakest precondition* that guarantees that the agents in A can cooperate to achieve postcondition q.

The intuitive description of contract statements can be used to justify the following definition of the weakest precondition semantics:

$$\mathsf{wp}.\langle f\rangle.A.q = (\lambda\sigma \bullet q.(f.\sigma))$$

$$\mathsf{wp}.(\mathsf{if}\ p\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}).A.q = (p \cap \mathsf{wp}.S_1.A.q) \cup (\neg p \cap \mathsf{wp}.S_2.A.q)$$

$$\mathsf{wp}.(S_1\ ;\ S_2).A.q = \mathsf{wp}.S_1.A.(\mathsf{wp}.S_2.A.q)$$

$$\mathsf{wp}.\langle R\rangle_a.A.q = \begin{cases} (\lambda\sigma \bullet \exists\sigma' \bullet R.\sigma.\sigma' \wedge q.\sigma') & \text{if } a \in A \\ (\lambda\sigma \bullet \forall\sigma' \bullet R.\sigma.\sigma' \Rightarrow q.\sigma') & \text{if } a \notin A \end{cases}$$

$$\mathsf{wp}.(S_1\ []_a\ S_2).A.q = \begin{cases} \mathsf{wp}.S_1.A.q \cup \mathsf{wp}.S_2.A.q & \text{if } a \in A \\ \mathsf{wp}.S_1.A.q \cap \mathsf{wp}.S_2.A.q & \text{if } a \notin A \end{cases}$$

These definitions are consistent with Dijkstra's original semantics for the language of guarded commands [11] and with later extensions to it, corresponding to nondeterministic assignments, choices, and miracles [3, 4, 17].

The semantics of a recursive contract is given in a standard way, using fixpoints. Assume that a recursive contract statement $(\mathsf{rec}_a\ X \bullet S)$ and a coalition A are given. Since S is built using the syntax of contract statements, we can define a function that maps any predicate transformer X to the result of replacing every construct except X in S by its weakest precondition predicate transformer semantics (for the coalition A). Let us call this function $f_{S,A}$. Then $f_{S,A}$ can be shown to be a monotonic function on the complete lattice of predicate transformers, and by the well-known Knaster-Tarski fixpoint theorem it has a complete lattice of fixpoints. We then define

$$\mathsf{wp}.(\mathsf{rec}_a\ X \bullet S).A = \begin{cases} \mu.f_{S,A} & \text{if } a \in A \\ \nu.f_{S,A} & \text{if } a \notin A \end{cases}$$

We take the least fixed point μ when non-termination is considered bad (from the point of view of the coalition A), as is the case when agent $a \in A$ is responsible for termination. Dually, we take the greatest fixpoint ν when termination is considered good, i.e., when an agent not in A is responsible for termination. A more careful and detailed treatment of recursion is found in [6].

The fixpoint definition of the semantics of recursion makes use of the fact that for all coalitions A and all contracts S the predicate transformer $\mathsf{wp}.S.A$ is

monotonic, i.e.,

$$p \subseteq q \Rightarrow \mathsf{wp}.S.A.q \subseteq \mathsf{wp}.S.A.q$$

holds for all predicates p and q. This is in fact the only one of Dijkstra's original four "healthiness" properties [11] that are satisfied by all contracts.

As the predicate transformers form a complete lattice, we can define standard lattice operations on them:

$$\begin{aligned}
\mathsf{abort} &= (\lambda q \bullet \mathsf{false}) \\
\mathsf{magic} &= (\lambda q \bullet \mathsf{true}) \\
(F_1 \sqcup F_2).q &= F_1.q \cap F_2.q \\
(F_1 \sqcap F_2).q &= F_1.q \cup F_2.q
\end{aligned}$$

Here abort is the bottom of the lattice, and magic is the top of the lattice, while the two binary operations are lattice meet and lattice join for predicate transformers.

In addition to these operations, we define standard composition operators for predicate transformers:

$$\begin{aligned}
(F_1 \,;\, F_2).q &= F_1.(F_2.q) \\
\mathsf{if}\ p\ \mathsf{then}\ F_1\ \mathsf{else}\ F_2\ \mathsf{fi}.q &= (p \cap F_1.q) \cup (\neg p \cap F_2.q)
\end{aligned}$$

Finally, let us define the following constant predicate transformers:

$$\begin{aligned}
\langle f \rangle.q.\sigma &\equiv q.(f.\sigma) \\
\{R\}.q.\sigma &\equiv R.\sigma \cap q \neq \varnothing \\
[R].q.\sigma &\equiv R.\sigma \subseteq q
\end{aligned}$$

With these definitions, we can give simpler definitions of the predicate transformers for contracts, that more clearly show the homomorphic connection between the operations on contracts and the operations on predicate transformers:

$$\begin{aligned}
\mathsf{wp}.\langle f \rangle.A &= \langle f \rangle \\
\mathsf{wp}.(\mathsf{if}\ p\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}).A &= \mathsf{if}\ p\ \mathsf{then}\ \mathsf{wp}.S_1\ \mathsf{else}\ \mathsf{wp}.S_2\ \mathsf{fi} \\
\mathsf{wp}.(S_1 \,;\, S_2).A &= \mathsf{wp}.S_1.A \,;\, \mathsf{wp}.S_2.A \\
\mathsf{wp}.\langle R \rangle_a.A &= \begin{cases} \{R\} & \text{if } a \in A \\ [R] & \text{if } a \notin A \end{cases} \\
\mathsf{wp}.(S_1 \,[]_a\, S_2).A &= \begin{cases} \mathsf{wp}.S_1.A \sqcup \mathsf{wp}.S_2.A & \text{if } a \in A \\ \mathsf{wp}.S_1.A \sqcap \mathsf{wp}.S_2.A & \text{if } a \notin A \end{cases} \\
\mathsf{wp}.(\mathsf{rec}_a\, X \bullet S).A &= \begin{cases} (\mu\, X \bullet \mathsf{wp}.S.A) & \text{if } a \in A \\ (\nu\, X \bullet \mathsf{wp}.S.A) & \text{if } a \notin A \end{cases}
\end{aligned}$$

In the last definition, we assume that $\mathsf{wp}.X.A = X$, so that the fixpoint is taken over a predicate transformer.

We finally make a slight extension to the contract formalism that allows us to also have implicit agents. We postulate that the set Ω of agents always con-

tains two distinguished agents, *angel* and *demon*. Any coalition of agents A from Ω must be such that $angel \in A$ and $demon \notin A$. With this definition, we can introduce the following abbreviations for contracts:

$$\{R\} = \langle R \rangle_{angel}$$
$$[R] = \langle R \rangle_{demon}$$
$$\sqcup = []_{angel}$$
$$\sqcap = []_{demon}$$
$$\mu = \mathsf{rec}_{angel}$$
$$\nu = \mathsf{rec}_{demon}$$

This convention means that we can use predicate transformer notation directly in contracts. We will find this convention quite useful below, when we analyze the temporal properties of contracts.

2.3.2 *Correctness and winning strategies*

We say that *agents A can use contract S in initial state σ to establish postcondition q* (written $\sigma \{\!|\ S\ |\!\}_A\ q$) if there is a *winning strategy* for the agents in A which guarantees that initial configuration (S, σ) will lead to one of the following two alternatives:

(a) termination in such a way that the final configuration is some (Λ, γ) where γ is either a final state in q or $\bot_b$ for some $b \notin A$:

$$(S, \sigma) \rightarrow \cdots \rightarrow (\Lambda, \gamma) \quad \text{where } \gamma \in q \cup \{\bot_b \mid b \notin A\}$$

(b) an infinite execution caused by a recursion for which some agent $b \notin A$ is responsible.

Thus $\sigma \{\!|\ S\ |\!\}_A\ q$ means that, no matter what the other agents do, the agents in A can (by making suitable choices) achieve postcondition q, or make sure that some agent outside A either breaches the contract or causes nontermination.[1]

This is easily generalized to a general notion of *correctness*; we define *correctness of contract S for agents A, precondition p and postcondition q* as follows:

$$p \{\!|\ S\ |\!\}_A\ q \triangleq (\forall \sigma \in p \bullet \sigma \{\!|\ S\ |\!\}_A\ q)$$

The winning strategy theorem of [6] can now easily be generalized to take into account collections of agents, to give the following:

[1] If nested or mutual recursion is involved, then it may not be clear who is to blame for infinite executions. This problem is similar to the problem of how to decide who wins an infinite game. We will simply avoid nested or mutual recursion here (alternatively, we could consider Theorem 2.3.1 below to define how such situations should be interpreted).

Theorem 2.3.1 *Assume that contract statement S, coalition A, precondition p and postcondition q are given. Then* $p \{\!|\ S\ |\!\}_A\ q$ *if and only if* $p \subseteq \mathsf{wp}.S.A.q$.

Let us as an example show how to determine when agent a has a winning strategy for reaching the goal $x \geq 0$ in our example contract above. Let us as before define

$$\begin{aligned} S &= S_1\ ;\ S_2 \\ S_1 &= x := x+1\ []_a\ x := x+2 \\ S_2 &= x := x-1\ []_b\ x := x-2 \end{aligned}$$

Let $A = \{a\}$. By the rules for calculating weakest preconditions, we have that

$$\begin{aligned} \mathsf{wp}.S.A.(x \geq 0) &= \mathsf{wp}.S_1.A.(\mathsf{wp}.S_2.A.(x \geq 0)) \\ \mathsf{wp}.S_2.A.(x \geq 0) &= \langle x := x-1 \rangle.(x \geq 0) \cap \langle x := x-2 \rangle.(x \geq 0) \\ &= (x-1 \geq 0) \cap (x-2 \geq 0) \\ &= (x \geq 1) \cap (x \geq 2) \\ &= x \geq 1 \wedge x \geq 2 \\ &= x \geq 2 \\ \mathsf{wp}.S_1.A.(x \geq 2) &= \langle x := x+1 \rangle.(x \geq 2) \cup \langle x := x+2 \rangle.(x \geq 2) \\ &= (x+1 \geq 2) \cup (x+2 \geq 2) \\ &= (x \geq 1) \cup (x \geq 0) \\ &= x \geq 1 \vee x \geq 0 \\ &= x \geq 0 \end{aligned}$$

Thus, we have shown that

$$\mathsf{wp}.S.A.(x \geq 0) = x \geq 0$$

In other words, the agent a can achieve the postcondition $x \geq 0$ whenever the initial state satisfies $x \geq 0$. Thus we have shown the correctness property

$$x \geq 0\ \{\!|\ S\ |\!\}_{\{a\}}\ x \geq 0$$

From the wp-semantics and Theorem 2.3.1 it is straightforward to derive rules for proving correctness assertions, in the style of Hoare logic:

- *Functional update*

$$\frac{(\forall\, \sigma \in p \bullet q.(f.\sigma))}{p\ \{\!|\ \langle f \rangle\ |\!\}_A\ q}$$

- *Conditional composition*

$$\frac{p \cap b\ \{\!|\ S_1\ |\!\}_A\ q \qquad p \cap \neg b\ \{\!|\ S_2\ |\!\}_A\ q}{p\ \{\!|\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}\ |\!\}_A\ q}$$

- *Sequential update*

$$\frac{p \{\!|\ S_1\ |\!\}_A\ r \qquad r \{\!|\ S_2\ |\!\}_A\ q}{p \{\!|\ S_1 ; S_2\ |\!\}_A\ q}$$

- *Relational update*

$$\frac{(\forall \sigma \in p \bullet \exists \sigma' \bullet R.\sigma.\sigma' \wedge q.\sigma')}{p \{\!|\ \langle R \rangle_a\ |\!\}_A\ q}\ a \in A \qquad \frac{(\forall \sigma \in p \bullet \forall \sigma' \bullet R.\sigma.\sigma' \Rightarrow q.\sigma')}{p \{\!|\ \langle R \rangle_a\ |\!\}_A\ q}\ a \notin A$$

- *Choice*

$$\frac{p \{\!|\ S_1\ |\!\}_A\ q}{p \{\!|\ S_1 \,[\!]_a S_2\ |\!\}_A\ q}\ a \in A \qquad \frac{p \{\!|\ S_2\ |\!\}_A\ q}{p \{\!|\ S_1 \,[\!]_a S_2\ |\!\}_A\ q}\ a \in A$$

$$\frac{p \{\!|\ S_1\ |\!\}_A\ q \qquad p \{\!|\ S_1\ |\!\}_A\ q}{p \{\!|\ S_1 \,[\!]_a\, S_2\ |\!\}_A\ q}\ a \notin A$$

- *Loop*

$$\frac{p \cap b \cap t = w \{\!|\ S\ |\!\}_A\ p \cap t < w}{p \{\!|\ \mathsf{while}_a b\ \mathsf{do}\ S\ \mathsf{od}\ |\!\}_A\ p \cap \neg b}\ a \in A$$

$$\frac{p \cap b \{\!|\ S\ |\!\}_A\ p}{p \{\!|\ \mathsf{while}_a b\ \mathsf{do}\ S\ \mathsf{od}\ |\!\}_A\ p \cap \neg b}\ a \notin A$$

- *Consequence*

$$\frac{p' \subseteq p \qquad p \{\!|\ S\ |\!\}_A\ q \qquad q \subseteq q'}{p' \{\!|\ S\ |\!\}_A\ q'}$$

In the rules for the while-loop, t (the termination argument for the loop) is assumed to range over some well-founded set W, and w is a fresh variable also ranging over W.

These are close to the traditional Hoare Logic rules for total correctness. We include a rule for the while-loop rather than for recursion, for simplicity. The existential quantifier in the first rule for relational update and the existence of two alternative rules for choice (when $a \in A$) indicate that we can show the existence of a general winning strategy by providing a witness during the correctness proof. In fact, the proof encodes a winning strategy, in the sense that if we provide an existential witness (for a relational update) then we describe how the agent in question should make its choice in order to contribute to establishing the post-condition. Similarly, the selection of the appropriate rule for a choice encodes a description of how an agent should make the choice during an execution.

2.3.3 *Refinement of contracts*

The predicate transformer semantics is based on total correctness. Traditionally, a notion of *refinement* is derived from a corresponding notion of correctness, so that a refinement $S \sqsubseteq S'$ holds iff S' satisfies all correctness properties that S satisfies.

Since we define correctness for a collection of agents (whose ability to guarantee a certain outcome we are investigating), refinement will also be relativised similarly. We say that *contract S is refined by contract S' for coalition A* (written $S \sqsubseteq_A S'$), if S' preserves all correctness properties of S, for A. By Theorem 2.3.1, we have

$$S \sqsubseteq_A S' \equiv (\forall q \bullet \mathsf{wp}.S.A.q \subseteq \mathsf{wp}.S'.A.q)$$

The traditional notion of refinement [3] is here recovered in the case when the coalition A is empty; i.e., if all the nondeterminism involved is demonic. Furthermore, the generalization of refinement to include both angelic and demonic nondeterminism [4, 6] is recovered by identifying the agents in A with as the angel and the agents outside A as the demon.

Given a contract, we can use the predicate transformer formulation of refinement to derive rules that allow us to improve a contract from the point of view of a specific coalition A, in the sense that any goals achievable with the original contract are still achievable with the new contract. These refinement rules can be used for *stepwise refinement* of contracts, where we start from an initial high level specification with the aim of deriving a more efficient (and usually lower level) implementation of the specification. In this paper we do not consider refinement, as the focus is on establishing temporal properties. The refinement relation $\sqsubseteq_A$ is investigated in more detail in [8].

2.4 Enforcing behavioral properties

The previous section has concentrated on what goals an agent can achieve while following a contract. Here we will instead look at what kind of behavior an agent can enforce by following a contract.

2.4.1 Analyzing behavior

Consider again the example contract of the previous section, but now assuming that we have just an angel and a demon involved in the contract:

$$S = (x := x + 1 \sqcap x := x + 2)\,;$$
$$(x := x - 1 \sqcup x := x - 2)$$

A behavior property would, e.g., be that the angel can force the condition $0 \leq x \leq 2$ to hold in each state when executing the contract, if $x = 0$ initially. Using temporal logic notation, we can express this as

$$x = 0 \;\{\!|\; S \;|\!\}\; \Box(0 \leq x \leq 2)$$

Here $\Box(0 \leq x \leq 2)$ says that $0 \leq x \leq 2$ is *always* true. Note that this property need not hold for every possible execution, it is sufficient that there is a way for the angel to *enforce* the property by making suitable choices during the execution.

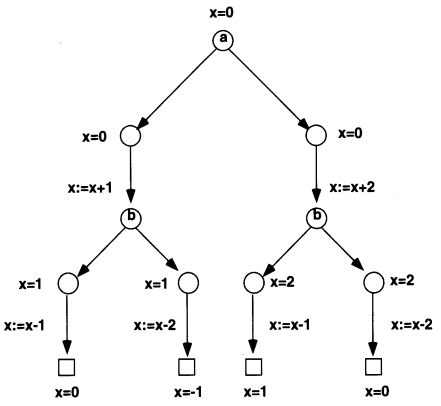


Figure 2.2. Behavior of contract

Using the operational semantics of S, we can determine all the possible execution sequences of this contract. Fig. 2.2 shows these, also indicating the value of x at each intermediate state (a is the demon, b is the angel). From this we see that the angel can indeed enforce the condition $0 \leq x \leq 2$, irrespectively of which alternative the demon chooses. If the demon chooses the left branch, then the angel should also choose the left branch, and if the demon chooses the right branch, then it does not matter which branch the angel chooses. The condition $1 \leq x \leq 2$ is, on the other hand, an example of a property that cannot be enforced by the angel.

In a similar way, we can show that the angel can also enforce that the condition $x = 1$ is *eventually* true when the initial state satisfies $x = 0$. This is expressed using temporal logic notation as

$$x = 0 \{\!|\ S\ |\!\} \Diamond(x = 1)$$

From Fig. 2.2 we see that this condition is true after two steps if the demon chooses the left alternative. If the demon chooses the right alternative, then the angel can enforce the condition upon termination by choosing the left alternative.

2.4.2 *Constructing an interpreter*

Let us consider more carefully how to check whether a temporal property can be enforced when carrying out a contract. Consider a contract statement S that operates on a state space Σ, and includes agents Ω. Let us check whether the temporal property $\Box p$ can be enforced by a coalition of agents $A \subseteq \Omega$.

For any predicate $p \subseteq \Sigma$, we define the the contract Always.p, called a *tester for* $\Box p$, by

$$\mathsf{Always}.p = (\nu X \bullet \{p\}\ ;\ [s \neq \Lambda]\ ;\ \mathit{step}\ ;\ X)$$
$$\mathit{step} = \langle s, \sigma := s', \sigma' \mid (s, \sigma) \rightarrow (s', \sigma') \rangle_{ch.s}$$

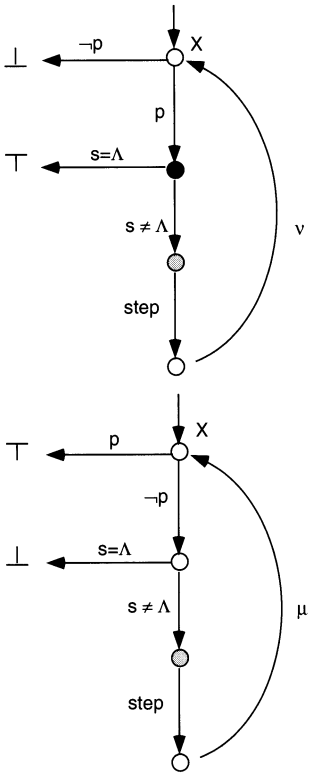


Figure 2.3. (a) A property holds always. (b) A property holds eventually

This contract operates on states τ with two components s and σ. We define $p.\tau = p.(\sigma.\tau)$. The function $ch.s$ gives the agent that makes the choice in the statement s, if there is one, otherwise there is no agent index. In other words, the tester is an *interpreter* for contract statements, which executes them with the purpose of determining whether a specific temporal property is valid.

We illustrate the behavior of the tester with the diagram in Fig. 2.3. The diagram shows the angelic choices as hollow circles, and the demonic choices as filled circles. A grey circle indicates that we do not know whether the choice is angelic or demonic. The X labels the node at which the iteration starts. The arrow labeled ν indicates that we have ν-iteration, i.e., the arrow can be traversed any number of times, without a breach of contract. An arrow labeled μ indicates μ-iteration, where the arrow can be traversed only a finite number of times during each iteration.

We now have the following general result, which we give without proof. (In a separate paper [9] we give formal definitions of behaviours and temporal properties, together with proofs of verification and refinement rules).

Lemma 2.4.1 *Let S, p, and C be as above. Let A be a coalition of agents in* Ω. *Then*

$$\sigma_0 \{\!|\ S\ |\!\}_A \Box p \equiv \mathsf{wp}.(\mathsf{Always}.p).A.\mathsf{false}.\tau$$

where $\sigma.\tau = \sigma_0$ *and* $s.\tau = S$

This same result is expressed somewhat more clearly as a correctness property:

$$\sigma_0 \{\!|\ S\ |\!\}_A \Box p \equiv \tau \{\!|\ \mathsf{Always}.p\ |\!\}_A \mathsf{false}$$

Lemma 2.4.1 shows that we can reduce the question of whether a temporal property can be enforced for a contract to the question of whether a certain goal can be achieved. In this case, the goal false cannot really be established, so success can only be achieved by miraculous termination, or by nontermination caused by an agent that does not belong to A.

The tester contract does not, in fact, constitute a fundamental extension to the notion of contracts; it can be modeled with existing contract constructs (although it requires an infinite choice construct, if the number of agents is infinite).

In a similar way, we can define a tester Eventually.p for the property $\Diamond p$, by

$$\mathsf{Eventually}.p = (\mu\ X \bullet [\neg p]\ ;\ \{s \neq \Lambda\}\ ;\ \mathit{step}\ ;\ X)$$

This tester is described in Fig. 2.3. We have the following result for this tester:

Lemma 2.4.2

$$\sigma_0 \{\!|\ S\ |\!\}_A \Diamond p \equiv \tau \{\!|\ \mathsf{Eventually}.p\ |\!\}_A \mathsf{false}$$

where $\sigma.\tau = \sigma_0$ *and* $s.\tau = S$.

Again, this shows how the question whether a temporal property can be enforced is reduced to a question about whether a goal can be achieved. In this case, the eventually-property does not hold if execution continues forever without ever encountering a state where p holds.

2.4.3 *Other temporal properties*

A more complicated behavior is illustrated by the *until* operator. We say that a property p holds until property q, denoted $p\,\mathcal{U}\,q$, if q will hold eventually, and until then p holds. We have that

$$\sigma_0 \{\!|\ S\ |\!\}_A\ p\,\mathcal{U}\,q \equiv \tau \{\!|\ \mathsf{Until}.p.q)\ |\!\}_A \mathsf{false}$$

where $\sigma.\tau = \sigma_0$ and $s.\tau = S$ and the tester for until is defined by

$$\mathsf{Until}.p.q = (\mu X \bullet [\neg q]\ ;\ \{p\}\ ;\ \{s \neq \Lambda\}\ ;\ \mathrm{step}\ ;\ X)$$

This tester is illustrated by the diagram in Fig. 2.4.

The *weak until*, denoted $p\,\mathcal{W}\,q$, can be defined in a similar matter. It differs from the previous operator in that it is also satisfied if q is never satisfied, provided p is always satisfied. We have that

$$\sigma_0 \{\!|\ S\ |\!\}\ p\mathcal{W}q \equiv \tau \{\!|\ \mathsf{Wuntil}.p.q\ |\!\}_A \mathsf{false}$$

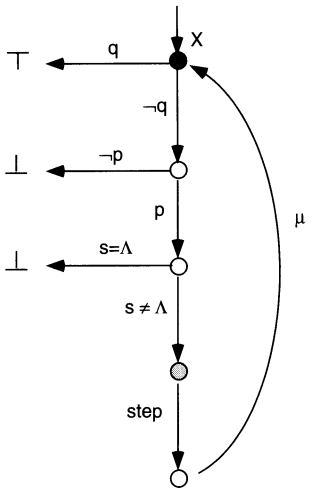


Figure 2.4. A property p holds until q

where $\sigma.\tau = \sigma_0$ and $s.\tau = S$ and

$$\mathsf{Wuntil}.p.q = (\nu X \bullet [\neg q]\ ;\ \{p\}\ ;\ \{s \neq \Lambda\}\ ;\ \mathrm{step}\ ;\ X)$$

The always and eventually operators arise as special cases of the until operators: $\Box p = p\ \mathcal{W}\ \mathsf{false}$ and $\Diamond p = \mathsf{true}\ \mathcal{U}\ p$.

Another interesting property is p *leads to* q, denoted $p \rightsquigarrow q$. We have that the tester for leads-to is

$$\mathsf{Leadsto}.p.q = [p]\ ;\ (\mu X \bullet [\neg q]\ ;\ \{s \neq \Lambda\}\ ;\ \mathrm{step}\ ;\ X)$$

and it holds if and only if

$$\sigma_0\ \{\!|\ S\ |\!\}_A\ p \rightsquigarrow q \equiv \tau\ \{\!|\ \mathsf{Leadsto}.p.q\ |\!\}_A\ \mathsf{false}$$

where $\sigma.\tau = \sigma_0$ and $s.\tau = S$. The behavior of this tester is illustrated by the diagram in Fig. 2.5.

An even more useful property is to say that property p *always leads to* property q, denoted by $\Box(p \rightsquigarrow q)$. This requires that we use two loops, a ν- loop and a μ-loop. We have that

$$\sigma_0\ \{\!|\ S\ |\!\}_A\ \Box(p \rightsquigarrow q) \equiv \tau\ \{\!|\ \mathsf{Aleadsto}.p.q\ |\!\}_A\ \mathsf{false}$$

where $\sigma.\tau = \sigma_0$ and $s.\tau = S$ and

$$\begin{aligned}\mathsf{Aleadsto}.p.q = (\nu Y \bullet [\neg p]\ ;\ [s \neq \Lambda]\ ;\ \mathrm{step}\ ;\ Y \\ \sqcap\ [p]\ ;\ (\mu X \bullet [\neg q]\ ;\ \{s \neq \Lambda\}\ ;\ \mathrm{step}\ ;\ X \sqcap [q])\ ;\ [s \neq \Lambda]\ ;\ \mathrm{step}\ ;\ Y)\end{aligned}$$

This is illustrated in Fig. 2.6.

The above mentioned behavioral properties all have one thing in common, they are *insensitive to finite stuttering*. This means that if a step of the computation does not change the state, then the effect is the same as if that step had been omitted.

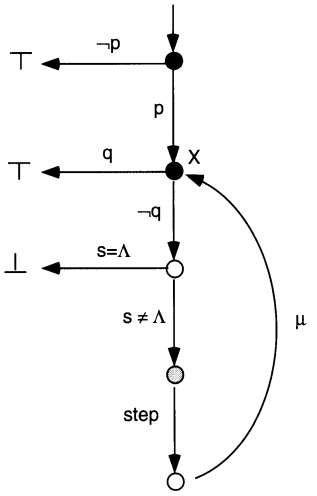


Figure 2.5. A property p leads to q

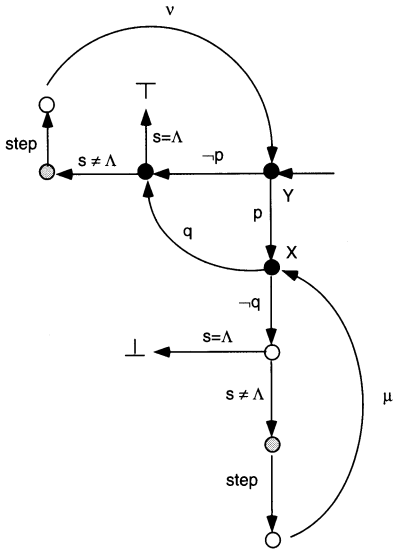


Figure 2.6. A property p always leads to q

In the diagram, this means that a stuttering step will lead back to the same state in the diagram. Being insensitive to stuttering means that the number of steps that are taken is not important for the behavioral property, only the sequence of properties that arise during the execution. Note that a computation should not be insensitive to infinite stuttering, as this is amounts to a form of nontermination (internal divergence).

2.5 Analyzing behavior of action systems

Let us now look at how to enforce temporal properties for action systems. As action systems are just contracts, we can use the techniques developed above for this. However, we will show that the simple format for action systems allows us to simplify the characterizations and proof obligations considerably.

2.5.1 *Classification of action systems*

In Section 2.2.5 we noted that an action system in general is of the form

$$\mathcal{A}(a,b,c) = B\,;\,(\mathsf{rec}_c\,X \bullet (S_1\,[]_b\,S_2\,[]_b\,\ldots\,[]_b\,S_n)\,;\,X\,[]_a\,\langle g\rangle_a)\,;\,E$$

Given a coalition A that is trying to achieve some goal, there are eight different possibilities to consider: whether $a,b \in A$, or $a \in A, b \notin A$, or $a \notin A, b \in A$, or $a \notin A, b \notin A$, and, in each case, whether $c \in A$ or $c \notin A$. We will briefly characterize the intuition behind each of these cases, assuming that agent a is the user (the environment) and that agent b is the computer (the system). In each case, $c \in A$ means that the computation must terminate eventually if the goal is to be achieved, while $c \notin A$ means that the goal can be achieved even if the iteration goes on forever.

- *Angelic iteration* ($a,b \in A$): At each step, the user decides whether to quit (which is possible if g holds) or whether to continue one more iteration. In the latter case, the user decides which alternative S_i to choose for the next iteration. Angelic iteration models an *event loop*, where the user can choose what action or event to execute next, and also may choose to exit the loop whenever this is allowed by the exit constraint.

- *Angelic iteration with demonic exit* ($a \notin A, b \in A$): This case is similar to the previous one, except that the choice whether to terminate or not is made by the computer and not by the user. In other words, it is like an event loop, where termination may happen at any time when termination is enabled, the choice of when to terminate being outside the control of the user.

- *Demonic iteration* ($a \notin A, b \notin A$): The computer decides whether to stop or to continue the iteration, and in the latter case, which of the alternative actions to continue with, in a way that cannot be controlled by the user. This form of iteration models a *concurrent system*, where the nondeterminism in the choice of the next iteration action expresses the arbitrary interleaving of enabled actions.

- *Demonic iteration with angelic exit* ($a \in A, b \notin A$): Here we have a similar situation as the previous, a concurrent system, where, however, termination is decided by the user. At each step, the user can decide whether to terminate or continue (provided the exit condition is satisfied), but the user cannot influence the choice of the next action, if she decides to continue.

We get more traditional systems as special cases of these very general forms of iteration. Dijkstra's guarded iteration statement is a special kind of demonic iteration, where some action is enabled if and only if the exit condition does not hold. A traditional temporal logic model is essentially a demonic iteration where the exit condition is always false, i.e., the system never terminates, and no abortions are permitted.

Our formalization introduces three main extensions to the traditional temporal logic model: the possibility that an execution may terminate, the possibility of angelic choice during the execution, and the possibility of a failed or miraculously successful execution.

Action systems can be used to model both interactive systems, where the choice of actions is under the control of the user, and concurrent systems, where the choice of actions is outside the control of the user. In fact, the action contract formalism is much more expressive than either one of these two formalisms, because the actions themselves may also be either angelic or demonic. In a concurrent as well as in an interactive system, we may have angelic choices made inside an action. This roughly corresponds to an input statement in the action. We can also have demonic choice inside an action, which roughly corresponds to a specification statement, where only partial information about the result is known.

2.5.2 *Analyzing behavior*

In the action system $\mathcal{A} = (\mathsf{rec}_c\ X \bullet S[]_a\langle g\rangle_a)$ we assume that the execution of S is atomic, in the sense that the state can not be observed inside the execution of S. Hence, to determine whether a property like $\Box p$ or $\Diamond p$ holds, we only observe the state at the beginning, immediately before each iteration, and at the end. This means that a state may violate the property p inside the execution of S, without violating the property $\Box p$ and it may satisfy the property p inside the execution of S, without satisfying the property $\Diamond p$. The justification for this is that we consider S as a *specification* of what kind of state change is taking place, rather than an actual implementation. If the internal working of S needs to be taken into account, then each internal step has to be modeled as a separate action.

Let us now consider how to determine whether an agent can enforce a temporal property like $\Box p$ during the execution of an action system. Action systems introduce a notion of atomicity that we have not modeled before, so we need to extend our operational semantics first.

We augment the syntax and operational semantics of contracts with a feature to indicates that a sequence of execution steps are internal, thus resulting in unobservable internal states. To do this, we introduce two additional statements into contracts: hide and unhide. We update the operational semantics for contracts by assuming that the state component is of the form (σ, o), where o is a boolean value, indicating whether the state is observable or not. This component is not changed by the previously introduced contract constructs. Thus, we have, e.g.,

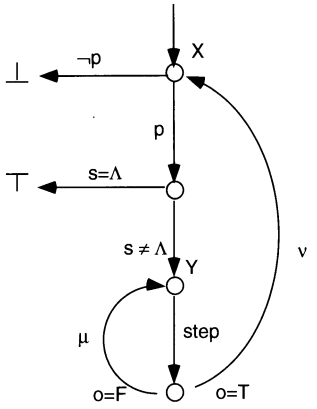


Figure 2.7. Modified tester for $\Box p$

that

$$\frac{p.\sigma}{(\text{if } p \text{ then } S_1 \text{ else } S_2 \text{ fi}, (\sigma, o)) \rightarrow (S_1, (\sigma, o))}$$

and similarly for the other contract constructs. For the hiding and unhiding operations, we introduce two new axioms:

$$\overline{(\mathsf{hide}, (\sigma, o)) \rightarrow (\Lambda, (\sigma, \mathsf{F}))} \qquad \overline{(\mathsf{unhide}, (\sigma, o)) \rightarrow (\Lambda, (\sigma, \mathsf{T}))}$$

The hide and unhide operations will thus just toggle the flag o, indicating whether the state is considered observable or not (we assume that there are no nested hidings). A contract statement whose internal computation is hidden is denoted $\langle S \rangle$, defined by

$$\langle S \rangle = \mathsf{hide}\,;\, S\,;\, \mathsf{unhide}$$

We assume as a syntactic restriction that there are no unmatched hide or unhide operations in a contract. We also do not allow nested hiding and unhiding in actions.

We also need to modify the tester, to take the hidden states into account. The modified tester for the property $\Box p$ is as follows:

$$\mathsf{Always}.p = (\nu X \bullet \{p\}\,;\, [s \neq \Lambda]\,;\, (\mu\, Y \bullet \mathit{step}\,;\, \mathsf{if}\ o\ \mathsf{then}\ X\ \ \mathsf{else}\ \ Y\ \mathsf{fi}))$$

The behavior of this interpreter is shown in Fig. 2.7. We have to take a position here on whether nonterminating unobservable computations (internal divergence) are good or bad. We choose here to consider them bad, although it might also be possible to argue for the opposite interpretation. Thus, internal divergence is equivalent to abortion (i.e., the designated angel breaching the contract), hence the μ label on the arrow in the inner loop.

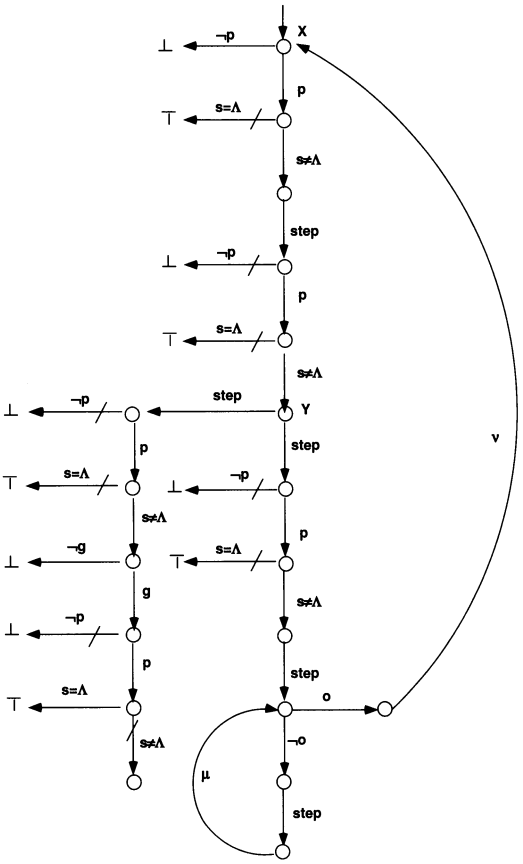


Figure 2.8. Action system tester for $\Box p$

Let us next show how we can compute the precondition for agents A to enforce the property $\Box p$ in the action system

$$\mathcal{A} = (\mathsf{rec}_c\, X \bullet \langle S \rangle \,;\, X[]_a \langle g \rangle_a)$$

in the case when $a \in A$. We have that

$$\sigma \{\!|\ \mathcal{A}\ |\!\}_A \Box p \equiv (\nu X \bullet \{p\} \,;\, [\neg g] \,;\, \mathsf{wp}.S.A \,;\, X).\mathsf{false}.\sigma$$

We can show that this is indeed the case, by considering how the coalition A would execute the contract Always.p from initial state $(\mathcal{A}, (\sigma, T))$. This is done by unfolding the iteration appropriately, as shown in Fig. 2.8. We have crossed out all those branches in the figure that cannot be taken, because the condition is known to be false. By eliminating these branches, as well as branches that the coalition would avoid because they would lead to certain failure, we derive the simpler diagram shown in Fig. 2.9. This proves that our characterization of the always tester for the action system $\mathcal{A}$ given above is correct.

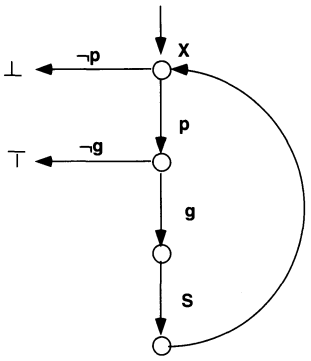


Figure 2.9. Simple action system tester for $\Box p$

The above gave the basic result that we need in order to reason about the temporal behavior of action systems. The main advantage here is that we can argue directly about the weakest preconditions of the actions, without having to go the indirect route of an interpreter for the system.

2.6 Verifying enforcement

We have shown above how to characterize enforcement of temporal properties for action systems using weakest preconditions. In this section, we will look in more detail at how one should prove enforcement in practice.

2.6.1 Predicate-level conditions for correctness

When reasoning about systems in practice, we want to talk about enforcement (correctness) with respect to a precondition rather than a specific initial state. The obvious generalization is

$$p \{\!|\ \mathcal{A}\ |\!\}_A \Box p \equiv (\forall \sigma \in p \bullet \sigma \{\!|\ \mathcal{A}\ |\!\}_A \Box p)$$

and similarly for other temporal properties. Furthermore, it is more practical to reason about fixpoints on the predicate level rather than on the predicate transformer level. A straightforward argument shows that from the rule shown in Section 2.5.2 we get the following rule:

$$p \{\!|\ \mathcal{A}\ |\!\}_A \Box q \equiv p \subseteq (\nu x \bullet q \cap (g \cup \mathsf{wp}.S.A.x))$$

for the case when $a \in A$. In a similar way, we can derive predicate-level characterizations of correctness for the different temporal properties and for the different cases of which agents belong to the coalition that we are interested in. The following lemmas collect those cases that we will use in the examples in Section 2.6.4.

Lemma 2.6.1 *Assume that action system* $\mathcal{A} = (\mathsf{rec}_c X \bullet \langle S \rangle \ ; \ X[]_a \langle g \rangle_a)$ *and coalition A are given. Then*

$$p \{\!| \ \mathcal{A} \ |\!\}_A \Box q \equiv \begin{cases} p \subseteq (\nu x \bullet q \cap (g \cup \mathsf{wp}.S.A.x)) & \text{if } a \in A \\ p \subseteq (\nu x \bullet q \cap \mathsf{wp}.S.A.x) & \text{if } a \notin A \end{cases}$$

$$p \{\!| \ \mathcal{A} \ |\!\}_A \Diamond q \equiv \begin{cases} p \subseteq (\mu x \bullet q \cup \mathsf{wp}.S.A.x) & \text{if } a \in A \\ p \subseteq (\mu x \bullet q \cup (\neg g \cap \mathsf{wp}.S.A.x)) & \text{if } a \notin A \end{cases}$$

$$p \{\!| \ \mathcal{A} \ |\!\}_A \, q \, \mathcal{U} \, r \equiv \begin{cases} p \subseteq (\mu x \bullet r \cup (q \cap \mathsf{wp}.S.A.x)) & \text{if } a \in A \\ p \subseteq (\mu x \bullet r \cup (\neg g \cap q \cap \mathsf{wp}.S.A.x)) & \text{if } a \notin A \end{cases}$$

In Section 2.3 we considered achieving goals (postconditions) with contracts. Since our generalized notion of temporal properties also includes finite scenarios (aborting, termination or miraculous), we can consider achieving a postcondition q as enforcing a special temporal property $\triangle q$ (*finally* q).

We can also give the characterization for a finally-property in the same way as for other temporal properties. For simplicity we assume that $c \notin A$. We have that

Lemma 2.6.2 *Assume that action system* $\mathcal{A} = (\mathsf{rec}_c X \bullet \langle S \rangle \ ; \ X[]_a \langle g \rangle_a)$ *and coalition A are given. If* $c \in A$*, then*

$$p \{\!| \ \mathcal{A} \ |\!\}_A \triangle q \equiv \begin{cases} p \subseteq (\mu x \bullet (g \cap q) \cup \mathsf{wp}.S.A.x) & \text{if } a \in A \\ p \subseteq (\mu x \bullet (\mathsf{wp}.S.A.\mathsf{false} \cap q) \cup (\neg g \cap \mathsf{wp}.S.A.x)) & \text{if } a \notin A \end{cases}$$

On the other hand, if $c \notin A$*, then*

$$p \{\!| \ \mathcal{A} \ |\!\}_A \triangle q \equiv \begin{cases} p \subseteq (\nu x \bullet (g \cap q) \cup \mathsf{wp}.S.A.x) & \text{if } a \in A \\ p \subseteq (\nu x \bullet (\mathsf{wp}.S.A.\mathsf{false} \cap q) \cup (\neg g \cap \mathsf{wp}.S.A.x)) & \text{if } a \notin A \end{cases}$$

Here the intuition is that if $a \in A$, then the termination is (angelically) chosen whenever g and q both hold. If $a \notin A$, then continuation is (demonically) chosen whenever q holds, if possible (i.e., if $\neg \mathsf{wp}.S.A.\mathsf{false}$ holds).

Note that this is the first temporal property where the agent c comes into play. In our generalizations of classical temporal operators, the notion of who is responsible for infinite executions does not matter. However, it does matter (as was the original intention) when considering establishing a postcondition.

The conditions for the case $a \notin A$ in Lemma 2.6.2 contain the odd-looking predicate $\mathsf{wp}.S.A.\mathsf{false}$, but in the case when termination is deterministic we get a simplification,because then $\mathsf{wp}.S.A.\mathsf{false} = g$).

2.6.2 *Invariant-based methods*

Lemma 2.6.1 shows how proving enforcement of temporal properties is reduced to proving properties of the form $p \subseteq e$ where e is a μ- or ν-expression. From fixpoint theory we know that such properties can be proved using an invariant (ν) or an invariant and a termination function (μ):

Lemma 2.6.3 *Assume that action system* $\mathcal{A} = (\mathsf{rec}_c X \bullet \langle S \rangle \ ; \ X[]_a \langle g \rangle_a$ *is given, together with a coalition A. Then*

(a) *Always-properties can be proved using invariants, as follows:*

$$\frac{p \subseteq I \qquad \neg g \cap I \,\{\!|\, S \,|\!\}_A\, I \qquad I \subseteq q}{p \,\{\!|\, \mathcal{A} \,|\!\}_A\, \Box q}\; a \in A$$

$$\frac{p \subseteq I \qquad I \,\{\!|\, S \,|\!\}_A\, I \qquad I \subseteq q}{p \,\{\!|\, \mathcal{A} \,|\!\}_A\, \Box q}\; a \notin A$$

(b) *Eventually-properties can be proved using invariants and termination arguments, as follows:*

$$\frac{p \subseteq q \cup I \qquad \neg q \cap I \cap t = w \,\{\!|\, S \,|\!\}_A\, q \cup (I \cap t < w)}{p \,\{\!|\, \mathcal{A} \,|\!\}_A\, \Diamond q}\; a \in A$$

$$\frac{\begin{array}{c} p \subseteq q \cup (\neg g \cap I) \\ \neg q \cap \neg g \cap I \cap t = w \,\{\!|\, S \,|\!\}_A\, q \cup (\neg g \cap I \cap t < w) \end{array}}{p \,\{\!|\, \mathcal{A} \,|\!\}_A\, \Diamond q}\; a \notin A$$

where the state function t ranges over some well-founded set.

(c) *Until-properties can be proved as follows:*

$$\frac{p \subseteq r \cup I \qquad \neg r \cap I \cap t = w \,\{\!|\, S \,|\!\}_A\, r \cup (I \cap t < w) \qquad I \subseteq q}{p \,\{\!|\, \mathcal{A} \,|\!\}_A\, q \,\mathcal{U}\, r}\; a \in A$$

$$\frac{\begin{array}{c} p \subseteq r \cup (\neg g \cap I) \\ \neg g \cap \neg r \cap I \cap t = w \,\{\!|\, S \,|\!\}_A\, r \cup (\neg g \cap I \cap t < w) \\ I \subseteq q \end{array}}{p \,\{\!|\, \mathcal{A} \,|\!\}_A\, q \,\mathcal{U}\, r}\; a \notin A$$

where the state function t again ranges over some well-founded set.

Enforcing finally-properties is essentially proving correctness for loops, with different combinations of angelic and demonic nondeterminism:

Lemma 2.6.4 *Finally-properties can be proved as follows, in the case when* $c \in A$*:*

$$\frac{p \subseteq (g \cap q) \cup I \qquad \neg(g \cap q) \cap I \cap t = w \,\{\!|\, S \,|\!\}_A\, (g \cap q) \cup (I \cap t < w)}{p \,\{\!|\, \mathcal{A} \,|\!\}_A\, \triangle q}\; a \in A$$

$$\frac{p \subseteq I \qquad \neg \mathsf{wp}.S.A.\,\mathsf{false} \cap I \cap t = w \,\{\!|\, S \,|\!\}_A\, I \cap t < w \qquad g \cap I \subseteq q}{p \,\{\!|\, \mathcal{A} \,|\!\}_A\, \triangle q}\; a \notin A$$

where the state function t again ranges over some well-founded set. In the case when $c \notin A$ *the rules are similar, but without termination function t.*

2.6.3 *Demonstrative methods*

In some cases, temporal properties can be proved by demonstrating a specific sequence of correctness steps. This idea will be used in examples in Section 2.6.4. We now show how such methods can be derived from the general characterizations of temporal properties.

An eventually-property $\Diamond q$ can be proved by showing that a specific number of steps will lead to the condition q holding:

Lemma 2.6.5 *Assume that action system $\mathcal{A}$ is given as before. Then*

$$\frac{p \{\!|\ S^n\ |\!\}_A\ q}{p \{\!|\ \mathcal{A}\ |\!\}_A \Diamond q}\ a \in A \qquad \frac{p \{\!|\ (\{\neg g\}\,;S)^n\ |\!\}_A\ q}{p \{\!|\ \mathcal{A}\ |\!\}_A \Diamond q}\ a \notin A$$

where S^n means n-fold sequential composition $S\,;S\,;\cdots;S$ (and $S^0 = \mathsf{skip}$).

Proof. We prove that this rule is valid, for the case $a \in A$. Assume $p \{\!|\ S^n\ |\!\}_A\ q$. We first note that it is straightforward to show (by induction on n) that $T^n.q \subseteq (\mu x \bullet q \cup T.x)$, for arbitrary predicate transformer T and predicate q. Thus, we have

$$\begin{array}{ll} & p \{\!|\ \mathcal{A}\ |\!\}_A \Diamond q \\ \equiv & \{\text{original rule for correctness}\} \\ & p \subseteq (\mu x \bullet q \cup \mathsf{wp}.S.A.x) \\ \Leftarrow & \{\text{comment above}\} \\ & p \subseteq (\mathsf{wp}.S.A)^n.q \\ \equiv & \{\text{homomorphism (Section 2.3.1)}\} \\ & p \subseteq (\mathsf{wp}.S^n.A).q \\ \equiv & \{\text{definition of correctness}\} \\ & p \{\!|\ S^n\ |\!\}_A\ q \end{array}$$

□

Until-properties can be proved by exhibiting a suitable correctness sequence.

Lemma 2.6.6 *Assume that action system $\mathcal{A}$ is given as before. Then*

$$\frac{q_i \subseteq q\ (i = 0..n-1) \qquad q_i \{\!|\ S\ |\!\}_A\ q_{i+1}\ (i = 0..n-1)}{p \{\!|\ \mathcal{A}\ |\!\}_A\ q\,\mathcal{U}\,r}\ a \in A$$

and

$$\frac{q_i \subseteq q\ (i = 0..n-1) \qquad q_i \{\!|\ \{\neg g\}\,;S\ |\!\}_A\ q_{i+1}\ (i = 0..n-1)}{p \{\!|\ \mathcal{A}\ |\!\}_A\ q\,\mathcal{U}\,r}\ a \notin A$$

where $p = q_0$ and $q_n = r$.

Proof. We prove that this rule is valid, for the case $a \in A$. Assume $q_i \subseteq q$ and $q_i \{\!|\ S\ |\!\}_A\ q_{i+1}$ for $i = 0..n-1$. We show that $q_i \subseteq (\mu x \bullet r \cup (q \cap \mathsf{wp}.S.A.x))$ for

$i = 0..n$, by induction down from n. As a base case we have (where $T = \mathsf{wp}.S.A$)

$$\begin{array}{ll} & q_n \\ \subseteq & \{\text{assumptions}\} \\ & q \cap T.r \\ \subseteq & \{\text{monotonicity}\} \\ & r \cup (q \cap T.(\mu x \bullet r \cup (q \cap T.x))) \\ = & \{\text{fold fixpoint}\} \\ & (\mu x \bullet r \cup (q \cap T.x))) \end{array}$$

and the step case is (for $0 < i \leq n$)

$$\begin{array}{ll} & q_{i-1} \\ \subseteq & \{\text{assumptions}\} \\ & q \cap T.q_i \\ \subseteq & \{\text{monotonicity, induction assumption}\} \\ & r \cup (q \cap T.(\mu x \bullet r \cup (q \cap T.x))) \\ = & \{\text{fold fixpoint}\} \\ & (\mu x \bullet r \cup (q \cap T.x)) \end{array}$$

which by induction gives us $q_0 \subseteq (\mu x \bullet r \cup (q \cap T.x))$ from which $p \{\!|\ \mathcal{A}\ |\!\}_A\ q\ \mathcal{U}\ r$ follows by Lemma 2.6.1, since $p = q_0$. □

2.6.4 *Enforcement in example systems*

Let us now apply these techniques to analyzing enforcement of temporal properties in the three example action systems that we described earlier.

The game of Nim

The game of Nim is described as the following action system:

$$\begin{array}{lll} \mathit{Nim} & = & (f := \mathsf{T}\ []_c\ f := \mathsf{F})\ ;\ \mathit{Play} \\ \mathit{Play} & =_z & f \wedge x > 0 \rightarrow \langle x := x' \mid x' < x \leq x' + 2 \rangle_a\ ;f := \neg f\ ;\ \mathit{Play} \\ & & []_c\ \neg f \wedge x > 0 \rightarrow \langle x := x' \mid x' < x \leq x' + 2 \rangle_b\ ;f := \neg f\ ;\ \mathit{Play} \\ & & []_c\ x = 0 \rightarrow \mathsf{skip} \end{array}$$

Before considering questions about winning or losing, we consider the general question "Will the game always terminate"? In order to answer this question in the most general way, we take the point of view of an empty coalition (so all agents are demonic, i.e., trying to prevent termination). The property that we want to enforce is $\triangle\mathsf{true}$. We use Lemma 2.6.3 (d), according to which answering this question is equivalent to proving termination of the traditional loop program

$$\mathsf{while}\ x > 0\ \mathsf{do}\ \mathsf{if}\ f\ \mathsf{then}\ [x := x' \mid x' < x \leq x' + 2]\ ;\ \langle f := \neg f \rangle$$

$$\text{else } [x := x' \mid x' < x \leq x' + 2] \,;\, \langle f := \neg f \rangle$$

fi

od

This is straightforward, with invariant true and termination argument x. Since termination is guaranteed regardless of whether the agent z (who is responsible for termination) is part of the coalition we consider or not, we can considered z to be a dummy and leave it out of the discussion when analyzing other properties.

The most obvious question that we can ask about this system is "Under what initial conditions can agent a (or b) win the game"? The desired postcondition from the point of view of agent a is $f \wedge x = 0$, while from the point of view of agent b it is $\neg f \wedge x = 0$.

We first show that in *Play*, agent a can win under the precondition

$$p = (f \wedge x \bmod 3 \neq 1) \vee (\neg f \wedge x \bmod 3 = 1)$$

regardless of how the scheduler works. We use Lemma 2.6.3 (d) again, with coalition $A = \{a\}$ and with invariant $(f \wedge x \bmod 3 \neq 1) \vee (\neg f \wedge x \bmod 3 = 1)$, i.e., the same as the precondition. The idea is that a can always make the state change from a situation where $x \bmod 3 \neq 1$ to a situation where $x \bmod 3 = 1$ while b must then re-establish $x \bmod 3 \neq 1$. The result is the same in the case $A = \{a, c\}$, since the scheduler is essentially deterministic inside *Play*.

Now it is easy to show that the initialization always establishes the precondition p if the scheduler is angelic ($c \in A$) but never if the scheduler is demonic ($c \notin A$). The conclusion of this is that in the original game, we can always win if we are allowed to decide who should start (after we know how many sticks are in the pile).

Wolf, goat and cabbage

The action system that describes the wolf, goat and cabbage problem is as follows:

$$\begin{aligned}
\mathit{CrossRiver} &= m, w, g, c := \mathsf{F}, \mathsf{F}, \mathsf{F}, \mathsf{F} \,;\, \mathit{Transport} \\
\mathit{Transport} &=_a \quad m = w \rightarrow m, w := \neg m, \neg w \,;\, \mathit{Transport} \\
&\quad []_a\, m = g \rightarrow m, g := \neg m, \neg g \,;\, \mathit{Transport} \\
&\quad []_a\, m = c \rightarrow m, c := \neg m, \neg c \,;\, \mathit{Transport} \\
&\quad []_a\, m := \neg m \,;\, \mathit{Transport} \\
&\quad []_a\, \mathsf{skip}
\end{aligned}$$

We want to prove that the agent (the man) can enforce the following temporal property using the contract:

$$(w = g \Rightarrow m = w) \wedge (g = c \Rightarrow m = g) \;\; \mathcal{U} \;\; m \wedge w \wedge g \wedge c$$

The simplest way to show this is to verify the following sequence of correctness steps:

$$\mathsf{true}$$

$$
\begin{array}{c}
\{\!|\ m, w, g, c := \mathsf{F}, \mathsf{F}, \mathsf{F}, \mathsf{F}\ |\!\} \\
\neg m \wedge \neg w \wedge \neg g \wedge \neg c \\
\{\!|\ S\ |\!\} \\
m \wedge \neg w \wedge g \wedge \neg c \\
\{\!|\ S\ |\!\} \\
\neg m \wedge \neg w \wedge g \wedge \neg c \\
\{\!|\ S\ |\!\} \\
m \wedge w \wedge g \wedge \neg c \\
\{\!|\ S\ |\!\} \\
\neg m \wedge w \wedge \neg g \wedge \neg c \\
\{\!|\ S\ |\!\} \\
m \wedge w \wedge \neg g \wedge c \\
\{\!|\ S\ |\!\} \\
\neg m \wedge w \wedge \neg g \wedge c \\
\{\!|\ S\ |\!\} \\
m \wedge w \wedge g \wedge c
\end{array}
$$

where S stands for the action of the system, i.e.,

$$
\begin{array}{l}
\quad \{m = w\}\ ;\ m, w := \neg m, \neg w \\
\sqcup\ \{m = g\}\ ;\ m, g := \neg m, \neg g \\
\sqcup\ \{m = c\}\ ;\ m, c := \neg m, \neg c \\
\sqcup\ m := \neg m
\end{array}
$$

By Lemma 2.6.5 this is sufficient, since each of the intermediate conditions implies $(w = g \Rightarrow m = w) \wedge (g = c \Rightarrow m = g)$.

The Dim Sun restaurant

The Dim Sun restaurant was described as follows:

$$
\begin{array}{lll}
\mathit{Dim\ Sun} = x_0, x_1, x_2, f := 0, 0, 0, 3\ ;\ \mathit{Serve} & & \\
\mathit{Serve} =_z & \langle r > 0 \rangle_e\ ;\ (\ \langle f \neq 0 \rangle_d\ ;\ (\langle x_0, r := x_0 + 1, r - 1 \rangle [\,]_a \mathsf{skip})\ ;\ \langle f := 0 \rangle\ ;\ \mathit{Serve} & \\
& \quad [\,]_d\ \langle f \neq 1 \rangle_d\ ;\ (\langle x_1, r := x_1 + 1, r - 1 \rangle [\,]_b \mathsf{skip})\ ;\ \langle f := 1 \rangle\ ;\ \mathit{Serve} & \\
& \quad [\,]_d\ \langle f \neq 2 \rangle_d\ ;\ (\langle x_2, r := x_2 + 1, r - 1 \rangle [\,]_c \mathsf{skip})\ ;\ \langle f := 2 \rangle\ ;\ \mathit{Serve}\ ) & \\
& [\,]_e\ \mathsf{skip} &
\end{array}
$$

With this setup we can prove that with the help of the servant, a customer can get at least half of the items that have been taken:

$$x_0 = 0 \wedge x_1 = 0 \wedge x_2 = 0 \wedge f = 3\ \{\!|\ \mathcal{A}\ |\!\}_{\{a,d\}}\ \Box(x_0 \geq x_1 + x_2)$$

The proof uses Lemma 2.6.3 with invariant $(f = 0 \wedge x_0 > x_1 + x_2) \vee (f \neq 0 \wedge x_0 \geq x_1 + x_2)$.

Similarly, we can prove that two cooperating customers can get almost half of the items, provided that the manager helps by not closing too early:

$$x_0 = 0 \wedge x_1 = 0 \wedge x_2 = 0 \wedge f = 3 \; \{\!|\; \mathcal{A} \;|\!\}_{\{a,b,e\}} \; \triangle(r = 0 \wedge x_0 + x_1 \geq x_2 - 1)$$

In this case, the invariant is $x_0 + x_1 \geq x_2 - 1$ and the termination argument is, e.g., $3r + 3 - f$.

2.7 Conclusions and related work

The main purpose of this paper has been to show how to model enforcement of temporal properties during execution of contracts. Our results generalize the results that we have presented earlier, in particular in [6], where only achievement of specific goals with a contract was considered. At the same time, the results provide a generalization of the standard temporal logic analysis framework, where only one kind of nondeterminism is allowed (demonic). Another contribution here is the generalization of action systems. The traditional notion of action systems only allows demonic choice, and is mainly used to model concurrent systems. In [6] and [10], we generalized this to action systems with angelic choice, to model interactive systems. Here we carry this one step further, and give a general contract model for action systems that allows any number of agents to participate in the execution of the action system, getting concurrent and interactive systems as special cases. A noteworthy feature of this generalization is that now also termination in action systems is nondeterministic. Action systems turn out to be quite good for describing systems, as they allow simplified characterizations and proof obligations for enforcement of temporal properties. At the same time, action systems introduce a notion of atomicity that is not directly modeled in contracts. The examples have been chosen to illustrate the new kinds of applications that now can be handled in our approach. More traditional examples of concurrent and interactive systems have been described elsewhere.

Temporal properties have been defined and used before in a predicate transformer framework with only demonic nondeterminism in order to generalize traditional reasoning systems for sequential and parallel programs [19, 13, 15]. In our more general predicate transformer framework (with angelic nondeterminism) verification of temporal properties of contracts is reduced to traditional correctness properties of special fixpoint contracts. These fixpoint contracts are built much in the same way as corresponding specifications of temporal properties using μ/ν-calculus, as is common, e.g., in connection with model checking [12]. However, in our framework these correctness properties can be verified using traditional invariant methods, with rules similar to those in traditional temporal reasoning systems [16]. Our generalization to include agents, coalitions, and angelic nondeterminism is similar to independent recent work by Alur, Henzinger, and Kupferman [2] on *alternating-time temporal logic*. They have a more elaborate model of games and interaction, but their view of computations is more traditional, without abortion, termination, or miracles.

References

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In G. A. et al., editor, *Proc. 16th ICALP*, pages 1–17, Stresa, Italy, 1989. Springer-Verlag.

[2] R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th Symposium on Foundations of Computer Science (FOCS)*, pages 100–109. IEEE, 1997.

[3] R .J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam, 1980.

[4] R. J. R. Back and J. von Wright. Duality in specification languages: a lattice-theoretical approach. *Acta Informatica*, 27:583–625, 1990.

[5] R. J. R. Back and J. von Wright. Games and winning strategies. *Information Processing Letters*, 53(3):165–172, February 1995.

[6] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[7] R. J. R. Back and J. von Wright. Contracts, games and refinement. *Information and Computation*, 156:25–45, 2000.

[8] R: J. R. Back and J. von Wright. Contracts as mathematical entities in programming logic. In *Proc. Workshop on Abstraction and Refinement*, Osaka, September 1999. Also available as TUCS Tech. Rpt. 372.

[9] R. J. R. Back and J. von Wright. Verification and refinement of action contracts. Tech. Rpt. 374, Turku Centre for Computer Science, November 2000.

[10] R. J. R. Back, A. Mikhajlova, and J. von Wright. Reasoning about interactive systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of the World Congress on Formal Methods (FM'99)*, volume 1709 of *LNCS*, pages 1460–1476. Springer-Verlag, September 1999.

[11] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.

[12] E. Emerson. Automated temporal reasoning about reactive systems. In F. Moller and G. Bortwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*. Springer–Verlag, 1996.

[13] W. Hesselink. *Programs, Recursion and Unbounded Choice*. Cambridge University Press, 1992.

[14] E. Knapp. A predicate transformer for progress. *Information Processing Letters*, 33:323–330, 1989.

[15] J. Lukkien. Operational semantics and generalised weakest preconditions. *Science of Computer Programming*, 22:137–155, 1994.

[16] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83:97–130, 1991.

[17] C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, January 1988.

[18] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[19] J. Morris. Temporal predicate transformers and fair termination. *Acta Informatica*, 27:287–313, 1990.

[20] Y. Moschovakis. A model of concurrency with fair merge and full recursion. *Information and Computation*, pages 114–171, 1991.

[21] W. Thomas. On the synthesis of strategies in infinite games. In *12th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 900 of *Lecture Notes in Computer Science*, pages 1–13, 1995.

3

Asynchronous progress

Ernie Cohen

Abstract

We propose weakening the definition of progress to a branching-time operator, making it more amenable to compositional proof and simplifying the predicates needed to reason about highly asynchronous programs. The new progress operator ("achieves") coincides with the "leads-to" operator on all "observable" progress properties (those where the target predicate is stable) and satisfies the same composition properties as leads-to, including the PSP theorem. The advantage of achievement lies in its compositionality: a program inherits all achievement properties of its "decoupled components". (For example, a dataflow network inherits achievement properties from each of its processes.) The compositionality of achievement captures, in a UNITY-like logic, the well-known operational trick of reasoning about an asynchronous program by considering only certain well-behaved executions.

3.1 Introduction

It is well known that progress properties (such as leads-to [1]) are not preserved under parallel composition. That is, it is not generally possible to obtain a useful progress property of a system from a progress property of one of its components. Instead, it is usually necessary to work globally with atomic progress steps (obtained by combining local liveness properties with global safety properties). This often results in unreasonably complicated proofs.

For example, consider the following trivial producer-consumer system. The producer repeatedly chooses a function, applies it to his local value, and sends the function along a FIFO channel to the consumer; the consumer, on receiving a function, applies it to his local variable. Initially, the two variables are equal and the channel is empty; we call such a state *clean*. We would like to prove that, if the producer eventually stops, the system terminates in a clean state.

Assertional proofs of this program are rather painful; they generally require either the introduction of an auxiliary inductive definition that captures the behavior

A. McIver et al. (eds.), *Programming Methodology*

of one of the processes (e.g., in order to formulate an invariant like "the system is in a state reachable by running the producer from a clean state") or introducing history variables on the channel (which amounts to reasoning about a static execution object instead of a dynamic program). In either case, the programming logic fails to provide substantial reasoning leverage.

A more attractive, though informal, operational argument might go as follows: starting from a clean state, the producer is guaranteed to execute first. At this point, the producer cannot interfere with the consumer's first step, so we can pretend that the consumer executes next, bringing the system back to a clean state. This is repeated for every message sent by the producer, so when the system halts, the state is again clean.

A number of theorems try to systematize this sort of reasoning, typically using commutativity to turn arbitrary executions into well-behaved ones. However, theorems that yield linear-time properties have to talk about states actually arising in a computation, making them difficult to compose. For example, applying such a theorem to the producer-consumer example would yield only properties of the initial and final states (since they are the only ones guaranteed to be clean); we would prefer a program property capturing the effect of production and subsequent consumption of a single message.

We propose a new progress operator $\rightsquigarrow$ ("achieves"), that supports this kind of reasoning within the UNITY programming logic. Achievement has a number of attractive features:

- It supports the key reasoning rules of the UNITY leads-to operator; in particular, it is transitive, disjunctive, and satisfies the PSP theorem.

- It coincides with leads-to on those properties that are "observable" (i.e., those whose target predicates are stable). Thus, it is as expressive as leads-to for all practical purposes.

- Unlike leads-to, it supports a form of compositional reasoning: a program inherits achievement properties from each of its "decoupled components" (e.g., the processes of a dataflow network). Decoupling can itself be established compositionally, usually through simple structural analysis.

- Most techniques for reasoning about concurrency control (such as reduction[10, 4] or serializability [6]) are based on pretending that certain operations execute "atomically". Decoupled components, on the other hand, effectively execute "immediately". This makes them easier to compose and allows them to serve as asynchronous maintainers as invariants (section 3.6).

- Achievement and decoupling are defined semantically (i.e., in terms of the properties of a program, not its transitions). They are thus independent of program presentation, unlike related theories like communication-closed layers or stubborn sets [15], which are defined at the level of transitions.

- Unlike techniques for compositional temporal logic reasoning, our theory allows multiple processes to write to the same variable. This allows us to reason about FIFO channels without having to resort to history variables.

- Unlike interleaving set temporal logic [9], which requires reasoning about entire executions, achievement obtains the same effect using simpler UNITY-like program reasoning.

In this paper, we show the key concepts and theorems, and some simple examples. Proofs of all results can be found in [2], which also contains a number of examples, including the sliding window protocol and the tree protocol for database concurrency control.

3.2 Programs

Our programs are countable, unconditionally fair, nondeterministic transition systems. As a starting point, we describe them using operators from UNITY [1].

Let S be a fixed set of states. As usual, we describe subsets of S using state predicates (notation: p, q, r, s), and identify elements of S with their characteristic predicates. An *action* (notation: f, g) is a binary relation on S; we identify actions with (universally disjunctive) predicate transformers giving their strongest postconditions. We will make use of the following actions (given in order of decreasing binding power):

$$
\begin{aligned}
1.p &= p \\
0.p &= \mathit{false} \\
(f; g).p &= g.(f.p) \\
(\wedge q).p &= p \wedge q \\
(f \vee g).p &= f.p \vee g.p \\
(f \Rightarrow g).p &= f.p \Rightarrow g.p \\
(\exists x).p &= (\exists x : p) \\
x := e &= (\exists x'); (\wedge(x' = e)); (\exists x); (\wedge(x = x')); (\exists x') \\
&\qquad \text{where } x' \text{ is a fresh variable} \\
(f \textbf{ if } p) &= ((\wedge p); f \vee (\wedge \neg p))
\end{aligned}
$$

The everywhere operator of [5] is extended to predicate transformers by

$$[h] \equiv [(\forall p : h.p)]$$

A *program* (notation: $A, B, \ldots$) is a countable set of actions. A program is executed by repeatedly stuttering or executing one of its actions, subject to the restriction that each action is chosen infinitely often; formally, an execution e is

an infinite sequence of states e_i such that

$$(\forall i \geq 0 : (\exists a \in A|\{1\} : [e_{i+1} \Rightarrow a.e_i]))$$
$$\wedge\ (\forall i \geq 0, a \in A : (\exists j \geq i : [e_{j+1} \Rightarrow a.e_j]))$$

Under this semantics, union of programs corresponds to fair parallel composition, so we use the symbol $|$ as a synonym for set union when composing programs.

The motivating problem of this paper is the desire to prove properties of the form $(p \mapsto q \textbf{ in } A)$ ("p leads-to q in A"), which says that every execution of A that starts with a p-state contains a q-state:

$$(p \mapsto q \textbf{ in } A) \equiv (\forall e : e \text{ an execution of } A : [e_0 \Rightarrow p] \Rightarrow (\exists i : [e_i \Rightarrow q]))$$

The standard way to prove $\mapsto$ properties is with the operators **U** and **E**, defined by

$$(p \textbf{ U } q \textbf{ in } A) \equiv (\forall a \in A : (\wedge(p \wedge \neg q)); a; (\wedge(\neg p \wedge \neg q)) = 0)$$
$$(p \textbf{ E } q \textbf{ in } A) \equiv (p \textbf{ U } q \textbf{ in } A) \wedge (\exists a \in A : (\wedge(p \wedge \neg q)); a; (\wedge \neg q) = 0)$$

$(p \textbf{ U } q \textbf{ in } A)$ ("p unless q in A") means that no A transition falsifies p without truthifying q (unless q is true already); this also means that in any execution of A, p, once true, remains true up to the first moment (if any) when q holds. $(p \textbf{ E } q \textbf{ in } A)$ ("p ensures q in A") means that, in addition, some transition of A is guaranteed to yield a q state when executed from a $p \wedge \neg q$ state. **U** properties are are used to specify safety, while **E** properties specify "atomic" progress.

Given **U** and **E**, we have the following (complete set of) rules for deriving $\mapsto$ properties:

$$\begin{aligned} (p \textbf{ E } q) &\Rightarrow (p \mapsto q) \\ (p \mapsto q) \wedge (q \mapsto r) &\Rightarrow (p \mapsto r) \\ (\forall i : p_i \mapsto q_i) &\Rightarrow (\exists i : p_i) \mapsto (\exists i : q_i) \\ (p \mapsto q) \wedge (r \textbf{ U } s) &\Rightarrow ((p \wedge r) \mapsto (q \wedge r) \vee s) \end{aligned} \tag{3.1}$$

the last rule known in UNITY lingo as "progress-safety-progress" (PSP).

The main reason for introducing **E** (instead of just working with $\mapsto$ and **U**) is that, unlike $\mapsto$, **E** properties can be composed, using the union rule:

$$(p \textbf{ E } q \textbf{ in } A) \wedge (p \textbf{ U } q \textbf{ in } B) \Rightarrow (p \textbf{ E } q \textbf{ in } A|B)$$

The main purpose of this paper is to define a replacement for $\mapsto$ that has a similar union rule (under suitable semantic constraints on A and B), while preserving the composition rules of (3.1).

It is not hard to see that, for the purpose of showing progress properties of programs under union, the **U** and **E** properties of a program are a fully abstract semantics. Therefore, we define $\sim$ (congruence) and $\triangleleft$ (containment) of programs in the obvious way (**op** : ranges over $\{\textbf{U}, \textbf{E}\}$):

$$A \sim B \equiv (\forall p, q, \textbf{op} : (p \textbf{ op } q \textbf{ in } A) \Leftrightarrow (p \textbf{ op } q \textbf{ in } B))$$
$$A \triangleleft B \equiv (A|B \sim B)$$

An example of the advantage of using semantic notions (as opposed to equality and subset) can be seen when we extend guarding to programs

$$(A \textbf{ if } p) \equiv \{(a \textbf{ if } p) \text{ s.t. } a \in A\}$$

; we then have $(A \textbf{ if } p) \triangleleft A$.

We will make frequent use of the following two properties derived from **U**:

$$(p \textbf{ stable in } A) \equiv (p \textbf{ U } \mathit{false} \textbf{ in } A)$$
$$\langle A \rangle.p \equiv (\forall q : [p \Rightarrow q] \wedge (q \textbf{ stable in } A) : q)$$

$(p \textbf{ stable in } A)$ holds if no transition of A can falsify p. $\langle A \rangle.p$ is the strongest predicate that both contains p and is stable in A; i.e. it describes the set of all states reachable from p-states via a (possibly empty) sequence of A transitions.

3.2.1 *Continuity*

Some of our results make use of the following semantic property of finite programs. Let *ch* range over totally ordered sets of predicates, and define

$$(A \textbf{ cont}) \equiv (\forall ch : (\forall p \in Ch : (p \textbf{ E } q \textbf{ in } A)) \Rightarrow ((\exists p \in Ch : p) \textbf{ E } q \textbf{ in } A))$$

Intuitively, $(A \textbf{ cont})$ means that whenever A guarantees atomic progress to (i.e., ensurity of) a goal from each of a weakening sequence of starting points, it can achieve atomic progress from their disjunction.

Although not all programs are continuous, most programs of interest can be shown to be continuous using the following theorems:

$$(\{f\} \textbf{ cont}) \tag{3.2}$$
$$(A \textbf{ cont}) \wedge (B \textbf{ cont}) \Rightarrow (A|B \textbf{ cont}) \tag{3.3}$$
$$(\forall i,j : (A_i \textbf{ cont}) \wedge [p_i \wedge p_j \Rightarrow i = j]) \Rightarrow ((|i : (A_i \textbf{ if } p_i)) \textbf{ cont}) \tag{3.4}$$

These rules say that any singleton program is continuous, and that continuity is preserved by finite union or arbitrary disjoint union.

3.3 Achievement

We would like to define a replacement for $\mapsto$, $\leadsto$ ("achieves"), such that an achievement property of the consumer is an achievement property of the whole producer-consumer system. The reason that this doesn't work with $\mapsto$ is that the producer might send another message before the consumer gets a chance to execute. For example, the consumer might be guaranteed to eventually make the channel empty when running in isolation, but not when run in parallel with the producer.

An obvious way to overcome this problem is to define $(p \leadsto q \textbf{ in } A)$ so that it holds if, from a p state, A is guaranteed to reach some state reachable from a q

state (i.e., an $\langle A \rangle.q$ state). To make sure that $\rightsquigarrow$ is transitive, we similarly weaken the antecedent p, yielding the proposed definition

$$? \ (p \ \rightsquigarrow \ q \ \textbf{in} \ A) \equiv (\langle A \rangle.p \ \mapsto \ \langle A \rangle.q \ \textbf{in} \ A)$$

However, this definition is too lenient – because it allows progress "backward in time", it is incompatible with the PSP theorem. For example, if A is the program $\{x := true\}$, we would have $(x \ \rightsquigarrow \ \neg x \ \textbf{in} \ A)$ and $(x \ \mathbf{U} \ false \ \textbf{in} \ A)$; the PSP theorem then yields $(x \ \rightsquigarrow \ false \ \textbf{in} \ A)$, which is not what we want.

The remedy is to build into the definition of $\rightsquigarrow$ a quantification over all possible **U** properties with which it might be combined (using PSP). This leads to the definition

$$\begin{aligned}&(p \ \rightsquigarrow \ q \ \textbf{in} \ A) \equiv \\ &\quad (\forall r, s : (r \ \mathbf{U} \ s \ \textbf{in} \ A) \Rightarrow (\langle A \rangle.(s \vee (r \wedge p)) \ \mapsto \ \langle A \rangle.(s \vee (r \wedge q)) \ \textbf{in} \ A))\end{aligned}$$

(the antecedent has again been weakened to recover transitivity.) To a good approximation, $(p \ \rightsquigarrow \ q \ \textbf{in} \ A)$ means that, for any p-state $s0$,

$$\langle A \rangle.s0 \mapsto \langle A \rangle.(q \wedge \langle A \rangle.s0)$$

that is, from any state reachable from $s0$ (via A), A is guaranteed to reach a state that is reachable from $s0$ via a path that contains a q-state.

The definition of $\rightsquigarrow$ is obviously much too complex to use directly. Thankfully, we don't need to, because we can reason about $\rightsquigarrow$ pretty much as we reason about $\mapsto$. In particular, it satisfies the analogues of (3.1):

$$(p \mapsto q) \Rightarrow (p \rightsquigarrow q) \tag{3.5}$$

$$(p \rightsquigarrow q) \wedge (q \rightsquigarrow r) \Rightarrow (p \rightsquigarrow r) \tag{3.6}$$

$$(\forall i : p_i \rightsquigarrow q_i) \Rightarrow ((\exists i : p_i) \rightsquigarrow (\exists i : q_i)) \tag{3.7}$$

$$(p \rightsquigarrow q) \wedge (r \mathbf{U} s) \Rightarrow ((p \wedge r) \rightsquigarrow (q \wedge r) \vee s)) \tag{3.8}$$

We also need a way to get from $\rightsquigarrow$ back to $\mapsto$:

$$(p \rightsquigarrow q) \wedge (q \ \textbf{stable}) \Rightarrow (p \mapsto q) \tag{3.9}$$

That is, any achievement property whose target is stable is also a leads-to property. We argue that these are the only leads-to properties that really matter, since progress to a predicate that is not stable might never be witnessed by an asynchronous observer. If one accepts this argument, then $\rightsquigarrow$ would appear to be at least as good as $\mapsto$.

The main advantage of achievement is the following powerful composition property: define $(A \ \textbf{dec} \ B)$ ("A is decoupled from B") and $(A \trianglelefteq B)$ ("A is a decoupled component of B") as follows (where **op** ranges over the operators **E**, **U**):

$$\begin{aligned}&(A \ \textbf{dec} \ B) \equiv (\forall p, q, \textbf{op} : (p \ \textbf{op} \ q \ \textbf{in} \ A) \Rightarrow (\langle B \rangle.p \ \textbf{op} \ \langle B \rangle.q \ \textbf{in} \ A)) \\ &(A \trianglelefteq B) \equiv (A \triangleleft B) \wedge (A \ \textbf{dec} \ B) \quad ;\end{aligned}$$

then we have

$$(p \leadsto q \textbf{ in } A) \wedge (A \trianglelefteq B) \Rightarrow (p \leadsto q \textbf{ in } B) \tag{3.10}$$

In other words, if A is a decoupled component of B, then every achievement property of A is also an achievement property of B. Put differently, working with achievement, we can choose which decoupled component is the next one to execute. Clearly, this does not hold for leads-to.

3.4 Decoupling

Like $\leadsto$, the definition of **dec** is too complicated to use directly. Fortunately, decoupling can be established using the following (incomplete) set of rules:

$$(A|B \textbf{ dec } A) \tag{3.11}$$

$$(\forall i : (A_i \textbf{ dec } B)) \Rightarrow ((|i : A_i) \textbf{ dec } B) \tag{3.12}$$

$$(\forall i : (A \textbf{ dec } B_i)) \wedge (A \textbf{ cont}) \Rightarrow (A \textbf{ dec } (|i : B_i)) \tag{3.13}$$

$$(A \textbf{ dec } B) \wedge (p, \neg p \textbf{ stable in } B) \Rightarrow ((A \textbf{ if } p) \textbf{ dec } B) \tag{3.14}$$

$$(\forall i : (A \textbf{ dec } (B \textbf{ if } p_i))) \wedge [(\exists i : p_i)] \Rightarrow (A \textbf{ dec } B) \tag{3.15}$$

$$(\neg p \textbf{ stable in } B) \Rightarrow ((A \textbf{ if } p) \textbf{ dec } (B \textbf{ if } \neg p)) \tag{3.16}$$

It turns out to be useful to consider explicitly the property $(A \textbf{ dec } A|B)$, which we abbreviate $(A \textbf{ wdec } B)$ ("A is weakly decoupled from B"). (Note that $A \triangleleft B \wedge (A \textbf{ wdec } B) \Rightarrow (A \trianglelefteq B)$.) As a rule of thumb, two programs whose interactions are free of race conditions are weakly decoupled from each other, while $(A \textbf{ dec } B)$ means that, in addition, B cannot send information directly to A. Some useful rules for establishing weak decoupling are the following:

$$(A \textbf{ dec } B) \Rightarrow (A \textbf{ wdec } B) \tag{3.17}$$

$$(\forall i,j : (A_i \textbf{ wdec } B) \wedge (A_i \textbf{ wdec } A_j)) \Rightarrow ((|i : A_i) \textbf{ wdec } B) \tag{3.18}$$

$$(\forall i : (A \textbf{ wdec } B_i)) \wedge (A \textbf{ cont}) \Rightarrow (A \textbf{ wdec } (|i : B_i)) \tag{3.19}$$

$$(A \textbf{ wdec } B) \wedge (p \textbf{ stable in } B) \Rightarrow ((A \textbf{ if } p) \textbf{ wdec } B) \tag{3.20}$$

$$(\forall i : (A \textbf{ wdec } (B \textbf{ if } p_i))) \wedge [(\exists i : p_i)] \Rightarrow (A \textbf{ wdec } B) \tag{3.21}$$

$$((A \textbf{ if } p) \textbf{ wdec } (B \textbf{ if } \neg p)) \tag{3.22}$$

As these rules show, decoupling has a better left-union rule ((3.12) vs. (3.18)), which is why we work with decoupling whenever possible. For example, in a producer-consumer system, producers are decoupled from consumers, while consumers are only weakly decoupled from producers. This means that we can allow race conditions in the producer, while keeping it a decoupled component of the system, but not in the consumer (except under unusual circumstances).

For singleton programs, decoupling can be established with the following theorems:

$$(\exists p, q : [p \vee q] \wedge [p; g; f \Rightarrow f; g] \wedge [q; g \Rightarrow 1]) \Rightarrow (\{f\}\ \mathbf{dec}\ \{g\}) \quad (3.23)$$

$$(\exists p, q, r : [p \vee q \vee r] \wedge [r \Rightarrow f] \wedge [p; g; f \Rightarrow f; g] \wedge [q; g \Rightarrow 1]) \Rightarrow (\{f\}\ \mathbf{wdec}\ \{g\}) \quad (3.24)$$

The hypothesis of (3.23) says that g right-commutes with f from every state from which g can possibly change the state; the hypothesis of (3.24) says that g right-commutes with f from every state from which g can possibly change the state and f necessarily changes the state. For example, if f and g interact only through a FIFO channel on which f sends and g receives (both asynchronously), then $(\{f\}\ \mathbf{dec}\ \{g\})$ and $(\{g\}\ \mathbf{wdec}\ \{f\})$. Related forms of commutativity are studied in [14].

3.5 Example — Loosely-coupled programs

A *loosely-coupled* program [11] is one in which (1) every transition is total and deterministic, and (2) from any state from which two transitions can change the state, the transitions commute. (Dataflow networks [8] are the most familiar example.) In such a program, every transition is weakly decoupled from every other (by (3.24)); since singletons are continuous (by (3.2)), each transition is weakly decoupled from the rest of the system (by (3.19)). Grouping transitions arbitrarily into processes, each process is a decoupled component (by (3.18)). Thus, in reasoning about a system, we can choose, at each state, any enabled process to be the next one to execute. Proofs based on this are usually simpler than using the fixed point characterization of [8], since we can often reason about simple (first-order) predicates, instead of having to deal with message sequences.

As a concrete example of this kind of reasoning, consider the following loosely-coupled version of the producer-consumer system described in the introduction. Let ch be a FIFO channel, n a natural counter, let $ch!m$ (resp. $ch?m$) be the actions that send (resp. receive) the message m along the channel ch, and define

$$P = \{(n := n - 1;\ (\exists f : x := f.x;\ ch!f)\ \mathbf{if}\ n > 0)\}$$
$$C = \{((\exists f : ch?f;\ y := f.y)\ \mathbf{if}\ ch \neq <>)\}$$
$$clean \equiv x = y \wedge ch = <>$$
$$mid \equiv (\exists f : x = f.y \wedge ch = \langle f \rangle)$$

We can prove $(\mathit{clean} \mapsto \mathit{clean} \wedge n = 0 \textbf{ in } P|C)$ as follows:

$$
\begin{array}{lll}
1)\ \mathit{clean} \wedge n = N > 0 \mapsto \mathit{mid} \wedge n < N & \textbf{in } P & \text{def } P \\
2)\ \mathit{clean} \wedge n = N > 0 \rightsquigarrow \mathit{mid} \wedge n < N & \textbf{in } P & 1, (3.5) \\
3)\ (P \trianglelefteq P|C) & & (3.24) \\
4)\ \mathit{clean} \wedge n = N > 0 \rightsquigarrow \mathit{mid} \wedge n < N & \textbf{in } P|C & 2, 3, (3.10) \\
) & & \\
5)\ \mathit{mid} \wedge n < N \mapsto \mathit{clean} \wedge n < N & \textbf{in } C & \text{def } C \\
6)\ \mathit{mid} \wedge n < N \rightsquigarrow \mathit{clean} \wedge n < N & \textbf{in } C & 5, (3.5) \\
7)\ (C \trianglelefteq P|C) & & (3.24) \\
8)\ \mathit{mid} \wedge n < N \rightsquigarrow \mathit{clean} \wedge n < N & \textbf{in } P|C & 6, 7, (3.10) \\
) & & \\
9)\ \mathit{clean} \wedge n = N > 0 \rightsquigarrow \mathit{clean} \wedge n < N & \textbf{in } P|C & 4, 8, (3.6) \\
10)\ \mathit{clean} \wedge n = N \rightsquigarrow \mathit{clean} \wedge n = 0 & \textbf{in } P|C & 9, (3.6), \text{ induction} \\
11)\ \mathit{clean} \rightsquigarrow \mathit{clean} \wedge n = 0 & \textbf{in } P|C & 10, (3.7) \\
12)\ ((\mathit{clean} \wedge n = 0) \textbf{ stable in } P|C) & & \text{def } P, C \\
13)\ \mathit{clean} \mapsto \mathit{clean} \wedge n = 0 & \textbf{in } P|C & 11, 12, (3.9)
\end{array}
$$

3.6 Asynchronous safety

Invariants (or, more generally, stable predicates) play a key role in program development. However, asynchrony can make invariants unreasonably complicated. Instead of working with real invariants, we can work with predicates that are reestablished by decoupled components. Because decoupled components can be assumed to execute immediately, these predicates are almost as good as real invariants. The component that reestablishes the invariant is called a "sweeper", because it cleans up after other components.

Sweepers are defined as follows:

$$(A \textbf{ sw } B \textbf{ to } p) \equiv (A \trianglelefteq B) \wedge (\langle B \rangle.p \rightsquigarrow p \textbf{ in } A)$$

Note that sweeping generalizes stability, that is,

$$(p \textbf{ stable in } A) \Leftrightarrow (1 \textbf{ sw } A \textbf{ to } p)$$

A key property of stability is that a predicate is stable in a union of components if it is stable in each component. Sweeping enjoys similar compositionality:

$$(A \textbf{ cont}) \wedge (\forall i : (A \textbf{ sw } B_i \textbf{ to } p) \Rightarrow ((|i : A) \textbf{ sw } (|i : B_i) \textbf{ to } p) \qquad (3.25)$$

The other key property of stability is that it can be combined with progress (or achievement) using a special case of the PSP rule. The corresponding property for sweepers is

$$(A \textbf{ sw } B \textbf{ to } p) \wedge (q \rightsquigarrow r \textbf{ in } B) \Rightarrow (p \wedge q \rightsquigarrow p \wedge \langle A \rangle.r \textbf{ in } B) \qquad (3.26)$$

In most situations, workers can run far ahead of sweepers, and we don't want to have to prove $(\langle B \rangle.p \rightsquigarrow p \textbf{ in } A)$ directly, because $\langle B \rangle.p$ may be complicated;

we would rather sweep up after a single transition of B. In general, if $(A$ **wdec** $B)$ is established using the the rules of section 3.4, then

$$(p \mathbf{U} q \textbf{ in } B) \wedge (A \trianglelefteq B) \wedge (q \leadsto p \textbf{ in } A) \Rightarrow (A \textbf{ sw } B \textbf{ to } p)$$

3.6.1 Example

We modify the producer-consumer example slightly so that termination is caused by a separate component (instead of using a counter in the producer):

$$\begin{aligned} P0 &= \{((\exists f : x := f.x;\ ch!f) \textbf{ if } \neg stop)\} \\ P1 &= \{stop := true\} \\ P &= P0|P1 \\ C &= \{((\exists f : ch?f;\ y := f.y) \textbf{ if } ch \neq <>)\} \\ clean &\equiv (x = y \wedge ch = <>) \\ mid &\equiv (\exists f : x = f.y \wedge ch = \langle f \rangle) \end{aligned}$$

The proof from section 3.5 does not work here, because there is no state variable n to record progress. However, we can instead use a sweeper proof:

1)	$(C$ **wdec** $P0, P1)$			(3.24)
2)	$clean$	**U** mid	**in** $P0$	def $P0$
3)	mid	$\mapsto clean$	**in** C	def C
4)	mid	$\leadsto clean$	**in** C	(3.5)
5)	C **sw** $P0$ **to** $clean$			$1, 2, 4$
)				
6)	$clean$	**U** $false$	**in** $P1$	def $P1$
7)	$false$	$\leadsto clean$	**in** C	(3.5)
8)	C **sw** $P1$ **to** $clean$			$1, 6, 7$
)				
9)	C **sw** P **to** $clean$			$5, 8, (3.25)$
10)	$true$	$\mapsto stop$	**in** $P\|C$	def $P0$
11)	$true$	$\leadsto stop$	**in** $P\|C$	$10, (3.5)$
12)	$clean$	$\leadsto clean \wedge \langle C \rangle.stop$	**in** $P\|C$	$9, 11, (3.26)$
13)	$(stop$ **stable in** $C)$			def C
14)	$clean$	$\leadsto clean \wedge stop$	**in** $P\|C$	$12, 13$
15)	$(clean \wedge stop$ **stable in** $P\|C)$			def P, C
16)	$clean$	$\mapsto clean \wedge stop$	**in** $P\|C$	$14, 15, (3.9)$

3.7 Caveats

While achievement has few disadvantages with respect to leads-to in the context of UNITY-like program development, it does have one disadvantage worth mentioning: unlike linear-time properties, achievement is not preserved by program

refinement[1], so to use a refinement step, it is first necessary to convert achievement properties back to leads-to. This is hardly surprising; related properties like serializability suffer from the same problem.

A minor annoyance in the theory is the continuity requirement. The definition of (A **dec** B) is of the form "if A has this property, it also has that property"; when the property is an **E** property, there is no way in the logic to make sure that the "that" property is being guaranteed by the same transition that guarantees the "this" property (even though it is in most practical cases). This is a minor price to pay for a theory that works entirely at the level of properties, instead of transitions.

A more serious limitation of the theory is shown in the following example. Suppose we have two producer-consumer systems, A producing for B, and C producing for D. Suppose also that these systems multiplex their communications on a shared channel. Sweeper compositionality lets us prove

$$(B \textbf{ sw } A|B \textbf{ to } p) \wedge (D \textbf{ sw } C|D \textbf{ to } p) \Rightarrow (B|D \textbf{ sw } A|B|C|D \textbf{ to } p)$$

(assuming we've correctly labelled multiplexed messages so that B and D don't try to receive the same messages) so things are fine from the sweeper standpoint. However, we would like to prove something stronger, namely ($A|B|D$ **dec** $C|D$), which would, in effect, allow us to pretend that the communication is not multiplexed; we do not know how to strengthen the theory to make this possible. (This problem arose in trying (with Rajeev Joshi) to use sweepers to prove the correspondence of loose and tight executions in Seuss[12]; Joshi eventually resorted to reasoning about actions instead of properties [7].)

Finally, beacuase it is fundamentally about progress, the theory is highly asymmetric with respect to time. Decoupled components work as weak "left-movers"; we can always pretend they happen earlier. They can be composed precisely because all of them are moving in the same direction. Reduction theorems such as [4], on the other hand, allow both left- and right-movers, so message transmissions can be moved later (instead of just moving receptions earlier).

3.8 Conclusions

We have argued that achievement has some desirable properties that make it technically superior to leads-to, particularly when reasoning about asynchronous programs. More generally, we have shown that it is possible to weaken linear-time operators to branching-time operators so as to make them more robust to asynchrony, without changing the essential structure of the logic.

[1]For example, $(true \leadsto x \textbf{ in } \{\})\{(x := false), (x := any)$, but $\neg(true \leadsto x \textbf{ in } \{\})\{(x := false)$.

3.9 Acknowledgements

This work was originally inspired by Jay Misra's paper [11]; it has benefitted from insightful discussions with Jay Misra, J. R. Rao, and Rajeev Joshi, and from the insightful comments of the anonymous referee.

References

[1] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988

[2] E. Cohen. *Modular Progress Proofs of Asynchronous Programs.* PhD. Thesis, University of Texas at Austin, 1993. Available from ftp://ftp.research.telcordia.com/pub/ernie/research/diss.ps.gz .

[3] Ernie Cohen. Separation and reduction. In *Mathematics of Program Construction, 5th International Conference, Portugal, July 2000. Science of Computer Programming*, pages 45–59. Springer-Verlag, 2000.

[4] E. Cohen, L. Lamport. *Reduction in TLA.* In CONCUR98 Springer-Verlag, 1998.

[5] E. Dijkstra and C. Scholten. *Predicate transformers and Program Semantics.* Springer-Verlag.

[6] K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger. *The notions of consistency and predicate locks in a database system.* CACM, 19(11):624-633, 1976.

[7] R. Joshi. *Immediacy: a technique for reasoning about asynchrony.* PhD. Thesis, University of Texas at Austin, 1999.

[8] G. Kahn. *The semantics of a simple language for parallel programming.* In *Proceedings of IFIP Congress '74* North-Holland, 1974.

[9] S. Katz, D. Peled. *Verification of Distributed Programs using Representative Interleaving Sequences.* Distributed Computing, 6:107-120, Springer-Verlag, 1992.

[10] R. J. Lipton. *Reduction: A Method of Proving Properties of Parallel Programs.* CACM 18(12):717-721, 1975.

[11] J. Misra. *Loosely Coupled Programs.* In *Parallel Architectures and Languages Europe*, pages 1—26, June 1991.

[12] J. Misra. *A discipline of multiprogramming — programming theory for distributed applications.* Springer-Verlag, 2001.

[13] J. Pachl. *A simple proof of a completeness result for leads-to in the UNITY logic.* IPL, January 1992.

[14] J. R. Rao. *Extensions of the UNITY Methodology, Compositionality, Fairness and Probability in Parallelism.* Springer LNCS #908, 1995.

[15] A. Valmari. *Stubborn Sets for Reduced State Space Generation.* 10th International Confeence on Application and Theory of petri Nets, Bonn (2) pp 1-22, 1989.

4

A reduction theorem for concurrent object-oriented programs

Jayadev Misra[1]

Abstract

A typical execution of a concurrent program is an interleaving of the threads of its components. It is well known that the net effect of a concurrent execution may be quite different from the serial executions of its components. In this paper we introduce a programming notation for concurrent object-oriented programs, called **Seuss**, and show that concurrent executions of its programs are, under certain conditions, equivalent to serial executions. This allows us to reason about a Seuss program as if its components will be executed serially whereas an implementation may execute its components concurrently, for performance reasons.

4.1 Introduction

A typical execution of a concurrent program is an interleaving of the threads of its component programs. For instance, consider a concurrent program that has α and β as component programs, where the structures of α, β are as follows:

α:: α_1; α_2; α_3, and

β:: β_1; β_2; β_3.

The concurrent execution $\alpha_1\ \beta_1\ \alpha_2\ \beta_2\ \alpha_3\ \beta_3$ interleaves the two sequential executions. It is well known that the net effect of a concurrent execution may be quite different from the serial executions of the components. In this example, suppose α_1, β_1 are "read the value of variable x", α_2, β_2 are "increment the value read" and α_3, β_3 are "store the incremented value in x". Then, the given interleaved execution increases the value of x by 1 whereas an execution in which the threads are not interleaved increases x by 2.

[1]This material is based in part upon work supported by the National Science Foundation Award CCR–9803842.

A. McIver et al. (eds.), *Programming Methodology*

The method of reduction was proposed by Lipton[3] to simplify reasoning about concurrent executions. Lipton develops certain conditions under which the steps of a component program may be considered indivisible (i.e., occurring sequentially) in a concurrent execution. A step f in a component is a *right mover* if for any step h of another component whenever fh is defined then so is hf and they yield the same result (i.e., their executions result in the same final state). Similarly, g is a *left mover* if for any h of another component hg is defined implies gh is defined, and $hg = gh$. Lipton shows that a sequence of steps of a component, $r_0\ r_1\ ...\ r_n\ c\ l_0\ l_1\ ...\ l_m$, may be considered indivisible for proof of termination of a concurrent program if each r_i is a right mover, l_j a left mover and c is unconstrained. This result has been extended to proofs of more general properties by Lamport and Schneider [2], Misra [4], and, more recently, by Cohen and Lamport [1].

In section 4.2, we introduce a programming notation for concurrent object-oriented programming, called **Seuss**. Briefly, a seuss program consists of **boxes**; a box is similar to an object instance. A box has local variables whose values define the state of the box. A box has **actions** and **methods**, both of which will be referred to as **procedures**. Actions are executed autonomously; a method is executed by being called by an action or a method of another box. In section 4.2.2, we introduce two different execution styles for programs, *tight and loose*. In a tight execution an action is completed before another action is started. In a loose execution the actions may be executed concurrently provided they satisfy certain *compatibility* requirements. A tight execution, being a single thread of control, may be understood more easily than a loose execution. Loose execution, on the other hand, is the norm where the computing platform consists of a large number of processors.

In this paper we develop a reduction theorem that establishes that for every loose execution there is a corresponding tight execution: if a loose execution of some finite set of actions starting in state s terminates in state t then there is a tight execution of those actions that can also end in state t starting in state s. This result is demonstrated by prescribing how to transform a loose execution into a tight execution in the above sense. This correspondence allows a programmer to understand a program in terms of its tight executions – a single thread of control – whereas an implementation may exploit the available concurrency through a loose execution.

The proof of the reduction theorem is considerably more difficult in our case because (1) procedure calls introduce interleavings of "execution trees" rather than execution sequences, and (2) executions of any pair of actions may be interleaved provided the actions are compatible. The notion of compatibility is central to our theory. Roughly, two procedures are compatible if their interleaved execution may be simulated by executing them one after the other in some order. We give an exact definition and show how compatibility of procedures may be proven.

Compatibility information can not be deduced automatically. Yet it is unrealistic to expect the user to provide this information for all pairs of procedures; in most cases, different boxes will be coded by different users, and no user may

even know which other procedures will be executing. Therefore, we have developed a theory whereby compatibility of procedures belonging to different boxes may be deduced automatically from the compatibility information about procedures belonging to the same box. Users simply specify which procedures in a box are compatible and an algorithm then determines which pairs of actions are compatible, and may be executed concurrently.

Plan of the paper

In the next section, a brief introduction to Seuss is given; the reader may consult [5] for a detailed treatment. An abstract model of Seuss is given in section 4.3. In section 4.2.1 we state certain restrictions on programs which we elaborate in section 4.4. The definition of compatibility appears in section 4.5. A statement of the reduction theorem and its proof are given in section 4.6. Concluding remarks appear in section 4.7.

4.2 The Seuss programming notation

The central construct in Seuss is **box**; it plays the role of an object. A program consists of a set of boxes. Typically, a user defines generic boxes, called **cat**s (*cat* is short for *category*), and creates several boxes from each cat through instantiation. A cat is similar to a class; a box is similar to a class instance.

The state of a box is given by the values of its variables. The variables are local to the box. Therefore, their values can be changed only by the steps taken within the box. To enable other boxes to change the state of a box, each box includes a set of **procedures** that may be called from outside. Procedure call is the only mechanism for interaction among boxes.

A procedure is either an **action** or a **method**. A method is called by a procedure of another box. An action is not called like a traditional procedure; it is executed from time to time under the following fairness rule: each action is executed eventually. Both actions and methods can change the state (values of the variables) of their own box, and, possibly, of other boxes by calling their methods. A method may have parameters; an action does not have any parameter.

A method may *accept* or *reject* a call made upon it. If the state of the box does not permit a method to execute – for instance, a *get* method on a channel can not execute if the channel is empty – then the call is rejected. Otherwise, the call is accepted. Some methods accept every call; such methods are called **total** methods. A method that may reject a call is called a **partial** method. Similarly, we have total and partial actions.

4.2.1 Seuss syntax

In this section, we introduce a notation for writing programs. The notation is intended for implementation on top of a variety of host languages. Therefore, no

commitment has been made to the syntax of any particular language (there are different implementations with C++ and Java as host languages) and syntactic aspects that are unrelated to the model are left unspecified in the notation.

Notational Conventions

The notation is described using BNF. All non-terminal identifiers are in Roman and all terminal identifiers are in boldface type. The traditional meta symbols of BNF – ::= { } [ ] ( ) – are used, along with $\vee$ to stand for alternation (the usual symbol for alternation, "|", is a terminal symbol in our notation). The special symbols used as terminals are | |/ ; : :: in the syntax given below. A syntactic unit enclosed within "{" and "}" in a production may be instantiated zero or more times, and a unit within "[" and "]" may be instantiated zero or one time. In the right-hand side of a production, $(p \vee q)$ denotes that a choice is to be made between the syntactic units p and q in instantiating this production; we omit the parentheses, "(" and ")", when no confusion can arise. Text enclosed within " { " and " } " in a program is to be treated as a comment.

Program

program ::= **program** program-name {cat $\vee$ box} **end**
cat ::= **cat** cat-name [parameters]: {variable} {procedure} **end**
box ::= **box** box-name [parameters]: cat-name

A program consists of a set of cats and boxes in any order. The declaration of a cat or box includes its name and, possibly, parameters. The names of programs, cats and boxes are identifiers. The parameters of a cat or box could be ordinary variables, cats or boxes. A cat consists of (zero or more) variable declarations followed by procedure declarations. A box is an instance of a cat. Variables are declared and initialized in a cat as in traditional programming languages.

Example

We use a single running example to illustrate the syntax of Seuss. A ubiquitous concept in multiprogramming is the *Semaphore*. The skeletal program given below includes a definition of *Semaphore* as a cat and two instances of *Semaphore*, *s* and *t*. Cat *user* describes a group of users that execute their critical sections only if they hold both semaphores, *s*, *t*; there are three instances of *user*.

---

program *MutualExclusion*
  cat *Semaphore*
    var *n*: nat **init** 1 {initially, the semaphore value is 1}
    {The procedures of *Semaphore* are to be included here}
  end {*Semaphore*}

  box s, t : *Semaphore*

  cat *user*
    var *hs*, *ht*: boolean **init** *false*
    {*hs* is *true* when *user* holds *s*. Similarly, *ht*.}
    {The procedures of *user* are to be included here}
  end {*user*}

  box u, v, w : *user*
end {*MutualExclusion*}

---

procedure

procedure ::= partial-procedure $\vee$ total-procedure
partial-procedure ::= **partial** partial-method $\vee$ partial-action
total-procedure ::= **total** total-method $\vee$ total-action
partial-method ::= **method** head :: partial-body
partial-action ::= **action** [label] :: partial-body
total-method ::= **method** head :: total-body
total-action ::= **action** [label] :: total-body

A procedure is either **partial** or **total**; also, a procedure is either a **method** or an **action**. Thus, there are four possible headings identifying each procedure. Each method has a head and a body. The head is similar to the form used in typical imperative languages; it has a procedure name followed by a list of formal parameters and their types. The labels are optional for actions; they have no effect on program execution.

Example (contd.)

We add the procedure names to the previous skeletal program.

---

program *MutualExclusion*
  cat *Semaphore*
    var *n*: nat **init** 1 {initially, the semaphore value is 1}
    partial method *P*:: { Body of *P* goes here}
    total method *V*:: { Body of *V* goes here}
  end {*Semaphore*}

```
  box s, t : Semaphore

  cat user
    var hs, ht: boolean init false
    partial action s.acquire:: {acquire s and set hs true.}
    partial action t.acquire:: {acquire s and set hs true.}
    partial action execute::
      {Execute this body if both hs, ht are true. Then, set hs, ht false.}
  end {user}

  box u, v, w : user
end {MutualExclusion}
```

---

procedure body

A procedure body has different forms for partial and total procedures. For this manuscript, we take a total-body to be any sequential program. The partial-body is defined by:

partial-body ::= alternative {(| alternative) ∨ (|/alternative)}
alternative ::= precondition [; preprocedure] → total-body
precondition ::= predicate
preprocedure ::= partial-method-call

The body of a partial procedure consists of one or more alternatives. Each alternative has a precondition, an optional preprocedure and a total-body. A precondition is a predicate on the state of the box to which this procedure belongs (i.e., it is constrained to name only the local variables of the box in which the procedure appears). A preprocedure is a call upon a partial method (in some other box).

Example (contd.)

Below, we include code for each procedure body. The partial actions *s.acquire* and *t.acquire* in *user* include calls upon the partial methods *s.P* and *t.P* as preprocedures. The partial action *execute* in *user* calls the total methods *s.V* and *t.V* in its body. The partial action *P* in *Semaphore* has no preprocedure.

```
program MutualExclusion
  cat Semaphore
    var n: nat init 1 {initially, the semaphore value is 1}
    partial method P:: n > 0 → n := n − 1
    total method V:: n := n + 1
  end {Semaphore}
```

```
box s, t : Semaphore

cat user
  var hs, ht: boolean init false
  partial action s.acquire:: ¬hs; s.P → hs := true
  partial action t.acquire:: ¬ht; t.P → ht := true
  partial action execute::
    hs ∧ ht → critical section; s.V; t.V; hs := false; ht := false
end {user}

box u, v, w : user
end {MutualExclusion}
```

The operational semantics of Seuss programs is described in section 4.2.2. The program, given above, may become deadlocked, that is, it may not allow any user to enter its critical section because one may have acquired s and another t. This problem may be avoided by acquiring s, t in order (that is, by changing the precondition of *t.acquire* to $hs \wedge \neg ht$).

Multiple alternatives

Each alternative in a partial procedure is *positive* or *negative*: the first alternative is always positive; an alternative preceded by | is positive and one preceded by |/is negative. For each partial procedure at most one of its alternatives holds in any state; that is, the preconditions in the alternatives of a partial procedure are pairwise disjoint. The distinction between positive and negative alternatives is explained under the operational semantics of Seuss in section 4.2.2.

Restrictions on programs

Procedure Call

A total-body can include a call only to a total method; a partial method cannot be called by a total body. A partial method can only appear as a preprocedure in an alternative of a partial procedure. The syntax specifies that an alternative can have at most one preprocedure. In the example in page 74, partial action *s.acquire* calls *s.P* as a preprocedure, and *execute* calls the total methods *s.V*, *t.V* in its total body (i.e., in the code following →).

Partial Order on Boxes

See section 4.4.1.

Termination Condition

Execution of each total body (the body part of any action, total or partial) must terminate; this is a proof obligation that has to be discharged by the programmer.

The termination condition can be proven by induction on the "level" of a procedure. First, show that any procedure that calls no other procedure terminates whenever it accepts a call. Next, show that execution of any procedure p terminates assuming that executions of all procedures that p calls terminate.

4.2.2 *Seuss semantics (operational)*

At run time, a program consists of a set of boxes; their states are initialized at the beginning of the run. There are two different execution styles for a program. In a *tight execution* one action is executed at a time. There is no notion of concurrent execution; each action completes before the next action is started. In a *loose execution* actions may be executed concurrently.

The programmer understands a program by reasoning about its tight executions only. We have developed a logic for this reasoning. An implementation may choose a loose execution for a program in order to maximize resource utilization. Loose execution is described in Sec. 4.6.1.

Tight execution

A tight execution consists of an infinite number of steps; in each step, an action of a box is chosen and executed as described below (in section 4.2.2). The choice of action to execute in a step is arbitrary except for the following fairness constraint: each action of each box is chosen eventually.

Observe that methods are executed only when they are called from other methods or actions, though actions execute autonomously (and eventually).

Procedure execution

A method is executed when it is called. To simplify description, we imagine that an action is called by a *scheduler*. Then the distinction between a method and an action vanishes; each procedure is executed when called.

A procedure *accepts* or *rejects* a call. A total procedure always *accepts* calls; its body is executed whenever it is called. Termination condition (see section 4.2.1) ensures that execution of each total procedure terminates. A partial procedure may *accept* or *reject* a call. Consider a partial procedure g that consists of a single (positive) alternative; then, g is of the following form:

partial method $g(x, y)$:: p; $h(u, v) \rightarrow S$

Execution of g can be described by the following rules.

---

if $\neg p$ then *reject*
else {p holds} call h with parameters (u, v);
  if h *rejects* then *reject*
  else {h *accepts*}

```
    execute S using parameters, if any, returned by h;
    return parameters, if any, to the caller of g and accept
  endif
endif
```

As stated earlier, the programmer must ensure that execution of each total procedure terminates. It can be then be shown that the execution of any partial procedure g terminates, by using induction on the partial order induced by $\geq_g$ (see section 4.2.1).

The caller is oblivious to rejection, because then its body is not executed and its state remains unchanged. If all alternatives in a program are positive, then the effect of execution of an action is either rejection – then the state does not change for any box – or acceptance – some box state may change then. This is because, if any procedure rejects during the execution of an action then the entire action rejects. If any procedure accepts – the lowest procedure, that has no preprocedure, accepts first, followed by acceptances by its callers in the reverse order of calls – then the entire action accepts. This execution strategy meets the *commit* requirement in databases where a transaction either executes to completion or does not execute at all.

We have described the execution of a partial procedure that has a single (positive) alternative. In case a procedure has several alternatives, positive and negative, the following execution strategy is adopted. Recall that preconditions of the alternatives are disjoint.

```
if preconditions of all alternatives are false then reject
else {precondition of exactly one alternative, f, holds}
  if f is a positive alternative then execute as described previously
  else {f is a negative alternative}
    execute f as a positive alternative except on completion of f:
      reject the call and do not return parameter values
  endif
endif
```

The execution of a negative alternative always results in rejection. The caller is still oblivious to rejection, because its body is not executed and its state remains unchanged. However, a called method may change the state of its own box even when it rejects a call, by executing a negative alternative.

For a partial action the effect of execution is identical for positive and negative alternatives because the scheduler does not discriminate between acceptance and rejection of an action. Therefore, partial actions, generally, have no negative alternatives.

4.3 A model of Seuss programs

In this section, we formalize the notion of box, procedure and executions of procedures (program execution is treated in section 4.6). The *cats* of Seuss are not modeled because they have no relevance at run time. Also, we do not distinguish between action and method because this distinction is unnecessary for the proof of the theorem. Negative alternatives are not considered in the rest of this paper.

- A *box* is a pair (S, P) where
  S is a set of *states* and
  P is a set of *procedures*.
  Each procedure has a unique name and is designated either *partial* or *total*.

- A *procedure* is a tuple (T, N, E) where
  T is a set of *terminal* symbols; each is a binary relation over the states of its box.
  N is a set of *non-terminal* symbols; each is the name of a procedure of another box.
  E is a non-empty set of *executions*, where each execution is a finite string over $T \cup N$.
  An execution of a total procedure is a sequence where each element of the sequence is either a terminal or a total procedure of another box. An execution of a partial procedure is of the form: $b\ h\ e$, where b is a terminal, h – which is optional – is a non-terminal that names a partial procedure of another box, and e is a sequence in which each element is either a terminal or a total procedure of another box.

- A *program* is a finite set of boxes. Program state is given by the box states. (Therefore, each terminal symbol is a binary relation over the program states.)

Convention and Notation:

(1) Terminal symbols of different procedures are distinct.
(2) Each execution of procedure p begins with a $begin_p$ symbol and ends with an end_p symbol. Both of these are terminal symbols of procedure p.
(3) For terminal s, $s.box$ is the box of which s is a symbol. Similarly, $p.box$ is defined for a procedure p.

Justification for the Model

A terminal symbol of a procedure – an element of T – denotes a local step within the procedure. The local step can affect only the state of the corresponding box, and we allow a step to have non-deterministic outcome. Hence, each terminal is modeled as a binary relation over box states.

In the formal model, procedures are parameter-less. Although this would be an absurd assumption in practice, it simplifies mathematical modeling considerably.

We justify this assumption as follows. First, we can remove a value parameter from a procedure by creating a set of procedures, one for each possible value of the parameter, and the caller can decide which procedure to call based on the parameter value. Thus, all value parameters may be removed at the expense of increasing the set of procedures. Next, consider a procedure with result parameters; to be specific, let *read*(w) return a boolean value in w. The caller of *read* cannot decide a priori what the returned value will be. However, we can remove parameter w, as follows. First, model *read* by two different procedures, *readt* and *readf*, which return the values *true* and *false*, respectively. Now, we have two different execution fragments modeling the call upon *read*(w):

readt; $w :=$ *true*, and

readf; $w :=$ *false*.

An execution that calls *read*(w) will be represented by two executions in our model, one for each possible value returned by *read* for w. Thus, we can remove all parameters from procedures.

Next, we justify our model of procedure execution. An execution is a sequence of steps taken by a procedure and the procedures it calls. To motivate further discussion, consider a procedure P that calls *read*(w), described above, twice in succession. The terminal symbols of P are α, β where

α denotes $w :=$ *true*, and β denotes $w :=$ *false*.

The non-terminals of P are *readt* and *readf*, as described above.

An execution of P does the following steps twice: call *read* and then assign the value returned in the parameter to w. If P is executed alone then the possible executions are

$begin_P$ *readt* α *readt* α end_P, and

$begin_P$ *readf* β *readf* β end_P.

These are the *tight* executions of P. If, however, other procedures execute concurrently with P then the value of the boolean could change in between the two read operations (by other concurrently executing procedures) and the loose executions of P are:

$begin_P$ *readt* α *readt* α end_P,

$begin_P$ *readf* β *readf* β end_P,

$begin_P$ *readt* α *readf* β end_P, and

$begin_P$ *readf* β *readt* α end_P.

In particular, the execution $begin_P$ *readt* α *readf* β end_P denotes that the boolean value is changed from *true* to *false* by another procedure during the two calls to *read* by P. Our goal is to model concurrent executions; therefore, we admit all four executions, shown above, as possible executions of P.

We have not specified the initial states of the boxes, because we do not need the initial states to prove the main theorem.

4.4 Restrictions on programs

We impose two restrictions on programs.

- (Partial Order on Boxes) For each procedure, there is a partial order over the boxes of the program such that during execution of that procedure, one procedure may call another only if the former belongs to a higher box than the latter; see section 4.4.1. Different procedures may impose different partial orders on the boxes. A static partial order – i.e., one that is the same for all procedures – is inadequate in practice.
  A consequence of the requirement of partial order is that if some procedure of a box is executing then no procedure of that box is called; therefore, at most one procedure from any box is executing at any moment.
- (Box Condition) For any box, at most one of its procedures may execute at any time; see section 4.4.3. This restriction disallows concurrency within a box.

4.4.1 Partial order on boxes

Definition:

For procedures p, q, we write p *calls* q to mean that p has q as a non-terminal. Let $calls^+$ be the transitive closure of *calls*, and $calls^*$ the reflexive transitive closure of *calls*. Define a relation $calls_p$ over procedures where
$(x\ calls_p\ y) \equiv (p\ calls^*\ x) \wedge (x\ calls\ y)$.

In operational terms, $x\ calls_p\ y$ means procedure x may call procedure y in some execution of procedure p. Each program is required to satisfy the following condition.

Partial Order on Boxes:

For every procedure p, there is a partial order $\geq_p$ over the boxes such that
$x\ calls_p\ y \Rightarrow x.box >_p y.box$.

Note: $b >_p c$ is a shorthand for $b \geq_p c \ \wedge\ b \neq c$. Relation $\geq_p$ is reflexive and $>_p$ is irreflexive.

Observation 1:

$p\ calls^*\ x \Rightarrow p.box \geq_p x.box$, and
$p\ calls^+\ x \Rightarrow p.box >_p x.box$.

Proof: Define $calls^i$, for $i \geq 0$, as follows.

$p\ calls^0\ p$, and
$p\ calls^{i+1}\ q \equiv (\exists r :: p\ calls^i\ r\ \wedge\ r\ calls\ q)$.

Using induction over i we can show that

$p\ calls^i\ x \Rightarrow p.box \geq_p x.box$, for all i, $i \geq 0$
$p\ calls^i\ x \Rightarrow p.box >_p x.box$, for all i, $i > 0$.

The desired results follow by noting that

$p\ calls^*\ x \equiv (\exists i : i \geq 0 : p\ calls^i\ x)$, and
$p\ calls^+\ x \equiv (\exists i : i > 0 : p\ calls^i\ x)$. □

Note that $p\ calls^+\ q \Rightarrow$ {by Observation 1} $(p.box >_p q.box) \Rightarrow p, q$ are in different boxes. It follows that no call is ever made upon a box when one of its procedures has started but not completed its execution.

Observation 2:

$calls^+$ is an acyclic (i.e., irreflexive, asymmetric and transitive) relation over the procedures.
Proof: From its definition $calls^+$ is transitive. Also, $p\ calls^+\ p \Rightarrow$ {from Observation 1} $p.box >_p p.box$, a contradiction. Therefore, $calls^+$ is irreflexive. Asymmetry of $calls^+$ follows similarly.

Definition:

The *height* of a procedure is a natural number. The height is 0 if the procedure has no non-terminal. Otherwise, $p\ calls\ q \Rightarrow p.height > q.height$. This definition of height is well-grounded because $calls^+$ induces an acyclic relation on the procedures.

Definition:

An *execution tree* of procedure p is an ordered tree where (1) the root is labeled p, (2) every non-leaf node is labeled with a non-terminal symbol, and (3) the sequence of labels of the children of a non-leaf node q is an execution of q. A *full execution tree* is an execution tree in which each leaf node is labeled with a terminal symbol.

Any execution tree of procedure p is finite. This is because if procedure q is an ancestor of procedure r in this tree then $q\ calls_p\ r$; hence, $q.box >_p r.box$. Since the program has a finite number of boxes, each path in the tree is finite; also, the degree of each node is finite because each execution is finite in length. From Koenig's lemma, the tree is finite.

Definition:

The *frontier* of an execution tree is the ordered sequence of symbols in the leaf nodes of the tree. An *expanded execution* of procedure p is the frontier of some full execution tree of p. Hence, an expanded execution consists of terminals only.

4.4.2 *Procedures as relations*

With each terminal symbol we have associated a binary relation over program states. Next, we associate such a relation with each procedure and each execution

of a procedure; to simplify notation we use the same symbol for an execution (or a procedure) and its associated relation. For execution e, $(u, v) \in e$ means that if e is started in state u then it is possible for it to end in state v. For a procedure p, $(u, v) \in p$ means that there is an execution e of p such that $(u, v) \in e$. Formally,

- The relation for a procedure is the union of relations of all its executions.
- The relation for an execution $x_0, \cdots, x_n$ is the relational product of the sequence of relations corresponding to the x_i's.

Observe that a symbol x_i in an execution may be a terminal for which the relation has already been defined, or a non-terminal for which the relation has to be computed using this definition. We show in the following lemma that the rules given above define unique relations for each execution and procedure; the key to the proof is the acyclicity of $calls^+$.

Lemma 1:

There is a unique relation for each procedure and each execution.

Proof: We prove the result by induction on n, the height of a procedure.

For $n = 0$: The procedure has only terminals in all its executions. The relation associated with any execution of the procedure is the relational product of its terminals. The relation associated with the procedure is the union of all its executions, and, hence, is uniquely determined.

For $n > 0$: Each execution of the procedure has terminals (for which the relations are given) or non-terminals (whose heights are at most n, and, hence, they have unique relations associated with them). Therefore, the relation for an execution –which is the relational product of the sequence of relations of its terminals and non-terminals– is uniquely determined. So, the relation for the procedure is also uniquely determined. □

Note that an execution may have the empty relation associated with it, denoting that the steps of the execution will never appear contiguously in a program execution. Such is the case with the execution *read* α *read* β in the example of section 4.3, where two successive reads of the same variable yield different values. Such an execution may appear as a non-contiguous subsequence in a program execution where steps of another procedure's execution could alter the value of the variable in between the two read operations.

Henceforth, each symbol – terminal or non-terminal – has an associated binary relation over program states. Concatenation of symbols corresponds to their relational product. For strings x, y, we write $x \subseteq y$ to denote that the relation corresponding to x is a subset of the relation corresponding to y.

Observation 3:

For terminal symbols s, t of different boxes, $st = ts$ (i.e., the relations st and ts are identical).

4.4.3 *Box condition*

The execution strategy for a program ensures that at most one procedure from a box executes at any time. This strategy can be encoded in our model by making it impossible for procedure q to start if procedure p of the same box has started and not yet completed. This is formalized below.

Definition:

Let σ and τ be sequences of symbols (terminals and non-terminals). Procedure p is *incomplete* after σ (before τ in $\sigma\tau$) if σ contains fewer end_p's than $begin_p$'s.

Box Condition

Let p, q be procedures of the same box, and p be incomplete after σ. Then, $\sigma\ begin_q = \epsilon$, where ϵ denotes the empty relation.

The following lemma shows that under certain conditions a terminal symbol can be transposed with a non-terminal symbol adjacent to it .

Lemma 2:

Let p, q be procedures, t a terminal of p, and σ any sequence of symbols.
1. If p is incomplete after σ then $\sigma\ q\ t \subseteq \sigma\ t\ q$.
2. If p is incomplete after $\sigma\ t$ then $\sigma\ t\ q \subseteq \sigma\ q\ t$.
Proof: We prove the first part. The other part is left to the reader.

$$
\begin{array}{ll}
 & \sigma\ q\ t \\
= & \{q \text{ is the union of all its expanded executions, } g\} \\
 & (\cup_g(\sigma\ g\ t)) \\
= & \{\text{partition } g \text{ into } e,f;\ e \text{ has a terminal from } p.box, \text{ and } f \text{ does not}\} \\
 & (\cup_e(\sigma\ e\ t)) \cup (\cup_f(\sigma\ f\ t)) \\
= & \{e \text{ is of the form } \sigma' begin_r\ \sigma'', \text{ where:} \\
 & \quad \sigma' \text{ has no terminal from } p.box;\ r \text{ is some procedure from } p.box\} \\
 & (\cup(\sigma\sigma'\ begin_r\ \sigma''\ t)) \cup (\cup_f(\sigma\ f\ t)) \\
= & \{\sigma\sigma'\ begin_r = \epsilon, \text{ because from Box Condition:} \\
 & \quad p \text{ is incomplete after } \sigma, \text{ and hence, after } \sigma\sigma', \text{ and } r.box = p.box\} \\
 & (\cup_f(\sigma\ f\ t)) \\
= & \{f \text{ has no terminal from } p.box,\ t \text{ is a terminal of } p.box; \text{ Observation 3}\} \\
 & (\cup_f(\sigma\ t\ f)) \\
\subseteq & \{f \text{ is a subset of the (expanded) executions of } q\} \\
 & \sigma\ t\ q
\end{array}
$$

4.5 Compatibility

A loose execution of a program allows only compatible actions to be executed concurrently. We give a definition of compatibility in this section. We expect the

user to specify the compatibility relation for procedures within each box; then the compatibility relation among all procedures (in different boxes) can be computed automatically in linear time from the definition given below.

Procedures p, q are *compatible*, denoted by $p \sim q$, if all of the following conditions hold. Observe that $\sim$ is a symmetric relation.

C0. p *calls* $p' \Rightarrow p' \sim q$, and q *calls* $q' \Rightarrow p \sim q'$.

C1. If p, q are in the same box,
(p is total $\Rightarrow qp \subseteq pq$), and
(q is total $\Rightarrow pq \subseteq qp$).

C2. If p, q are in different boxes, the transitive closure of the relation $(\geq_p \cup \geq_q)$ is a partial order over the boxes.

Condition C0 requires that procedures that are called by compatible procedures be compatible. Condition C1 says that for p, q in the same box, the effect of executing a partial procedure and then a total procedure can be simulated by executing them in the reverse order. Condition C2 says that compatible procedures impose similar (i.e., non-conflicting) partial orders on boxes.

Notes:

(1) If partial procedures p, q of the same box call no other procedure then they are compatible.
(2) Total procedures p, q of the same box are compatible only if $pq = qp$.
(3) The condition (C0) is well-grounded because if p *calls* p' then the height of p exceeds that of p'.
(4) In a Seuss program compatibility of procedures with parameters has to be established by checking the compatibility with all possible values of parameters; see the example of channels in section 4.5.1

4.5.1 Examples of compatibility

Semaphore

Consider the *Semaphore* box of page 74. We show that $V \sim V$ and $P \sim V$, i.e.,

$VV = VV$, and
$PV \subseteq VP$

The first identity is trivial. For the second identity, we compute the relations corresponding to P and V, as follows:

$$
\begin{array}{ll}
 & P \\
= & \{\text{from the program text}\} \\
 & (n > 0) \times (n := n - 1) \\
= & \{\text{definitions of predicate and assignment}\}
\end{array}
$$

$$\begin{array}{ll} & \{(x,x) \mid x > 0\} \times \{(x,x-1) \mid x > 0\} \\ = & \{\text{simplifying}\} \\ & \{(x,x-1) \mid x > 0\} \end{array}$$

Similarly, $V = \{(x,x+1) \mid x \geq 0\}$. Taking relational product, $PV = \{(x,x) \mid x > 0\}$, and $VP = \{(x,x) \mid x \geq 0\}$. Therefore, $PV \subseteq VP$.

Channels

Consider the unbounded FIFO channel of sectionthat $get \sim put$, i.e., for any x, y,

$$get(x)\ put(y) \subseteq put(y)\ get(x)$$

That is, any state reachable by executing $get(x)\ put(y)$ is also reachable by executing $put(y)\ get(x)$ starting from the same initial state.

Let $(u,v) \in get(x)\ put(y)$. We show that $(u,v) \in put(y)\ get(x)$. Given $(u,v) \in get(x)\ put(y)$, we conclude from the definition of relational composition, that there is a state w such that $(u,w) \in get(x)$ and $(w,v) \in put(y)$. Since $(u,w) \in get(x)$, from the implementation of get, u represents a state where the channel is non-empty; i.e., the channel state s is of the form $a +\!\!+ S$, for some item a and a sequence of items S. Then we have

$$\{s = a +\!\!+ S\}\ put(y)\ \{s = a +\!\!+ S +\!\!+ y\}\ get(x)\ \{x +\!\!+ s = a +\!\!+ S +\!\!+ y\}$$
$$\{s = a +\!\!+ S\}\ get(x)\ \{x +\!\!+ s = a +\!\!+ S\}\ put(y)\ \{x +\!\!+ s = a +\!\!+ S +\!\!+ y\}$$

The final states, given by the values of x and s, are identical. This completes the proof.

The preceding argument shows that two procedures from different boxes that call *put* and *get* (i.e., a sender and a receiver) may execute concurrently. Further, since $get \sim get$ by definition, multiple receivers may also execute concurrently. However, it is not the case that $put \sim put$ for arbitrary x, y, that is,

$$put(x)\ put(y) \neq put(y)\ put(x)$$

because a FIFO channel is a sequence, and appending a pair of items in different orders results in different sequences. Therefore, multiple senders may not execute concurrently.

Next, consider concurrent executions of multiple senders and receivers, as is the case in a client-server type interaction. As we have noted in the last paragraph, multiple senders may not execute concurrently on a FIFO channel. Therefore, we use an unordered channel, of section for communication in this case. We show that $put \sim put$ and $put \sim get$ for unordered channel, i.e., for all x, y

$$\begin{array}{ll} put(x)\ put(y) & = put(y)\ put(x)\ , \text{and} \\ get(x)\ put(y) & \subseteq put(y)\ get(x) \end{array}$$

The proof of the first identity is trivial because *put* is implemented as a bag union. The proof of the second result is similar to that for the FIFO channel. We need consider the initial states where the bag b is non-empty. In the following, $x \cup b$ is an abbreviation for $\{x\} \cup b$.

$\{b = B, B \neq empty\}\ get(x)\ \{x \cup b = B\}\ put(y)\ \{x \in B,\ x \cup b = B \cup y\}$
$\{b = B, B \neq empty\}\ put(y)\ \{b = B \cup y\}\ get(x)\ \{x \in (B \cup y),\ x \cup b = B \cup y\}$

The postcondition of (1) implies the postcondition of (2) because $x \in B \Rightarrow x \in (B \cup y)$. Hence, any final state of $get(x)\ put(y)$ is also a final state of $put(y)\ get(x)$.

4.5.2 *Semi-commutativity of compatible procedures*

In Lemma 3, below, we prove a result for compatible procedures analogous to condition C1 of page 84. This result applies to any pair of compatible procedures, not necessarily those in the same box.

Lemma 3:

Let $p \sim q$ where p is total (p, q need not belong to the same box). Then $qp \subseteq pq$.

Proof: We apply induction on n, the sum of the heights of p and q, to prove the result. The result holds from the definition of $\sim$ if p, q are in the same box. Assume, therefore, that p, q are in different boxes.

For $n = 0$: Both p, q are at height 0; hence, p, q have only terminals in all their executions. Since, p, q are from different boxes, the result follows by repeated application of Observation 3.

For $n > 0$: From (C2), the transitive closure of $(\geq_p \cup \geq_q)$ is a partial order over the boxes; we abbreviate this relation by $\geq$. We prove the result for the case where $\neg(q.box > p.box)$. A similar argument applies for the remaining case, $\neg(p.box > q.box)$. Consider an execution, e, of p. Let x be any symbol in that execution. We show that $qx \subseteq xq$.

- x is a terminal: Consider any expanded execution of q. A terminal t in this expanded execution is a symbol of procedure r where $q\ calls^*\ r$.

$$
\begin{array}{ll}
 & x.box = t.box \\
\Rightarrow & \{x, t \text{ are terminals of } p, r, \text{ respectively}\} \\
 & x.box = t.box \ \wedge\ x.box = p.box \ \wedge\ t.box = r.box \\
\Rightarrow & \{\text{logic}\} \\
 & p.box = r.box \\
\Rightarrow & \{q\ calls^*\ r;\ \text{Observation 1}\} \\
 & p.box = r.box \ \wedge\ q.box \geq_q r.box \\
\Rightarrow & \{\text{logic}\} \\
 & q.box \geq_q p.box \\
\Rightarrow & \{\geq \text{ is the transitive closure of } (\geq_p \cup \geq_q)\} \\
 & q.box \geq p.box \\
\Rightarrow & \{p, q \text{ are from different boxes}\} \\
 & q.box > p.box \\
\Rightarrow & \{\text{assumption: } \neg(q.box > p.box)\} \\
 & false
\end{array}
$$

Thus, x, t belong to different boxes, and from Observation 3, $xt = tx$. Applying this argument for all terminals t in the expanded execution of q, we have $qx = xq$.

- x is a non-terminal: From (C0), $x \sim q$. The combined heights of x and q is less than n. Also, x is total, since it is a non-terminal of p, and p is total. From the induction hypothesis, $qx \subseteq xq$.

Next we show that for any execution e of p, $qe \subseteq eq$. Proof is by induction on the length of e. If the length of e is 1 then the result follows from $qx \subseteq xq$. For e of the form fx:

$$
\begin{array}{ll}
 & qfx \\
\subseteq & \{\text{Induction: } qf \subseteq fq\text{; monotonicity of relational product}\} \\
 & fqx \\
\subseteq & \{qx \subseteq xq\text{; monotonicity of relational product}\} \\
 & fxq
\end{array}
$$

Next, we show $qp \subseteq pq$.

$$
\begin{array}{ll}
 & qp \\
= & \{\text{definition of } p\} \\
 & q(\cup_{e \in p}\, e) \\
= & \{\text{distributivity of relational product over union}\} \\
 & (\cup_{e \in p}\, qe) \\
\subseteq & \{qe \subseteq eq \text{ from the above proof}\} \\
 & (\cup_{e \in p}\, eq) \\
= & \{\text{distributivity of relational product over union}\} \\
 & (\cup_{e \in p}\, e)q \\
= & \{\text{definition of } p\} \\
 & pq
\end{array}
$$

□

Lemma 4:

$(p \sim q \wedge p\ calls^*\ p' \wedge\ q\ calls^*\ q') \Rightarrow (p' \sim q')$.

Proof: The result follows from

$$(p \sim q\ \wedge p\ calls^i\ p'\ \wedge\ q\ calls^j\ q') \Rightarrow (p' \sim q')$$

which is proved by induction on $i + j$, $i, j \geq 0$.

4.6 Proof of the reduction theorem

A finite *loose execution* of a program is a finite sequence of steps taken by some of the procedures of the program. The executions of the procedures could be interleaved. A loose execution satisfies: (1) the steps taken by each procedure is an

expanded execution of that procedure, and (2) executions of two procedures are interleaved only if they are both part of the execution of the same procedure, or if they are compatible.

In this section, we formally define loose execution of a program and show a scheme to convert a loose execution into a tight execution. The reduction scheme establishes the following theorem.

Reduction Theorem:

Let E be a finite loose execution of a program. There exists a tight execution F of the program such that $E \subseteq F$.

4.6.1 Loose execution

A loose execution is given by: (1) a finite set of full execution trees (of some of the procedures), and (2) a finite sequence of terminals called a *run*. the relation corresponding to a loose execution is the relational product of the terminals in the run. Each execution tree (henceforth called a tree) depicts the steps of one action in this loose execution, and the run specifies the interleaving of the executed steps. The trees and the run satisfy the conditions M0 and M1, given below.

Condition M0 states that each symbol of the run can be uniquely identified with a leaf node of some tree, and conversely, and that the loose execution contains the procedure executions (the frontiers of the corresponding trees) as subsequences. Since each symbol of the run belongs to a tree we write $x.root$ for the root of the tree that symbol x belongs to.

Condition M1 states that if two procedures are incomplete at any point in the run then they either belong to the same tree (i.e., they are part of the same execution) or they are compatible.

- (**M0**) There is a 1-1 correspondence between the symbols in the run and the leaf nodes of the trees. The subsequence of the run corresponding to symbols from a tree T is the frontier of T.

- (**M1**) Suppose procedure p is incomplete before symbol s in the run. Then, either $p.root = s.root$ or $p.root \sim s.root$.

4.6.2 Reduction scheme

Suppose R is the run of some loose execution. We transform run R and the execution trees in stages; let R' denote the transformed run. The transformed run may consist of terminals as well as non-terminals, and its execution trees need not be full (i.e., leaf nodes may have non-terminal labels). We show how to transform the execution trees and the run so that the following invariants are maintained. Note the similarity of N0, N1 with M0, M1.

- **(N0)** There is a 1-1 correspondence between the symbols in the run and the leaf nodes of the trees. The subsequence of the run corresponding to symbols from a tree T is the frontier of T.

- **(N1)** Suppose procedure p is incomplete before symbol s in the run. Then, either $p.root = s.root$ or $p.root \sim s.root$.

- **(N2)** $R \subseteq R'$.

The conditions (N0, N1, N2) are initially satisfied by the given run and the execution trees: N0, N1 follow respectively from M0, M1, and N2 holds because $R = R'$.

The reduction process terminates when there are no *end* symbols in the run; then all symbols are the roots of the trees. This run corresponds to a tight execution, and according to N2, it establishes the reduction theorem. The resulting tight execution can simulate the original loose execution: if the original execution starting in a state u can lead to a final state v then so does the final tight execution.

For a run that contains an *end* symbol, we apply either a *replacement* step or a *transposition* step. Let the first *end* symbol appearing in the run belong to procedure q.

Replacement Step:

If a contiguous subsequence of the run corresponds to the frontier of a subtree rooted at q (then the subsequence is an execution of q) replace the subsequence by the symbol q, and delete the subtree rooted at q (retaining q as a leaf node).

This step preserves N0. N1 also holds because for any symbol x in the execution that is replaced by q, $p.root \sim x.root$ prior to replacement, and $x.root = q.root$. Hence, $p.root \sim q.root$ after the replacement. The relation for a procedure is weaker than for any of its executions; therefore, the replacement step preserves N2.

Transposition step:

If a run has an *end* symbol, and a replacement step is not applicable then execution of some procedure q is non-contiguous. We then apply a transposition step to transpose two adjacent symbols in the run (leaving the execution trees unchanged) that makes the symbols of q more contiguous. Continued transpositions make it possible to apply a replacement step eventually.

Suppose q is a partial procedure (similar arguments apply to partial procedures that have no preprocedures and to total procedures). An execution of procedure q is of the form $(begin_q\ b\ h\ \cdots\ x \cdots\ end_q)$ where h is the preprocedure of q and x is either a terminal symbol or a non-terminal, designating a total procedure, of q. All procedures that complete before q have already been replaced by non-terminals, because the first *end* symbol appearing in the run belongs to q. Note that h is a procedure that completes before q.

Suppose x is preceded by y which is not part of the execution of q. We show how to bring x closer to h. Transposing x, y preserves N0, N1. We show below that transposition preserves N2, as well.

- Case 0 (Both x, y are terminals): Let y be a terminal of procedures p. Procedure q is incomplete before y because its end_q symbol comes later. If p, q are in the same box then the relation corresponding to prefix σ of the run up to y is ϵ, from the Box condition. Hence, $\sigma\ y\ x = \sigma\ x\ y$. If p, q belong to different boxes, from Observation 3, the symbols x, y can be transposed.

- Case 1 (Both x, y are non-terminals): Symbol x is part of q's execution; therefore, $q.root\ calls^*\ x$. Symbol y is not a part of q's execution, nor can it be a part of the execution of any procedure that calls q because q is incomplete before y; therefore, $q.root \neq y.root$.

$$
\begin{array}{ll}
 & q \text{ is incomplete just before } y \\
\Rightarrow & \{\text{(N1)}\} \\
 & q.root = y.root \ \vee\ q.root \sim y.root \\
\Rightarrow & \{q.root \neq y.root \text{ (see above)}\} \\
 & q.root \sim y.root \\
\Rightarrow & \{q.root\ calls^*\ x \ \wedge\ y.root\ calls^*\ y; \text{ Lemma 4}\} \\
 & x \sim y \\
\Rightarrow & \{x \text{ is total; Lemma 3}\} \\
 & yx \subseteq xy
\end{array}
$$

- Case 2 (x is a terminal, y a non-terminal): q is incomplete just before y. Applying Lemma 2 (part 1), x, y may be transposed.

- Case 3 (x is a non-terminal, y is a terminal): Let Y be the procedure of which y is a symbol. Since the first *end* symbol in the run belongs to q, end_Y comes after x. Therefore, Y is incomplete before x. Applying Lemma 2 (part 2) with Y as the incomplete procedure, x, y may be transposed.

Thus, x, y may be transposed in all cases, preserving N3. Hence, all symbols in the execution of q to the right of h can be brought next to h.

Next, we bring the $begin_q$ symbol and the predicate b next to h, using an argument similar to Case 3, above. Thus, all of q's symbols to the left and right of h can be made contiguous around h, and a replacement step can then be applied.

For a total procedure q the reduction is done similarly; $begin_q$ serves the role of h in the above argument. For a procedure q that has no preprocedure, the reduction process is similar with b serving the role of h.

Proof of Termination of the Reduction Scheme

We show that only a finite number of replacement and transposition steps can be applied to any loose execution. For a given run, consider the procedure q whose *end* symbol, end_q, is the first *end* symbol in the run. Define two parameters of the run, n, c, as follows.

n = number of end symbols in the run,
$c = \Sigma\, c_j$,

where c_j is the number of symbols not belonging to q between the preprocedure h of q and the j^{th} symbol of q, and the sum is over all symbols of q. c has an arbitrary value if the run has no *end* symbol.

The pair (n, c) decreases lexicographically with each transposition and replacement step. This is because a replacement step removes one end symbol from the run, thus decreasing n. A transposition step decreases c while keeping n unchanged. Ultimately, therefore, n will become 0; then the run has no *end* symbol, and, from (N0), the symbols are the roots of the execution trees.

4.7 Concluding remarks

The following variation of the Reduction theorem may be useful for applications on the world-wide web. Consider a Seuss program in which every procedure calls at most one other procedure. Define all pairs of procedures to be compatible. The reduction theorem then holds: any loose execution may be simulated by some tight execution.

The proof of this result is similar to the proof already given. As before, we reduce procedure q, whose end symbol, end_q, is the first end symbol in the run. If this procedure calls no other procedure then all its symbols are terminals and, by applying Case (0) and Case (2) of the transposition step, we can bring all its symbols together next to its first symbol. If the procedure calls another procedure then, according to the reduction procedure, the called procedure has already been reduced and we bring all the symbols next to the called procedure symbol in a similar fashion.

The major simplification in the reduction scheme for this special case is due to the fact that it is never necessary to transpose two non-terminals. Therefore, Case (1) of the transposition step never arises. Consequently, the condition for compatibility of two procedures (page 84) is irrelevant in this case.

Acknowledgments

This paper owes a great deal to discussions with Rajeev Joshi and Will Adams. I am grateful to Carroll Morgan who gave me useful comments on an earlier draft. Ernie Cohen has taught me a great deal about reduction theorems, in general.

References

[1] Ernie Cohen and Leslie Lamport. Reduction in TLA. In David Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes*

in Computer Science, pages 317–331. Springer-Verlag, 1998. Compaq SRC Research Note 1998-005.

[2] L. Lamport and Fred B. Schneider. Pretending atomicity. Technical Report 44, DEC Systems Research Center, May 1989.

[3] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[4] Jayadev Misra. Loosely coupled processes. *Future Generation Computer Systems*, 8:269–286, 1992. North-Holland.

[5] Jayadev Misra. *A Discipline of Multiprogramming*. Monographs in Computer Science. Springer-Verlag, New York, 2001. The first chapter is available at `http://www.cs.utexas.edu/users/psp/discipline.ps.gz`

5

Abstractions from time

Manfred Broy

Abstract

Mathematical models of the timed behaviour of system components form a hierarchy of timing concepts. This is demonstrated for systems that communicate via input and output streams. We distinguish *non-timed streams*, *discrete streams* with *discrete* and with *continuous time*, and *dense streams with continuous time*. We demonstrate how exchanges of the timing models during the system-development process are captured as classical abstraction steps.

5.1 Introduction

Although the timing of events is an important issue for many information processing systems, all the first attempts to provide logical, algebraic, or mathematical foundations for programming and for system development tried to abstract entirely from timing issues. This is of course fine as long as we are only interested in sequential, non-reactive algorithms. However, looking at interactive systems, especially at reactive embedded systems, timing issues immediately become crucial. In fact, many application systems of today have to react within timed bounds to time events. However, at a logical level of system specification and design the main issue is not the reaction within time bounds, but rather the reaction to abstract events. Only if there is no sensor to record such events, and if by physical theories time bounds on the events are available, can quantitative time replace the observation of events.

We are interested in the following in the description of components that react interactively to input by output. Operationally, input and output take place within a global time frame. It is one of the goals of this paper to show what consequences the abstraction from time within a semantic model actually has. In fact, the semantics becomes less robust since the flow of time leads to a quite explicit modelling of causality and thus to more realistic, simpler models of computation. As a consequence, fixpoint theory becomes more straightforward as well, and does not

A. McIver et al. (eds.), *Programming Methodology*

need more sophisticated theoretical concepts such as least fixpoints, complete partially-ordered sets or metric spaces (see [17] and [18]). This simplicity is lost, however, if we abstract from timing information partially or completely. Exemplars for these problems are models of computations based on the idea of full synchrony (see [5]). Here the lack of explicit information about causality leads into semantic pathologies such as *causal loops*.

In the following, we introduce a semantic model of system behaviour that includes discrete and dense streams with discrete and continuous time. In the first section, we introduce our mathematical basis. Then we show how to describe the syntactic interfaces and the dynamic behaviours of interactive systems. We introduce concepts for systematic and schematic abstractions of time. In particular, we show how the different time models can be related by refinement relations.

5.2 Streams

Streams are helpful models for many aspects of information processing systems. A stream describes the communication history of a channel, the flow of values assumed by a variable of a system, or the sequence of actions executed.

5.2.1 Mathematical foundation: streams

By $\mathbb{N}$ we denote the set of natural numbers $\{0, 1, \ldots\}$; by $\mathbb{N}^+$ we denote $\mathbb{N}\backslash\{0\}$. By $\{i, \ldots, j\}$ we denote for $i, j \in \mathbb{N}$ the set $\{n \in \mathbb{N} : i \leq n \leq j\}$. By $\mathbb{R}$ we denote the set of real numbers and by $\mathbb{R}^+$ the set $\{r \in \mathbb{R} : 0 < r\}$. By $[r : s]$ we denote for $r, s \in \mathbb{R}$ the set $\{x \in \mathbb{R} : r \leq x \leq s\}$, by $[r : s[$ we denote for $r, s \in \mathbb{R}$ the set $\{x \in \mathbb{R} : r \leq x < s\}$ and by $]r : s]$ we denote for $r, s \in \mathbb{R}$ the set $\{x \in \mathbb{R} : r < x \leq s\}$. By M^* we denote the set of finite sequences over the set M.

A time domain is a linearly ordered set of elements representing time points. A stream is a mapping

$$s : T \to S(M)$$

where T is a time domain and $S(M)$ is the stream domain. Typically, the stream domain $S(M)$ is identical to M or M^*.

An example of a time domain is the set $\mathbb{R}^+$ of the positive reals. *Continuous infinite streams* of sort M are mappings from the positive reals into the set M. Hence a continuous infinite stream s is a mapping

$$s : \mathbb{R}^+ \to M$$

A finite continuous stream is a mapping

$$s :]0 : r] \to M$$

where $r \in \mathbb{R}$. We call r the length of the stream s and denote it by $\#s$. Also concatenation easily extends from discrete to continuous streams.

Next we study four different classes of streams in connection with the modelling of time: *non-timed streams*, *discrete streams* with *discrete* and *continuous* time, and finally *dense streams* with *continuous time*.

5.2.2 *Modelling time*

Streams represent communication histories for sequential channels. Given a set M of messages, a non-timed history for a sequential channel is given by a discrete stream of sort M. Such a stream reflects the order in which the messages are communicated. It does not contain any quantitative aspects of the timing of its messages. Hence we speak of a *non-timed* stream.

If additional quantitative time information is contained, we speak of a timed stream. In the following, we are interested in separating aspects of data and message flow of a channel from timing aspects.

Typical time models that we find in the literature are the natural numbers $\mathbb{N}$ and the positive real numbers $\mathbb{R}^+$. We might also work with the rational numbers, however: as long as we do not study limits and infinitely small differences, there is not a significant difference between the real numbers and the rational numbers when modelling time. These numbers are all models of *linear* time. Linear time is most appropriate for system models with a *global* time.

A timed communication history for a channel carrying messages from a given set M is represented by a timed stream. A timed stream with discrete time is a finite or infinite sequence of messages with additional timing information from a discrete time space. We work with the following models of streams with notions of quantitative time.

	time domain	stream domain
non-timed streams	$T = \mathbb{N} \vee \exists n \in \mathbb{N} : T = \{0, \ldots, n\}$	M
discrete streams/ discrete time	$T \subseteq \mathbb{N}$	M^*
discrete streams/ continuous time	$T \subseteq \mathbb{R}^+$where T is finite or countable	M
dense streams/ continuous time	$T = \mathbb{R}^+ \vee \exists r \in \mathbb{R}^+ : T = [0 : r[$	M

Note that we use M^* as the stream domain in the case of discrete streams with discrete time to allow for several messages in one time slot.

$s{\downarrow}i$ denotes the communication history of the stream s till time i. We extend this notation, of truncating streams at time points, to sets W of streams pointwise, as follows

$$W{\downarrow}t = \{s{\downarrow}t : s \in W\}$$

Let s be a discrete stream with discrete time and with stream domain M^*; then $s : \mathbb{N} \to M^*$. By $(M^*)^{\mathbb{N}}$ we denote the set of discrete streams. By $\bar{s}$ we denote the finite or infinite discrete stream in $\mathbb{N} \to M$ that is the result of replacing

its time domain by that of non-timed streams while retaining the sequence of data elements. Consider as an example the stream $s : \mathbb{N} \to \{a, b, c\}^*$ where $s.n = \langle abc \rangle$. Then we get $\overline{s}.n = a$ if n **mod** $3 = 0$, $\overline{s}.n = b$ if n **mod** $3 = 1$ and $\overline{s}.n = c$ if n **mod** $3 = 2$. Seen as sequences we have:

$$s = \langle \langle abc \rangle \langle abc \rangle \langle abc \rangle \langle abc \rangle \ldots \rangle$$
$$\overline{s} = \langle abcabcabcabc \ldots \rangle$$

This corresponds to a time abstraction in which we forget all the timing information in the stream s and only keep the sequence of its elements.

Each discrete stream s contains a finite or infinite number $\#s$ of messages. For each discrete stream s we define a mapping

$$s\dagger : \{n \in \mathbb{N} : n < \#\overline{s}\} \to \mathbb{R}^+$$

that associates its time point with the i-th message in the discrete stream s.

Working with real numbers to represent time, the time points can be chosen more freely. Actually, we require strict monotonicity for the timing function of the time stamps since continuous time is the finest time granularity we can choose. The set of discrete streams over the message set M with continuous time is represented by the set

$$M^{\mathcal{R}}$$

Using real numbers for modelling time, we have to cope with Zeno's paradox. Given a stream s, we speak of Zeno's paradox if we have

$$\forall\, i \in \mathbf{dom}[s] : s\dagger i < t$$

for some time $t \in \mathbb{R}^+$, although $\#\overline{s} = \infty$. A simple example of a stream that exhibits Zeno's paradox is given in the following. Define the infinite stream s by the equations

$$s.i = i$$
$$s\dagger i = 1/2^i$$

Then the time function is strictly monotonic, and the stream is infinite, but its time points are bounded. In many applications such a behaviour is not of interest and should be excluded. We therefore require for any infinite stream s the proposition

$$\forall k \in \mathbb{N} : \exists i \in \mathbb{N} : s\dagger i > k$$

to avoid Zeno's paradox. A simple way to achieve this is to assume a minimal time distance $\delta \in \mathbb{R}$, $\delta > 0$, for all the messages in the timed stream s such that

$$s\dagger(i+1) - s\dagger i > \delta \quad \text{for all } i \text{ with } i, i+1 \in \mathbf{dom}[s].$$

The notation for streams with discrete time can easily be extended to streams with continuous time. For a stream s we denote for $t \in \mathbb{R}^+$ by

$$s{\downarrow}t$$

the stream of messages till time point t.

The crucial difference between discrete and continuous time is as follows. In the case of continuous time we have in contrast to discrete time:

- separability: we can always find a time point in between two given distinct time points; and
- limits: we can make our time intervals infinitely small leading to limit points.

Separability is certainly helpful since it supports the flexibility of the timing. Limits lead to Zeno's paradox and are better ruled out whenever possible.

A discrete stream is a sequence of messages such that we can speak about the first, second, third, and so on, message in a stream. Using continuous time a stream may contain uncountably many message elements. We speak of a *dense stream* in that case: a dense stream is represented by a function

$$s : \mathbb{R}^+ \to M$$

For every time $t \in \mathbb{R}^+$ we obtain a message $s(t) \in M$. By

$$M^{\mathbb{R}}$$

we denote the set of dense streams. We easily extend the notation $s{\downarrow}t$ to dense streams for $t \in \mathbb{R}$. $s{\downarrow}t$ is a finite stream obtained from s by restricting it to the time domain $]0 : t]$.

5.3 Components as functions on streams

In this section we introduce the general concept of a component as a function on timed streams. We consider the most general case of dense streams. (Since all other streams can be seen as special cases or abstractions of dense streams, these are included automatically.)

5.3.1 Behaviours of components

We work with channels as identifiers for streams. By C we denote the set of channels. Given a set of sorts T and a function

$$\text{sort} : C \to T$$

we speak of sorted channels. Given a set C of sorted channels we denote by

$$\vec{C}$$

the set of channel valuations

$$x : C \to M^{\mathbb{R}}$$

where $x.c$ is a timed stream of the appropriate sort $\text{sort}(c)$ of channel $c \in C$.

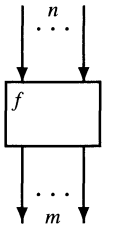


Figure 5.1. Graphical representation of a component as a dataflow node with n input and m output channels

A function

$$F : \vec{I} \to \mathbb{P}(\vec{O})$$

is called a *component behaviour*. F is called

- *timed* or *weakly causal*, if for all $t \in \mathbb{R}$ we have for all $x, z \in \vec{I}$:

$$x\downarrow t = z\downarrow t \Rightarrow F(x)\downarrow t = F(z)\downarrow t$$

- *time guarded by a finite delay* $\delta \in \mathbb{R}^+$, $\delta > 0$, or *causal*, if for all $t \in \mathbb{R}$ we have for all $x, z \in \vec{I}$:

$$x\downarrow t = z\downarrow t \Rightarrow F(x)\downarrow(t+\delta) = F(z)\downarrow(t+\delta)$$

A timed function has a proper time flow. That is, the choice of the output at the time point t does not depend on input that comes only after time t. Time guardedness models some delay in the reaction of a system, which introduces a fundamental notion of causality.

We use time-guarded stream-processing functions F to model the behaviour of a component. A graphical representation of the function F as a nondeterministic dataflow node is given in Fig. 5.1.

A behaviour F on discrete streams with discrete time is called *time-unbiased*, if for discrete input histories x and z we have

$$F.x = \{y : \exists x' \in \vec{I}, y' \in \vec{O} : y' \in F.x' \wedge \bar{y} = \overline{y'} \wedge \bar{x} = \overline{x'}\}$$

For time unbiased behaviours the timing of the messages in the input streams does not influence the messages in the output streams, but it may influence their timing.

5.4 Time abstraction

Refinement is the basic concept for the stepwise development of components. We describe only one form of refinement: *interaction refinement*. It is the basis of abstraction.

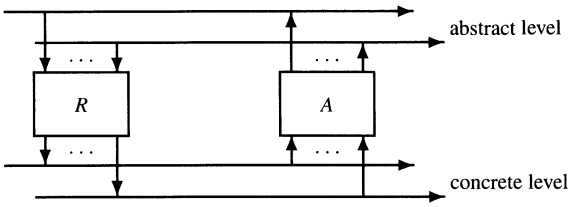


Figure 5.2. Communication History Refinement

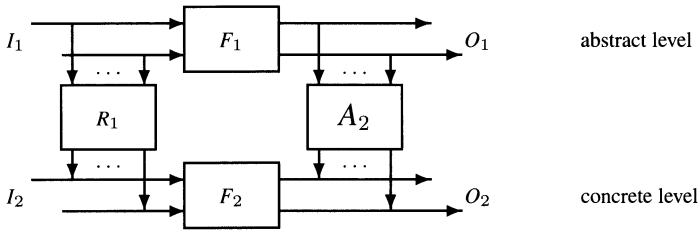


Figure 5.3. Commuting Diagram of Interaction Refinement (*U-simulation*)

5.4.1 General concepts of abstraction

By abstractions the syntactic interface of a component is changed. We work with pairs of abstraction and representation functions.

By interaction refinement we can change the number of input and output channels of a system, as well as the type and granularity of their messages, but still relate the behaviours in a formal way. A *communication-history refinement* requires timed functions

$$A : \vec{I} \to \mathbb{P}(\vec{O}) \qquad\qquad R : \vec{O} \to \mathbb{P}(\vec{I})$$

where

$$R \circ A = \mathrm{Id}$$

Here $R \circ A$ denotes the functial composition of R and A, defined by

$$(R \circ A).x = \{z \in A.y : y \in R.x\}$$

and Id denotes the identity relation

$$\mathrm{Id}\,.x = \{x\}$$

Fig. 5.2 shows the "commuting diagram" of history refinement.

Note that the requirement $R \circ A \subseteq \mathrm{Id}$ instead of $R \circ A = \mathrm{Id}$ is too weak, since this way we would allow abstract streams not to be represented at all.

Based on the idea of a history refinement we introduce the idea of an interaction refinement for components.

Given two communication history refinements

$$A_1 : \vec{I}_2 \to \mathbb{P}(\vec{I}_1) \qquad R_1 : \vec{I}_1 \to \mathbb{P}(\vec{I}_2)$$
$$A_2 : \vec{O}_2 \to \mathbb{P}(\vec{O}_1) \qquad R_2 : \vec{O}_1 \to \mathbb{P}(\vec{O}_2)$$

we call the behaviour

$$F_1 : \vec{I}_1 \to \mathbb{P}(\vec{O}_1)$$

an *interaction abstraction* of the behaviour

$$F_2 : \vec{I}_2 \to \mathbb{P}(\vec{O}_2)$$

if one of the following four propositions holds:

$$R_1 \circ F_2 \circ A_2 \subseteq F_1 \qquad \textit{U-simulation}$$
$$R_1 \circ F_2 \subseteq F_1 \circ R_2 \qquad \textit{downward simulation}$$
$$F_2 \circ A_2 \subseteq A_1 \circ F_1 \qquad \textit{upward simulation}$$
$$F_2 \subseteq A_1 \circ F_1 \circ R_2 \qquad U^{-1}\textit{-simulation}$$

Note that U^{-1}-simulation is the strongest condition, from which all others follow by straightforward algebraic manipulation.

5.4.2 *Abstractions from time*

In this section we study interaction abstractions that support abstractions from dense streams to streams with continuous time and further on to streams with discrete time and finally to non-timed streams.

From discrete timed to non-timed streams

A non-timed stream can be seen as the abstraction from all discrete timed streams with the same message set but arbitrary timing. The specification of the abstraction function A is simple:

$$A.y = \{\overline{y}\}$$

The specification of the representation function R is also simple:

$$R.x = \{y : \overline{y} = x\}\,.$$

Although the abstraction is so simple to specify, forgetting about time has serious consequences for functions that describe the behaviour of components. Time guardedness is lost and, as a consequence, we also lose the uniqueness of fixpoints — which has crucial impacts on the compositionality of the semantic models (for an extensive discussion, see [8]).

From continuous- to discrete Time

For relating discrete streams with discrete time to discrete streams with continuous time we work with an abstraction function

$$\alpha : M^{\mathcal{R}} \to (M^*)^{\mathbb{N}}$$

It is specified by the following equations (let $r \in M^{\mathcal{R}}$)

$$(\alpha.r).j = r.j$$
$$(\alpha.r)\dagger j = \min\{n \in \mathbb{N} : r\dagger j \leq n\}$$

We define the abstraction relation

$$A : M^{\mathcal{R}} \to \mathbb{P}(M^*)^{\mathbb{N}}$$

as

$$A.r = \{\alpha.r\}$$

and the representation specification

$$R : (M^*)^{\mathbb{N}} \to \mathbb{P}(M^{\mathcal{R}})$$

by the equation

$$R.s = \{r : s = \alpha.r\}$$

The step from discrete to continuous time (or back) is rather simple and does not have many consequences for the semantic techniques, as long as the time granularity chosen is fine enough to maintain time-guardedness.

From dense streams to discrete streams

To abstract a dense stream into a discrete stream we can use the following two techniques:

- sampling; and
- event discretisation.

In sampling, we select a countable number of time points as samples. We define the abstraction specification

$$A : M^{\mathbb{R}} \to \mathbb{P}(M^{\mathcal{R}})$$

that maps dense to discrete streams by

$$A.r = \{\alpha.r\}$$

where (choosing a simple variant of sampling in which we select the natural numbers as the sample time points)

$$(\alpha.r).j = r.j$$
$$(\alpha.r)\dagger j = j$$

A representation specification for *sampling* in continuous time is obtained by the function

$$R : M^{\mathcal{R}} \to \mathbb{P}(M^{\mathbb{R}})$$

defined by (we ignore for simplicity the possibility of successive identical messages in a stream)

$$R.s = \{r : s = \alpha.r\}$$

By sampling, continuous streams are related to discrete streams of events.

5.5 Conclusions

Modelling information processing systems appropriately is a matter of choosing the adequate abstractions in terms of the corresponding mathematical models. This applies for the models of time, in particular.

5.5.1 Time models in the literature

Time issues were always of great practical relevance for a number of applications of software systems such as embedded software and telecommunications. Nevertheless, in the scientific literature of mathematical models for software, time issues were considered only in the late seventies and then only in a few publications (see, for instance [20]). In the theoretical foundations of interactive systems, timing aspects were ignored in the beginning. It seems that the researchers tried hard to abstract from time, which was considered an operational notion. Early logical approaches were given in [12] and [6]. An early denotational model is found in [7].

Since then the interest in real time and its modelling has considerably increased in scientific research. In most of the approaches one particular time model is selected, without arguing much about the rationale of the particular choice. Often the time model is implicit (such as in statecharts, see [9], in SDL, see [19], or in Esterel [4]). This caused a lot of discussion about the right time model for such modelling languages. Other approaches such as the duration calculus (see [21]), where continuous time and dense message streams are essential, are explicitly directed towards time and a specific model of time.

Only a few publications discuss and compare different time models. One example is [10] which discusses the relation between the task of the specification and verification of real time programs and representations of time. Another example is [11] by Kopetz who compares what he calls dense time with what he calls sparse time. In sparse time events can only occur at "some sections of the time line". So sparse time seems to be what we call discrete time. A careful discussion of time models for hardware is found in [15], chapter 6. However, so far there is no approach that defines a formal relationship between systems working with different time models as we do with the concept of interaction abstraction.

Operational models of timing issues are found in [1], [2], [3], and [13, 14]. A specific issue is system models that incorporate aspects of continuous time and discrete events (see [2], [16]). In [17] and [18] the introduction of "hiatons", which are very similar to time ticks, are used to avoid problems with fixpoint theory.

5.5.2 *Concluding remarks*

Giving straightforward operational models that contain all the technical computational details of interactive nondeterministic computations is relatively simple. However, for systems-engineering purposes operational models are not very helpful. Finding appropriate abstractions for operational models of distributed systems is a difficult but nevertheless important task. Good abstract non-operational models are the basis for tractable system specifications and of a discipline of systems development.

Abstraction means forgetting information. Of course, we may forget only information that is not needed. Which information is needed does not only depend upon the explicit concept of observation, but also upon the considered forms of the composition of systems from subsystems.

As shown above, there are many ways to obtain time abstractions. Typical examples are

- from dense streams to discrete streams with continuous time,
- from discrete streams with continuous time to discrete time,
- from a finer discrete time to a coarser discrete time,
- from timed to non-timed streams.

In fact, all four abstraction steps mean that we use a coarser, more abstract time model. This way we lose some information about the timing of messages. As a consequence, messages at different time points may be represented by identical time points. This means we lose the principle of causality for certain input and output. This leads to intricate problems as we find them in the approaches that work with the assumption of so-called *perfect synchrony* (cf. [5]). The specification of interactive systems has to be done in a time/space frame. A specification should indicate which events (communication actions) can take place where, and when, and how they are causally related. Time information can be treated as any other information except, however, that the time flow follows certain laws. This is expressed by the timing requirements such as time guardedness. Such specification techniques are an important prerequisite for the development of safety-critical systems.

Acknowledgement

I am grateful to Ketil Stølen for a number of discussions that were helpful to clarify the basic concepts. It is a pleasure to thank my colleagues Olaf Müller and Jan Philipps for many useful remarks on a draft version of this paper.

References

[1] R. Alur, D. Dill. A theory of timed automata. *Theoretical Computer Science* 126, 1994, 183–235

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine. Algorithmic analysis of hybrid systems. *Theoretical Computer Science* 138, 1995, 3–34

[3] J.C.M. Baeten, J.A. Bergstra. Real Time Process Algebra. *Formal Aspects of Computing* 3, 1991, 142–188

[4] G. Berry, G. Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. INRIA, Research Report 842, 1988

[5] G. Berry. Preemption in Concurrent Systems. In: *Proceedings of the FSTTCS '93*, Lecture Notes in Computer Science 761, Springer Verlag 1993, 72–93

[6] A. Bernstein, P.K. Harter. Proving Real Time Properties of Programs with Temporal Logic. In: *Proceedings of the 8th Annual ACM Symposium on Operating Systems*, 1981, 1–11

[7] M. Broy. Applicative real time programming. In: *Information Processing 83, IFIP World Congress*, Paris 1983, North Holland 1983, 259–264

[8] M. Broy. Functional Specification of Time Sensitive Communicating Systems. *ACM Transactions on Software Engineering and Methodology* 2:1, January 1993, 1–46

[9] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 1987, 231–274

[10] M. Joseph. Problems, promises and performance: Some questions for real-time system specification. In: *Real Time: Theory in Practice, REX workshop*. Lecture Notes in Computer Science 600, 1991, 315–324

[11] H. Kopetz. Sparse Time versus Dense Time in Distributed Real-Time Systems. In: *Proceedings of the 12th International Conference on Distributed Computing Systems*. IEEE Computer Society Press 1992, 460–467,

[12] L. Lamport. TIMESETS: a new method for temporal reasoning about programs. In: D. Kozen (ed.): *Logics of Programs*. Lecture Notes in Computer Science 131, 1981, 177–196

[13] N. Lynch, F. Vaandrager. Action Transducers and Time Automata. *Formal Aspects of Computing* 8, 1996, 499–538

[14] N. Lynch, F. Vaandrager. Forward and Backward Simulations, Part II: Timing-Based Systems. *Information and Computation* 128:1, 1996

[15] T. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press. 1993

[16] O. Müller, P. Scholz. Functional Specification of Real-Time and Hybrid Systems. In: *HART'97, Proc. of the 1st Int. Workshop on Hybrid and Real-Time Systems*, Lecture Notes in Computer Science 1201, 1997, 273–286

[17] D. Park. On the Semantics of Fair Parallelism. In: D. Bjørner (ed.): *Abstract Software Specification*. Lecture Notes in Computer Science 86, Springer 1980, 504–526

[18] D. Park. The "Fairness" Problem and Nondeterministic Computing Networks. *Proc. 4th Foundations of Computer Science*, Mathematical Centre Tracts 159, Mathematisch Centrum Amsterdam, (1983) 133–161

[19] Specification and Description Language (SDL), Recommendation Z.100. Technical Report, CCITT, 1988

[20] N. Wirth. Towards a Discipline of Real Time Programming. *Communications of the ACM* 20:8, 1977, 577–583

[21] Zhou Chaochen, C.A.R. Hoare, A.P. Ravn. A Calculus of Durations. *Information Processing Letters* 40:5, 1991, 269–276

6

A predicative semantics for real-time refinement

Ian Hayes

Abstract

Real-time systems play an important role in many safety-critical systems. Hence it is essential to have a formal basis for the development of real-time software. In this chapter we present a predicative semantics for a real-time, wide-spectrum language. The semantics includes a special variable representing the current time, and uses timed traces to represent the values of external input and outputs over time so that reactive control systems can be handled. Because a real-time control system may be a nonterminating process, we allow the specification of nonterminating programs and the development of nonterminating repetitions. We present a set of refinement laws covering the constructs in the language. The laws make use of a relational style similar to that of Cliff Jones, although they have been generalised to handle nonterminating constructs.

6.1 Background

The sequential refinement calculus for non-real-time programs is a mature theory for the development of sequential programs [1, 2, 19, 20]. Our goal is to develop an equivalent theory for real-time programs. Work by Mahony modelled real-time systems by representing the observable variables as timed traces: functions from time (real numbers) to their type [17, 18]. That work concentrated on modeling system components over all time, and on decomposing systems into parallel combinations of such components, and had a semantics based on predicate transformers [16]. Mark Utting and Colin Fidge used a related approach to develop a sequential real-time refinement calculus that was also based on timed traces and predicate transformers [22, 23]. In that work, in common with a number of other approaches to real-time [21, 13], an execution time is associated with each component of a command. This leads to complex timing conditions as well as overly

A. McIver et al. (eds.), *Programming Methodology*

restricted timing constraints on the execution of individual commands and their components.

A breakthrough came with the introduction of the *deadline* command [9, 3]. The deadline command has a simple semantics: it takes no time to execute and guarantees to complete by a given time. For example, the following code reads the value of the input d_1 into the local variable x, calculates $f(x)$ and assigns it to y, writes y to the output d_2. The special variable τ stands for the current time. The starting time of the commands is captured in the auxiliary variable m, and the final command is a deadline of $m + U$; this ensures that the commands complete within U time units of their beginning.

$$
\begin{array}{l}
m := \tau; \quad -- \ \tau \text{ is the current time variable} \\
x : \mathbf{read}(d_1); \\
y := f(x); \\
d_2 : \mathbf{write}(y); \\
\mathbf{deadline}\, m + U
\end{array}
\tag{1}
$$

In isolation a deadline command cannot be implemented, but if it can be shown that all execution paths leading to a deadline command reach it before its deadline, then it can be removed. The deadline command allows *machine-independent* real-time programs to be expressed. It also allows one to separate out timing constraints to leave components that are purely calculations [5]; these components can then be developed as in the non-real-time calculus.

The semantics used in the earlier work was based on that of Utting and Fidge [22, 23]. The current time variable, τ, was treated in the same manner as in the standard refinement calculus with a before and after value for each command, but all other variables were treated as functions of time (real numbers), which were constrained by the execution of a command [10].

As real-time systems often use processes which are potentially nonterminating, we desired to extend the approach to handle these. At the specification level this was quite easy: the current time variable, τ, was allowed to take on the value infinity to indicate nontermination. However, the earlier semantics was based on weakest-precondition predicate transformers and hence dealt only with terminating commands and only allowed the development of terminating repetitions. As in the standard refinement calculus, nonterminating repetitions were identified with abort. Hence that semantics was unsuitable.

For the new semantics, the primary influences are the work of Hehner [11], and Hoare and He [12] using predicative semantics for program development. These were first used to tackle the semantics of nonterminating repetitions [7]. Hooman's work on real-time Hoare logic [13] also allows nonterminating repetitions and was influential in the approach taken to the laws for introducing nonterminating repetitions. Auxiliary variables and procedure parameters were also added to facilitate the expression of timing constraints [6].

The final influence on this paper is the work of Jones [15] on a relational approach to proof rules for sequential programs. This paper brings together the above pieces of work to give a relational, predicative semantics for a sequen-

tial real-time refinement calculus that supports nonterminating processes and auxiliary variables.

6.1.1 Related work

Hooman and Van Roosmalen [14] have developed a platform-independent approach to real-time software development similar to ours. Their approach makes use of timing annotations that are associated with commands. The annotations allow the capture in auxiliary timing variables of the time of occurrence of significant events that occur with the associated command, and the expression of timing deadlines on the command relative to such timing variables. They give an example similar to (1) above, using their notation:

$$in(d_1, x)[m?];$$
$$y := f(x);$$
$$out(d_2, y)[< m + U]$$

The constructs in square brackets are timing annotations [14, Sect. 2]. On the input the annotation '$m?$' indicates that the time at which the input occurs should be assigned to timing variable m, and on the output the annotation '$< m + U$' requires the output to take effect before $m + U$, i.e. within U time units of the input time. Hooman and Van Roosmalen keep timing annotations separate from the rest of the program. They give Hoare-like rules for reasoning about programs in their notation, but there is no semantics against which to justify the rules. The approach to real-time semantics given in this paper could be used to justify their Hoare axioms.

Section 6.2 introduces the machine-independent, wide-spectrum language and gives its semantics, along with suitable refinement rules. Section 6.3 presents an example refinement that makes use of the refinement laws, Section 6.4 discusses repetitions, and Section 6.5 discusses timing constraint analysis.

6.2 Language and semantics

We model time by nonnegative real numbers:

$$Time \mathrel{\widehat{=}} \{r : \mathbf{real}_\infty \mid 0 \leq r < \infty\}.$$

where $\mathbf{real}_\infty$ stands for the real numbers extended with plus and minus infinity, and real operators such as '$<$' are extended to work with infinite arguments. The real-time refinement calculus makes use of a special real-valued variable, τ, for the current time. To allow for nonterminating programs, we allow τ to take on the value infinity (∞):

$$Time_\infty \mathrel{\widehat{=}} Time \cup \{\infty\}.$$

We refer to the set of variables in scope as the environment, and use the name ρ for the environment. In real-time programs we distinguish four kinds of variables:

- inputs, $\rho.in$, which are under external control;
- outputs, $\rho.out$, which are under the control of the program;
- local variables, $\rho.local$, which are under the control of the program, but unlike outputs are not externally visible; and
- auxiliary variables, $\rho.aux$, which are similar to local variables, but are restricted to appear only in assumptions, specifications, deadline commands and assignments to auxiliary variables.

To simplify the presentation in this paper, we only treat the types of variables informally.

Inputs and outputs are modelled as timed traces: functions from *Time* to the declared type of the variable. This allows one to model both continuous and discrete inputs within the same framework. As a running example, we use the simple real-time task of closing a railway gate when a train is detected as being near, and reopening the gate when the train is out of the danger region. The example is treated in more detail in Sec. 6.3. The controller reads from the external inputs *near* and *out*, and writes to the output *gate*.

input *near*, *out* : *boolean*;
output *gate* : {*open*, *close*};

The inputs *near* and *out* are modelled as functions from *Time* to boolean, and the output *gate* is modelled as a function from *Time* to $\{open, close\}$. For t in *Time* (which does not include infinity), the expression $near(t)$ gives the value of *near* at time t.

The primitive commands in our language only constrain an output over the execution interval of the command, which is the left-open, right-closed interval from the start time, τ_0, to the finish time of the command, τ, which we write as $(\tau_0 \,..\, \tau]$. The initial value of the output at τ_0 is determined by the previous command, and then the command determines the values up to and including τ. Any programs composed of these commands using the standard structures like sequential composition, selection and repetition also satisfy this property.

In earlier work [22, 10] all variables, except τ, were modelled as functions of time (timed traces). With the addition of auxiliary variables [6] this is not possible, because assignments to auxiliary variables take no time and a timed trace only allows a variable to have a single value at any one time. Hence to represent the effect of a command on an auxiliary variable, z, we use a relation between its initial value (represented in predicates by z_0) and its final value (represented in predicates by z). Having introduced this model for auxiliary variables, we decided to use the same model for local variables. Either model could be used for local variables, but choosing a similar model for auxiliary and local variables makes the semantics a little simpler. In addition, this model is more abstract because it does not consider intermediate values of local variables during the execution of a command. We refer to the combination of local and auxiliary variables as the

state, and use the abbreviation $\rho.v$ to stand for the state variables, and decorations of $\rho.v$, such as $\rho.v_0$, to stand for the decorated state variables.

For the railway crossing example, we declare a local boolean variable, *sens*, and an auxiliary time-valued variable, *before*, as follows.

var *sens* : *boolean*;
aux *before* : *Time*;

We represent the semantics of a command by a predicate in a form similar to that of Hehner [11], and Hoare and He [12]. The predicate is in terms of the input and output traces over time, the initial and final values of the state variables, and the initial and final values of the current time, τ_0 and τ. The meaning function, $\mathcal{M}$, takes the variables in scope, ρ, and a command C and returns the corresponding predicate, $\mathcal{M}_\rho(C)$. Refinement of commands (in an environment, ρ) is defined as reverse entailment:

$$C \sqsubseteq_\rho D \mathrel{\widehat{=}} (\mathcal{M}_\rho(C) \Leftarrow \mathcal{M}_\rho(D)) \ ,$$

where '$P \Leftarrow Q$' holds if for all values of the variables, whenever Q holds, P holds. We use the relation '$\sqsupseteq_\rho$' for refinement equivalence, i.e. refinement in both directions. When the environment is clear from the context the subscript ρ may be omitted.

6.2.1 Real-time specification command

We introduce a possibly nonterminating real-time *specification command*,

$$\infty \vec{x}: [P, \ Q] \ ,$$

where $\vec{x}$ is a vector of variables (called the *frame*) that may be modified by the command, the predicate P is the assumption made by the specification, and the predicate Q is its effect. The '∞' at the beginning is just part of the syntax; it reminds us that the command might not terminate. The assumption P is assumed to hold at the start time of the command. It is a *single-state predicate*. That is, it may reference any of the variables in the environment plus τ, but it may not reference τ_0 or initial state (zero-subscripted) variables. The effect predicate Q describes a relation between before and after state variables in the environment, and τ_0 and τ, as well as a constraint on the values of the outputs. To simplify the presentation in this paper, we refer to such predicates as *relations*. We also assume that all predicates and relations are well formed with respect to the relevant environment in the context in which they are used.

We define a *terminating* specification command similarly. The only difference is the additional requirement that the effect should achieve $\tau < \infty$.

$$\vec{x}: [P, \ Q] \mathrel{\widehat{=}} \infty \vec{x}: [P, \ Q \wedge \tau < \infty]$$

This is the real-time equivalent of the Morgan specification command, which is guaranteed to terminate [19]. Below we state laws for the more general, possibly nonterminating, specification command, but special cases for a terminating

specification command are easily derived, and we make use of those in the examples.

For example, in the following specification the frame consists of the local variable, *sens*, and the time-valued, auxiliary variable, *before*. The notation *near* $\uparrow 0$ stands for the time at which the input *near* makes its first transition from *false* to *true*, or it is infinity if there is no such transition. If *near* does make a transition, then the following specification terminates at some time after that transition. However, if *near* never makes a transition, the specification never terminates.

$$\infty\, sens, before\colon \left[S,\ near \uparrow 0 \leq \tau < \infty \vee near \uparrow 0 = \tau = \infty\right]$$

The assumption S (which is needed for the refinement of this specification) will be explained further below.

The frame of a specification command lists those variables that may be modified by the command. The frame may not include inputs. The current time variable, τ, is implicitly in the frame. All outputs not in the frame are defined to be stable for the duration of the command, provided the assumption holds initially. We define the predicate *stable* by

$$stable(z, TS) \mathrel{\widehat{=}} TS \neq \{\} \Rightarrow (\exists x \bullet z(\!|TS|\!) = \{x\})$$

where $z(\!|TS|\!)$ is the image of the set (of times) TS through the function z (representing an external variable). We allow the first argument of stable to be a set (or vector) of variables, in which case all variables in the set are stable. To specify the closed interval of times from s until t, we use the notation $[\!-s \ldots t-\!]$. The open interval is specified by $(\!-s \ldots t-\!)$.

Any state variable, y, not in the frame is unchanged. Hence for these variables we require that $y_0 = y$, except that in the case of a nonterminating command there is no final state and hence the equality is not required if the final time is infinity. For a vector of outputs, $\vec{out}$, a vector of state variables, $\vec{z}$, and times t_0 and t, we introduce the following notation.

$$eq(\vec{out}, t_0, t, \vec{z}_0, \vec{z}) \mathrel{\widehat{=}} stable(\vec{out}, [\!-t_0 \ldots t-\!]) \wedge (t < \infty \Rightarrow \vec{z}_0 = \vec{z})$$

Definition 6.2.1 (real-time specification) *Given an environment, ρ, a specification command, $\infty \vec{x}\colon [P,\ Q]$, is* well-formed *provided its frame, $\vec{x}$, is contained in $\rho.local \cup \rho.aux \cup \rho.out$, P is a single-state predicate, and Q is a relation. The meaning of a possibly nonterminating* real-time specification *command is defined by the following,*

$$\mathcal{M}_\rho\left(\infty \vec{x}\colon [P,\ Q]\right) \mathrel{\widehat{=}} \tau_0 \leq \tau \wedge$$
$$(\tau_0 < \infty \wedge P\left[\begin{smallmatrix}\rho.v_0, \tau_0 \\ \rho.v, \tau\end{smallmatrix}\right] \Rightarrow (Q \wedge eq(\rho.out \setminus \vec{x}, \tau_0, \tau, \rho.v_0 \setminus \vec{x}_0, \rho.v \setminus \vec{x})))$$

where the operator '$\setminus$' is set difference. □

As abbreviations, if P is omitted, then it is taken to be *true*, and if the frame is empty the ':' is omitted. Note that if P does not hold initially the command still guarantees that time does not go backwards.

Because τ may take on the value infinity, the above specification command allows nontermination. If the command does not terminate then the final values of the state variables have no counterpart in reality. Hence it does not make sense to write specifications that require, for example, the final value of a local variable y to be zero and the command to not terminate: $y = 0 \wedge \tau = \infty$. There is no program code that can implement such a specification, so they are of little use. The following property states the condition under which an effect relation (a predicate) is independent of the final values of the state variables if the command does not terminate.

Definition 6.2.2 (nontermination state independent) *Given a relation, Q, that is well-formed in an environment, ρ, Q is* nontermination state independent *provided*

$$\tau = \infty \Rrightarrow (Q \Leftrightarrow (\exists \rho.v \bullet Q)) \quad \square$$

A command, C, is nontermination state independent if its meaning predicate, $\mathcal{M}_\rho(C)$, is nontermination state independent. In the definition of the specification command the equality within *eq* between the initial and final values of state variables that are not in the frame does not apply if the command does not terminate. All the primitive real-time commands defined in Sec. 6.2.2 satisfy this property, and compound commands preserve it. Hence the only commands that may not satisfy it are specification commands, because the effect Q constrains the final values of the state variables at time infinity. We require all specifications to satisfy this healthiness property.

The law for weakening an assumption is similar to that for the standard refinement calculus.

Law 6.2.3 (weaken assumption) *Provided $P \Rrightarrow P'$,*

$$\infty \vec{x}: [P,\ Q] \sqsubseteq \infty \vec{x}: [P',\ Q] \quad \square$$

A common refinement step is to strengthen the effect of a specification command. In the real-time case one can take into account the following: time cannot go backwards; if the start time of the command is infinity (i.e., the finishing time of the previous command was infinity) it is never executed so its effect is irrelevant; the assumption holds for the initial state of the variables; any outputs not in the frame are stable; and any state variables not in the frame are unchanged.

Law 6.2.4 (strengthen effect) *Provided*

$$\begin{array}{l} \tau_0 \leq \tau \wedge \tau_0 < \infty \wedge P\left[\frac{\rho.v_0, \tau_0}{\rho.v, \tau}\right] \wedge \\ eq(\rho.out \setminus \vec{x}, \tau_0, \tau, \rho.v_0 \setminus \vec{x}_0, \rho.v \setminus \vec{x}) \wedge Q' \\ \Rrightarrow Q \end{array}$$

then $\infty \vec{x}: [P,\ Q] \sqsubseteq \infty \vec{x}: [P,\ Q']$. $\square$

For a time interval I, and a predicate P, that contains unindexed occurrences of external inputs and outputs, the notation P **on** I stands for $(\forall t : I \bullet P @ t)$

where $P @ t$ stands for the predicate P with all occurrences of each external input or output, e, replaced by $e(t)$. For example, the notation

$$(gate = open) \textbf{ on } [\![\tau_0 \ldots near \uparrow 0]\!]$$

stands for

$$(\forall t : [\![\tau_0 \ldots near \uparrow 0]\!] \bullet gate(t) = open) \ .$$

The following is an example of strengthening the effect of a specification. The predicate *SENS* will be explained later.

$$gate: \left[\begin{array}{l} gate(\tau) = open \wedge \\ \tau \leq near \uparrow 0 \wedge \\ SENS \end{array}\right. , \left.\begin{array}{l} (gate = open) \textbf{ on } [\![\tau_0 \ldots near \uparrow 0]\!] \wedge \\ near \uparrow 0 \leq \tau \wedge SENS \end{array}\right] \tag{2}$$

$\sqsubseteq$ Law 6.2.4 (strengthen effect); Law 6.2.3 (weaken assumption)

$$gate: \left[\tau \leq near \uparrow 0 \wedge SENS, \begin{array}{l} stable(gate, [\![\tau_0 \ldots \tau]\!]) \wedge \\ near \uparrow 0 \leq \tau \end{array}\right] \tag{3}$$

The strengthening of the effect is valid provided the following condition holds. The occurrences of the conjunct $\tau < \infty$ come from the fact that the specifications are terminating.

$$\begin{array}{l} \tau_0 \leq \tau \wedge \tau_0 < \infty \wedge \\ gate(\tau_0) = open \wedge \tau_0 \leq near \uparrow 0 \wedge SENS\left[\frac{\tau_0}{\tau}\right] \wedge \\ stable(gate, [\![\tau_0 \ldots \tau]\!]) \wedge near \uparrow 0 \leq \tau \wedge \tau < \infty \\ \Rrightarrow (gate = open) \textbf{ on } [\![\tau_0 \ldots near \uparrow 0]\!] \wedge near \uparrow 0 \leq \tau \wedge SENS \wedge \tau < \infty \end{array}$$

Ignoring the occurrences of *SENS* (which is defined later) the remainder holds because the gate is initially open at time τ_0 and stable until time τ, which is after time $near \uparrow 0$.

A state variable can always be removed from the frame. This effectively strengthens the post-condition to ensure that the variable is unchanged.

Law 6.2.5 (contract frame) *For a state variable z, not in $\vec{x}$,*

$$\infty z, \vec{x}: [P, \ Q] \sqsubseteq \infty \vec{x}: [P, \ Q] \quad \square$$

If an output is to be stable for the whole of the execution time of a command, it can be removed from the frame.

Law 6.2.6 (output stable) *For an output o, not in $\vec{x}$,*

$$\infty o, \vec{x}: [P, \ Q \wedge stable(o, [\![\tau_0 \ldots \tau]\!])] \sqsupseteq \infty \vec{x}: [P, \ Q] \quad \square$$

For example, the following holds the *gate* open by keeping it stable (i.e., not changing it) over the required interval.

$$gate: \left[\tau \leq near \uparrow 0 \wedge SENS, \begin{array}{l} stable(gate, [\![\tau_0 \ldots \tau]\!]) \wedge \\ near \uparrow 0 \leq \tau \end{array}\right] \tag{3}$$

$\sqsubseteq$ Law 6.2.6 (output stable) for *gate*

$$[\tau \leq near \uparrow 0 \wedge SENS, \ near \uparrow 0 \leq \tau] \tag{4}$$

Let $\vec{x}$ be a vector of variables, not including any inputs; $\vec{E}$ be a vector of idle-stable expressions of the same length as $\vec{x}$ and assignment compatible with $\vec{x}$; D be a time-valued expression; z be a local variable; i be an input that is assignment compatible with z; o be an output; and E be an idle-stable expression that is assignment compatible with o.

$$\mathbf{skip} \mathrel{\widehat{=}} [\tau_0 = \tau] \tag{5}$$

$$\mathbf{idle} \mathrel{\widehat{=}} [\tau_0 \leq \tau] \tag{6}$$

$$\vec{x} := \vec{E} \mathrel{\widehat{=}} \vec{x}: \left[\vec{x} = (\vec{E}\left[\tfrac{\vec{x}_0}{\vec{x}}\right]) @ \tau_0\right], \text{-- } \vec{x} \text{ only locals} \tag{7}$$

$$\vec{x} := \vec{E} \mathrel{\widehat{=}} \vec{x}: \begin{bmatrix} \tau_0 = \tau \wedge \\ \vec{x} = (\vec{E}\left[\tfrac{\vec{x}_0}{\vec{x}}\right]) @ \tau_0 \end{bmatrix}, \text{ -- } \vec{x} \text{ only auxiliaries} \tag{8}$$

$$\mathbf{deadline}\, D \mathrel{\widehat{=}} [\tau_0 = \tau \leq D @ \tau] \tag{9}$$

$$z : \mathbf{read}(i) \mathrel{\widehat{=}} z: [z \in i(\!|\tau_0 \ldots \tau|\!)] \tag{10}$$

$$o : \mathbf{write}(E) \mathrel{\widehat{=}} o: [o(\tau) = E @ \tau_0] \tag{11}$$

Figure 6.1. Definition of primitive real-time commands

All our commands insist that time does not go backwards.

Law 6.2.7 (time progresses) *For any command, C, that is well-formed in an environment, ρ, the following holds:* $\mathcal{M}_\rho(C) \Rrightarrow \tau_0 \leq \tau$. □

6.2.2 *Primitive real-time commands*

The primitive real-time commands can be defined in terms of equivalent specification commands. In Fig. 6.1 we define: the null command, **skip**, that does nothing and takes no time; a command, **idle**, that does nothing but may take time; multiple assignment commands for both local and auxiliary variables; the deadline command; a command, **read**, to sample a value from an external input; and a command, **write**, to output a value to an external output, o.

We allow expressions used in programs, e.g. in assignments and guards, to refer to external variables without explicit time indices. When these expressions are used within predicates within the equivalent specification commands, all references to external variables need to be explicitly indexed. Hence we use the notation $E @ t$ to refer to the expression E with all occurrences of any external variable e replaced by $e(t)$, and all occurrences of τ replaced by t.

Because an expression takes time to evaluate, we require that its value does not change over the interval during which it is being evaluated. We refer to such expressions as being *idle-stable*, that is, their value does not change over time provided all the variables under the control of the program are stable. In practice this means that such expressions cannot refer to τ or to the value of external inputs.

Definition 6.2.8 (idle-stable) *Given an environment ρ, an expression E is* idle-stable *provided,*

$$\tau_0 \leq \tau < \infty \wedge \mathit{stable}(\rho.\mathit{out}, \llbracket \tau_0 \ldots \tau \rrbracket) \Rrightarrow E \mathbin{@} \tau_0 = E \mathbin{@} \tau \quad \square$$

The deadline command guarantees to meet its deadline, even if the deadline time has already passed. If the deadline has already passed, the effect of the deadline command is false, which means that the command is miraculous and cannot possibly be implemented.

6.2.3 *Sequential composition*

Because we allow nonterminating commands, we need to be careful with our definition of sequential composition. If the first command of the sequential composition does not terminate, then we want the effect of the sequential composition on the values of the outputs over time to be the same as the effect of the first command. This is achieved by ensuring that for any command in our language, if it is 'executed' at $\tau_0 = \infty$, it has no effect.

Law 6.2.9 (nontermination preserved) *For any command, C, that is well-formed in an environment, ρ, the following holds: $\tau_0 = \infty \Rrightarrow (\mathcal{M}_\rho(C) \Leftrightarrow \tau = \infty)$.* $\square$

For the specification command this is achieved by the assumption $\tau_0 < \infty$ in Def. 6.2.1 (real-time specification).

The definition of sequential composition combines the effects of the two commands via a hidden intermediate state ($\rho.v'$ in the definition below). First we introduce a forward relational composition operator, '$\fatsemi$'.

Definition 6.2.10 (relational composition) *Given an environment ρ and two relations R_1 and R_2 the (forward)* relational composition *of R_1 and R_2 is defined as follows,*

$$R_1 \fatsemi R_2 \mathrel{\widehat{=}} \exists \tau' : \mathit{Time}_\infty; \rho.v' : T_v \bullet R_1 \left[\begin{smallmatrix} \tau', \rho.v' \\ \tau, \rho.v \end{smallmatrix}\right] \wedge R_2 \left[\begin{smallmatrix} \tau', \rho.v' \\ \tau_0, \rho.v_0 \end{smallmatrix}\right] \quad .$$

where T_v is the type of $\rho.v'$. $\square$

Definition 6.2.11 (sequential composition) *Given an environment ρ, and real-time commands C and D, their* sequential composition *is defined as the relational composition of their meaning predicates.*

$$\mathcal{M}_\rho(C; D) \mathrel{\widehat{=}} \mathcal{M}_\rho(C) \fatsemi \mathcal{M}_\rho(D) \quad . \quad \square$$

Because both C and D guarantee $\tau_0 \leq \tau$, their sequential composition does also. Note that even if the assumption of the second command does not hold, the sequential composition still guarantees the effect of the first command for the external variables. It also guarantees that the finish time is greater than or equal to the finish time of the first command.

The following law is a generalisation of the standard law for refining a specification to a sequential composition of specifications. For the termination case both commands must terminate. The first establishes the intermediate single-state predicate P_1 as well as the relation R_1 between the start and finish states of the first command. The second command assumes P_1 initially and establishes the single-state predicate P_2 as well as the relation R_2 between its initial and final states. Hence the sequential composition establishes P_2 as well as the relational composition of R_1 and R_2 between its initial and final states.

For the nontermination case either the first command does not terminate and establishes Q_1, or the first command terminates establishing P_1 and R_1 and the second command does not terminate and establishes Q_2. The overall effect is thus either Q_1 or the composition of R_1 and Q_2.

Law 6.2.12 (sequential composition) *Given single-state predicates P_0, P_1 and P_2, and relations R_1, R_2, Q_1 and Q_2,*

$$\begin{array}{l} \infty\vec{x}\colon \left[P_0,\ (\tau < \infty \wedge P_2 \wedge (R_1 \fatsemi R_2)) \vee (\tau = \infty \wedge (Q_1 \vee (R_1 \fatsemi Q_2)))\right] \\ \sqsubseteq \\ \infty\vec{x}\colon \left[P_0,\ (\tau < \infty \wedge P_1 \wedge R_1) \vee (\tau = \infty \wedge Q_1)\right] ; \\ \infty\vec{x}\colon \left[P_1,\ (\tau < \infty \wedge P_2 \wedge R_2) \vee (\tau = \infty \wedge Q_2)\right] \quad \Box \end{array}$$

Taking Q_1 and Q_2 as *false* reduces the law back to the standard law of Jones [15] for terminating commands:

$$\vec{x}\colon \left[P_0,\ P_2 \wedge (R_1 \fatsemi R_2)\right] \sqsubseteq \vec{x}\colon \left[P_0,\ P_1 \wedge R_1\right] ; \vec{x}\colon \left[P_1,\ P_2 \wedge R_2\right] \ .$$

For example, if we instantiate the above law with P_0 the predicate S, P_1 the predicate *true*, P_2 the predicate $\tau \leq \mathit{near} \uparrow 0 + \mathit{err}$, R_1 the relation $\tau_0 = \tau = \mathit{before}$, R_2 the relation $\mathit{sens} \in \mathit{near}(\!|\tau_0 \ldots \mathit{near} \uparrow 0 + \mathit{err}|\!)$, and Q_1 and Q_2 both *false*, then because $R_1 \fatsemi R_2$ is the following

$$\begin{array}{l} (\exists \tau' : \mathit{Time}; \mathit{sens}' : \mathit{boolean}; \mathit{before}' : \mathit{Time} \bullet \tau_0 = \tau' = \mathit{before} \wedge \\ \qquad \mathit{sens} \in \mathit{near}(\!|\tau' \ldots \mathit{near} \uparrow 0 + \mathit{err}|\!)) \\ \equiv \tau_0 = \mathit{before} \wedge \mathit{sens} \in \mathit{near}(\!|\tau_0 \ldots \mathit{near} \uparrow 0 + \mathit{err}|\!) \end{array}$$

we can derive the following refinement.

$$\mathit{sens}, \mathit{before}\colon \left[S,\ \begin{array}{l} \mathit{before} = \tau_0 \wedge \tau \leq \mathit{near} \uparrow 0 + \mathit{err} \wedge \\ \mathit{sens} \in \mathit{near}(\!|\tau_0 \ldots \mathit{near} \uparrow 0 + \mathit{err}|\!) \end{array}\right] \tag{12}$$

$\sqsubseteq$ Law 6.2.12 (sequential composition)

$$\mathit{sens}, \mathit{before}\colon \left[S,\ \tau_0 = \tau = \mathit{before}\right] ; \tag{13}$$

$$\mathit{sens}, \mathit{before}\colon \left[\mathit{true},\ \begin{array}{l} \tau \leq \mathit{near} \uparrow 0 + \mathit{err} \wedge \\ \mathit{sens} \in \mathit{near}(\!|\tau_0 \ldots \mathit{near} \uparrow 0 + \mathit{err}|\!) \end{array}\right] \tag{14}$$

Specification (13) can be refined as follows.

(13) $\sqsubseteq$ Law 6.2.5 (contract frame) by *sens*; Law 6.2.3 (weaken assumption)

$$\mathit{before}\colon \left[\mathit{before} = \tau \wedge \tau = \tau_0\right]$$

$\sqsubseteq$ Def. 8 (auxiliary assignment)

$before := \tau$

A deadline command can be used to ensure that a command completes by a given time. The following law can be proved using Law 6.2.12 (sequential composition), with the deadline command given in its specification command equivalent (9).

Law 6.2.13 (separate deadline) *Provided D does not refer to initial variables,*

$$\vec{x}\colon \left[P,\ Q \wedge \tau \leq D\right] \sqsubseteq \vec{x}\colon \left[P,\ Q\right] ;\ \mathbf{deadline}\, D \quad \square$$

For example, the specification (14) can be refined as follows.

(14) $\sqsubseteq$ Law 6.2.5 (contract frame) by *before*; Law 6.2.4 (strengthen effect)

$sens\colon \left[sens \in near(\!|\tau_0 \ldots \tau|\!) \wedge \tau \leq near \uparrow 0 + err\right]$

$\sqsubseteq$ Law 6.2.13 (separate deadline)

$$sens\colon \left[sens \in near(\!|\tau_0 \ldots \tau|\!)\right] ; \qquad (15)$$
$$\mathbf{deadline}\, near \uparrow 0 + err$$

The specification (15) is equivalent to $sens : \mathbf{read}(near)$.

Commonly a specification is refined to a sequence of more than two specifications. The following law follows by multiple application of Law 6.2.12 (sequential composition) for the terminating case. A more complex version for nonterminating commands can also be devised.

Law 6.2.14 (multiple sequential compositions) *Given single-state predicates $P_0, P_1, \ldots, P_n$, and relations $R_1, R_2, \ldots, R_n$, where $n \geq 1$, then*

$$\vec{x}\colon \left[P_0,\ P_n \wedge (R_1 \fatsemi R_2 \fatsemi \cdots \fatsemi R_n)\right]$$
$$\sqsubseteq$$
$$\begin{array}{l} \vec{x}\colon \left[P_0,\ P_1 \wedge R_1\right] ; \\ \vec{x}\colon \left[P_1,\ P_2 \wedge R_2\right] ; \\ \vdots \\ \vec{x}\colon \left[P_{n-1},\ P_n \wedge R_n\right] \end{array} \quad \square$$

6.2.4 *Nondeterministic choice, guards and selection*

The selection (**if**) command is defined in terms of sequential composition and (nondeterministic) choice. We first define choice ($[\!]$).

Definition 6.2.15 (choice) *Given an environment, ρ, and real-time commands, C and C', the* nondeterministic choice *between C and C' is defined by the following.*

$$\mathcal{M}_\rho\,(C \,[\!]\, C') \mathrel{\widehat{=}} \mathcal{M}_\rho\,(C) \vee \mathcal{M}_\rho\,(C') \quad \square$$

Nondeterministic choice is symmetric, associative and idempotent.

For a selection command we model evaluation of a guard B by $\lceil B @ \tau \rceil$, i.e., a specification command with an empty frame. The guard may take time to evaluate (note that τ is implicitly in the frame of any specification command, including guards). A guard is only feasible if the guard B evaluates to true when the command is reached. A selection assumes that one of its guards holds, and hence that one of the guards is feasible. The guard expressions are required to be idle-stable so that their values do not change while they are being evaluated. The final **idle** command allows for the time taken to exit the selection.

Definition 6.2.16 (selection) *Given a set of real-time commands, $C_1, \ldots, C_n$, and idle-stable, boolean-valued expressions, $B_1, \ldots, B_n$, a* selection *command is defined as follows.*

$$\begin{array}{l}\textbf{if } B_1 \rightarrow C_1 \,[\!]\, \cdots \,[\!]\, B_n \rightarrow C_n \textbf{ fi} \mathrel{\widehat{=}} \\ (\lceil BB,\ \ B_1 @ \tau \rceil ; C_1 \,[\!]\, \cdots \,[\!]\, \lceil BB,\ \ B_n @ \tau \rceil ; C_n); \textbf{idle}\end{array}$$

where $BB \mathrel{\widehat{=}} B_1 @ \tau \vee \cdots \vee B_n @ \tau$. □

The definition of a selection puts no bounds on the time to evaluate the guards or the time to exit the selection. It is expected that deadline commands, either within branches of the selection or following the selection, will indirectly introduce time bounds on these activities. Evaluation of the guards of a selection command takes time. Hence if some assumption P holds before guard evaluation, P may no longer hold after guard evaluation. Even though none of the variables under the control of the program are modified during guard evaluation, P may refer to the current time τ or to external inputs, both of which may change during the time taken for guard evaluation. To avoid this problem we restrict our attention to assumptions that are invariant over the execution of an **idle** command. Such assumptions are referred to as being *idle-invariant*.

Definition 6.2.17 (idle-invariant) *A single-state predicate P is* idle-invariant *provided,*

$$\tau_0 \leq \tau < \infty \wedge \mathit{stable}(\rho.\mathit{out}, \llbracket \tau_0 \ldots \tau \rrbracket) \wedge P\left[\tfrac{\tau_0}{\tau}\right] \Rrightarrow P. \quad \square$$

Note that predicates of the form $\tau \leq D$ (where D is idle-stable) are not idle-invariant, but predicates of the form $D \leq \tau$ are. If the only references to τ in P are as indices of outputs, then P is idle-invariant.

Similarly, the effect of a specification command being refined to a selection is required to be impervious to the time taken to evaluate the guards and to exit the selection. We refer to it as being both *pre-idle-invariant* and *post-idle-invariant*. A relation R is pre-idle-invariant if prefixing it with an idle period has no effect. That is, whenever it holds over an interval from τ_0 to τ, then for any u less than or equal to τ_0 it holds over the interval from u to τ, provided the variables under the control of the program are not modified over the interval from u to τ_0.

Definition 6.2.18 (pre-idle-invariant) *A relation R is* pre-idle-invariant *provided,*

$$\tau_0 < \infty \wedge u \leq \tau_0 \leq \tau \wedge \mathit{stable}(\rho.\mathit{out}, \lbrack\!\lbrack u \ldots \tau_0 \rbrack\!\rbrack) \wedge R \Rrightarrow R \left[\frac{u}{\tau_0}\right] \quad \square$$

The interval from u to τ_0 corresponds to the idle period before executing the command with effect R.

A predicate R is post-idle-invariant if adding a postfix idle period has no effect. That is, for any u greater than or equal to τ, whenever R holds over an interval from τ_0 to τ, it also holds over the interval from τ_0 to u, provided the variables under the control of the program are not modified between τ and u.

Definition 6.2.19 (post-idle-invariant) *A relation R is* post-idle-invariant *provided,*

$$\tau_0 \leq \tau \leq u < \infty \wedge \mathit{stable}(\rho.\mathit{out}, \lbrack\!\lbrack \tau \ldots u \rbrack\!\rbrack) \wedge R \Rrightarrow R \left[\frac{u}{\tau}\right] \quad \square$$

The interval from τ to u corresponds to the idle period after executing the command with effect R. Note that we rule out τ and u being infinity; if τ is infinity the command does not terminate and nothing can follow it. If the only references to τ_0 and τ in R are as indices of outputs, then R is both pre- and post-idle-invariant.

Law 6.2.20 (selection) *Given an idle-invariant, single-state predicate P, a pre- and post-idle-invariant relation R, and idle-stable boolean-valued expressions $B_1, \ldots, B_n$, provided $P \Rrightarrow (B_1 \mathbin{@} \tau \vee \cdots \vee B_n \mathbin{@} \tau)$,*

$$\begin{array}{l} \quad \infty \vec{x}\colon \lbrack P,\ R \rbrack \\ \sqsubseteq \textbf{if } B_1 \rightarrow \infty \vec{x}\colon \lbrack P \wedge B_1 \mathbin{@} \tau,\ R \rbrack \ [\!] \ \ldots \ [\!] \ B_n \rightarrow \infty \vec{x}\colon \lbrack P \wedge B_n \mathbin{@} \tau,\ R \rbrack \textbf{ fi} \end{array}$$

$\square$

For example,

$$\begin{array}{l} \mathit{gate}\colon \left[\mathit{true},\ \begin{array}{l}(\mathit{gate}(\tau_0) = \mathit{close} \Rightarrow (\mathit{gate} = \mathit{close}) \textbf{ on } \lbrack\!\lbrack \tau_0 \ldots \tau \rbrack\!\rbrack) \wedge \\ \mathit{gate}(\tau) = \mathit{close}\end{array}\right] \\ \sqsubseteq \textbf{if } \mathit{gate} = \mathit{close} \rightarrow \\ \quad \mathit{gate}\colon \left[\mathit{gate}(\tau) = \mathit{close},\ \begin{array}{l}(\mathit{gate}(\tau_0) = \mathit{close} \Rightarrow \\ \quad (\mathit{gate} = \mathit{close}) \textbf{ on } \lbrack\!\lbrack \tau_0 \ldots \tau \rbrack\!\rbrack) \wedge \\ \mathit{gate}(\tau) = \mathit{close}\end{array}\right] \qquad (16) \\ \quad [\!] \ \mathit{gate} = \mathit{open} \rightarrow \\ \quad \mathit{gate}\colon \left[\mathit{gate}(\tau) = \mathit{open},\ \begin{array}{l}(\mathit{gate}(\tau_0) = \mathit{close} \Rightarrow \\ \quad (\mathit{gate} = \mathit{close}) \textbf{ on } \lbrack\!\lbrack \tau_0 \ldots \tau \rbrack\!\rbrack) \wedge \\ \mathit{gate}(\tau) = \mathit{close}\end{array}\right] \qquad (17) \\ \quad \textbf{fi} \end{array}$$

The first branch (16) can be refined via Law 6.2.6 (output stable) to **skip**, and the second branch (17) to $\mathit{gate} : \mathbf{write}(\mathit{close})$.

6.2.5 *Local and auxiliary variables*

A variable block introduces a new local or auxiliary variable. The allocation and deallocation of a local variable may take time. This is allowed for in the definition by the use of **idle** commands. Auxiliary variables require no allocation or deallocation time. In the definition of a local or auxiliary variable block we need to allow for the fact that a variable of the same name may be declared at an outer scope, i.e. that it is already in ρ. Hence we introduce a fresh variable name, not in ρ, that we use in the definition via appropriate renamings. If the variable name is itself fresh, it may be used instead and the renaming avoided.

Definition 6.2.21 (block) *Given an environment, ρ, a command, C, a nonempty type T, and a fresh variable, w, not in ρ,*

$$\mathcal{M}_\rho\,(|[\ \mathbf{var}\, y : T;\ C\ ]|) \mathrel{\widehat{=}} (\exists\, w_0, w : T \bullet \mathcal{M}_{\rho'}\left(\mathbf{idle};\ C\left[\tfrac{w_0,w}{y_0,y}\right];\ \mathbf{idle}\right))$$

where ρ' is ρ updated with the local variable w, and

$$\mathcal{M}_\rho\,(|[\ \mathbf{aux}\, y : T;\ C\ ]|) \mathrel{\widehat{=}} (\exists\, w_0, w : T \bullet \mathcal{M}_{\rho''}\left(C\left[\tfrac{w_0,w}{y_0,y}\right]\right))$$

where ρ'' is ρ updated with the auxiliary variable w. □

For the law to refine a specification to a local variable block, we require that the assumption of a specification be impervious to the time taken to allocate the local variable, and the effect be impervious to both the time taken to allocate and deallocate the local variable.

Law 6.2.22 (local variable) *Provided P is an idle-invariant, single-state predicate, R is a pre- and post-idle-invariant relation, T is a nonempty type, and y does not occur free in $\vec{x}$, P and R,*

$$\vec{x}{:}\ [P,\ \ R] \sqsubseteq |[\ \mathbf{var}\, y : T;\ y, \vec{x}{:}\ [P,\ \ R]\ ]| \quad \square$$

Because no time is required to allocate and deallocate auxiliary variables, the law for them is the same as above but without all the idle-invariant requirements. We abbreviate multiple declarations with distinct names by merging them into a single block, e.g., $|[\ \mathbf{var}\, y;\ \mathbf{aux}\, x;\ C\ ]| = |[\ \mathbf{var}\, y;\ |[\ \mathbf{aux}\, x;\ C\ ]|\ ]|$. For example,

$$\infty\ [S,\ \ near \uparrow 0 \le \tau < \infty \vee near \uparrow 0 = \tau = \infty] \tag{18}$$

$\sqsubseteq$ Law 6.2.22 (local variable)

$$|[\ \mathbf{var}\, sens : boolean;\ \mathbf{aux}\, before : Time;$$
$$\infty\, sens, before{:}\ [S,\ \ near \uparrow 0 \le \tau < \infty \vee near \uparrow 0 = \tau = \infty] \tag{19}$$
$$]|$$

provided the assumption, S, is idle-invariant (see below), and the effect is pre-idle-invariant, that is, provided $\tau_0 < \infty$ and $u \le \tau_0 \le \tau$ the following holds,

$$stable(gate, [\![u \ldots \tau_0]\!]) \wedge (near \uparrow 0 \le \tau < \infty \vee near \uparrow 0 = \tau = \infty)$$
$$\Rrightarrow near \uparrow 0 \le \tau < \infty \vee near \uparrow 0 = \tau = \infty$$

and post-idle-invariant, that is, provided $\tau_0 \leq \tau \leq u < \infty$ the following holds,

$$\begin{array}{l} \quad \mathit{stable}(\mathit{gate}, \llbracket \tau \ldots u \rrbracket) \wedge (\mathit{near} \uparrow 0 \leq \tau < \infty \vee \mathit{near} \uparrow 0 = \tau = \infty) \\ \Rrightarrow \mathit{near} \uparrow 0 \leq u < \infty \vee \mathit{near} \uparrow 0 = u = \infty \end{array}$$

which holds because $\tau \leq u < \infty$.

6.3 An example

The example we consider is that of a railway crossing. There are sensors that detect when a train arrives *near* to the crossing and when it has passed *out* of the region of the crossing.

$$\textbf{input}\ \mathit{near}, \mathit{out} : \mathit{boolean};$$

The gate at the crossing is controlled by an output *gate*, which has values either *open* or *close*.

$$\textbf{output}\ \mathit{gate} : \{\mathit{open}, \mathit{close}\}$$

We use the notation $\mathit{near} \uparrow 0$ to refer to the time at which the train reaches the *near* sensor and it rises (from *false* to *true*) for the first time, and $\mathit{out} \uparrow 0$ for the time it reaches the *out* sensor. The sensors remain true for a minimum period when a train passes. The *near* sensor is placed so that there is a period of at least 300 seconds between a train arriving at the sensor and its arriving at the crossing. From the time the *gate* is set to *close* it takes at most 100 seconds for the gate to actually reach the closed position, and a similar time for it to rise. The gate should start reopening within 5 seconds of the train passing the *out* sensor.

$$\begin{array}{l} \textbf{const}\ \mathit{err} = 1\,\mathtt{s}\,; \text{ -- minimum time the sensors are true} \\ \textbf{const}\ \mathit{train\_to\_crossing} = 300\,\mathtt{s}\,; \\ \textbf{const}\ \mathit{time\_to\_close\_gate} = 100\,\mathtt{s}\,; \\ \textbf{const}\ \mathit{out\_lim} = 5\,\mathtt{s}\,; \end{array}$$

The *out* sensor is placed to ensure that the train has left the crossing before it reaches the *out* sensor. The controller may assume the following holds initially.

$$\begin{array}{rl} & (\mathit{near} = \mathit{false})\ \textbf{on}\ \{\tau \ldots \mathit{near} \uparrow 0\} \wedge \\ & (\mathit{near} = \mathit{true})\ \textbf{on}\ \{\mathit{near} \uparrow 0 \ldots \mathit{near} \uparrow 0 + \mathit{err}\} \wedge \\ \mathit{SENS} \mathrel{\widehat{=}} & (\mathit{out} = \mathit{false})\ \textbf{on}\ \{\tau \ldots \mathit{out} \uparrow 0\} \wedge \\ & (\mathit{out} = \mathit{true})\ \textbf{on}\ \{\mathit{out} \uparrow 0 \ldots \mathit{out} \uparrow 0 + \mathit{err}\} \wedge \\ & \mathit{near} \uparrow 0 + \mathit{train\_to\_crossing} < \mathit{out} \uparrow 0 \end{array} \tag{20}$$

Because the above predicate is idle-invariant and no variables appearing within it are in the frame, it may be assumed throughout the development. Note that $\{\tau \ldots \mathit{near} \uparrow 0\}$ may be empty, but the predicate is still idle-invariant.

The specification of the gate controller is as follows, in which the constant $200\,\mathtt{s}$ is derived from *train_to_crossing* minus *time_to_close_gate*. The final conjunct

in the effect, $gate(\tau) = open$, is required because the interval in the second last conjunct may be empty.

$$gate: \left[\begin{array}{ll} & (gate = open)\ \mathbf{on}\ [\![\tau_0 \ldots near \uparrow 0]\!] \wedge \\ gate(\tau) = open & (gate = close)\ \mathbf{on} \\ \wedge\ \tau \leq near \uparrow 0, & \quad [\![near \uparrow 0 + 200\,\mathbf{s} \ldots out \uparrow 0]\!] \wedge \\ \wedge\ SENS & (gate = open)\ \mathbf{on}\ [\![out \uparrow 0 + out\_lim \ldots \tau]\!] \wedge \\ & gate(\tau) = open \end{array}\right]$$

$\sqsubseteq$ Law 6.2.14 (multiple sequential compositions)

$$gate: \left[\begin{array}{ll} gate(\tau) = open & \\ \wedge\ \tau \leq near \uparrow 0, & (gate = open)\ \mathbf{on}\ [\![\tau_0 \ldots near \uparrow 0]\!] \\ \wedge\ SENS & \wedge\ near \uparrow 0 \leq \tau \wedge SENS \end{array}\right]; \tag{2}$$

$$gate: \left[\begin{array}{ll} near \uparrow 0 \leq \tau & (gate = close)\ \mathbf{on} \\ \wedge\ SENS \quad , & \quad [\![near \uparrow 0 + 200\,\mathbf{s} \ldots out \uparrow 0]\!] \\ & \wedge\ out \uparrow 0 \leq \tau \wedge SENS \end{array}\right]; \tag{21}$$

$$gate: \left[\begin{array}{ll} out \uparrow 0 \leq \tau & (gate = open)\ \mathbf{on}\ [\![out \uparrow 0 + out\_lim \ldots \tau]\!] \\ \wedge\ SENS \quad , & \wedge\ gate(\tau) = open \end{array}\right] \tag{22}$$

The specification (2) is refined earlier. Specification (21) is refined as follows.

(21) $\sqsubseteq$ Law 6.2.12 (sequential composition)

$$gate: \left[\begin{array}{ll} near \uparrow 0 \leq \tau & gate(\tau) = close \wedge \tau \leq near \uparrow 0 + 200\,\mathbf{s} \\ \wedge\ SENS \quad , & \wedge\ SENS \end{array}\right]; \tag{23}$$

$$gate: \left[\begin{array}{lll} gate(\tau) = close \wedge & & (gate = close)\ \mathbf{on} \\ \tau \leq near \uparrow 0 + 200\,\mathbf{s}, & & \quad [\![near \uparrow 0 + 200\,\mathbf{s} \ldots out \uparrow 0]\!] \\ \wedge\ SENS & & \wedge\ out \uparrow 0 \leq \tau \wedge SENS \end{array}\right] \tag{24}$$

The specification (23) may be refined by setting *gate* to *close* by the deadline.

(23) $\sqsubseteq$ Law 6.2.13 (separate deadline); Law 6.2.4 (strengthen effect)

$$gate: [\![near \uparrow 0 \leq \tau \wedge SENS,\ \ gate(\tau) = close]\!]; \tag{25}$$
$$\mathbf{deadline}\ near \uparrow 0 + 200\,\mathbf{s}$$

Specification (25) can be implemented by $gate : \mathbf{write}(close)$. Specification (24) can be refined in a manner similar to (2) to give the following specification.

$$[\![SENS,\ \ out \uparrow 0 \leq \tau]\!] \tag{26}$$

Specification (22) can be refined to the following (similar to (23)).

$$gate : \mathbf{write}(open);\ \mathbf{deadline}\ out \uparrow 0 + out\_lim$$

The program so far is shown in Fig. 6.2. The initial assumption, $\tau \leq near \uparrow 0$, has been factored out of the specification (4). The remaining unrefined components, B and D, require a repetition for their implementation.

$A :: \{\tau \leq near \uparrow 0\};$
$B :: [SENS,\ near \uparrow 0 \leq \tau];$
$\quad gate : \mathbf{write}(close);$
$C :: \mathbf{deadline}\ near \uparrow 0 + 200\,\mathbf{s};$
$D :: [SENS,\ out \uparrow 0 \leq \tau];$
$\quad gate : \mathbf{write}(open);$
$E :: \mathbf{deadline}\ out \uparrow 0 + out\_lim$

Figure 6.2. Collected program without repetitions

6.4 Repetitions

The specification $[SENS,\ near \uparrow 0 \leq \tau]$ can be implemented by repeatedly testing the near sensor until it becomes true. The specification $[SENS,\ out \uparrow 0 \leq \tau]$ can be implemented in a similar manner. Hence we only consider the former here. To provide an example of refinement to a possibly nonterminating repetition, we generalise the specification to

$$\infty\ [S,\ near \uparrow 0 \leq \tau < \infty \vee near \uparrow 0 = \tau = \infty] \tag{18}$$

although in this particular example we know $near \uparrow 0 < \infty$. From the assumption *SENS* we may assume the near sensor is false until time $near \uparrow 0$ and then true for a minimum period of *err*. We need to sample the sensor frequently enough to ensure its high transition is not missed.

$$S \mathrel{\widehat{=}} \begin{array}{l}(near = false)\ \mathbf{on}\ (\tau \ldots near \uparrow 0) \wedge \\ (near = true)\ \mathbf{on}\ (near \uparrow 0 \ldots near \uparrow 0 + err)\end{array} \tag{27}$$

The predicate (27) is idle-invariant because

$$\begin{array}{l}\tau_0 \leq \tau < \infty \wedge stable(out, [\tau_0 \ldots \tau]) \wedge \\ (near = false)\ \mathbf{on}\ (\tau_0 \ldots near \uparrow 0) \wedge \\ (near = true)\ \mathbf{on}\ (near \uparrow 0 \ldots near \uparrow 0 + err) \\ \Rrightarrow (near = false)\ \mathbf{on}\ (\tau \ldots near \uparrow 0) \wedge \\ (near = true)\ \mathbf{on}\ (near \uparrow 0 \ldots near \uparrow 0 + err)\end{array}$$

We introduce a local variable *sens*, which is used for sampling the sensor, and an auxiliary variable *before*, which is used to record the time immediately before the sensor is sampled. Specification (18) has been refined in Sect. 6.2.5 to such a block with body (19).

$$\infty\ sens, before: [S,\ near \uparrow 0 \leq \tau < \infty \vee near \uparrow 0 = \tau = \infty] \tag{19}$$

Specification (19) can be refined by a repetition. We do not attempt to give a complete definition of repetitions; more complete details can be found elsewhere

[7, 8]. A repetition,

$$R \mathrel{\widehat{=}} \textbf{repeat}\ C\ \textbf{until}\ B,$$

can (as a first approximation) be characterised by the following recurrence.

$$R = \textbf{idle};\ C;\ (\lceil B \mathbin{@} \tau \rceil \ [\!]\ (\lceil \neg\, B \mathbin{@} \tau \rceil\ ;\ R))$$

Before the body of the repetition is executed there is an **idle** to allow for any overheads at the start of an iteration. After executing the body C, there is a (deterministic) choice between two guarded alternatives. The guard evaluation typically takes time (unless the guard is a constant, *true* or *false*). The first alternative corresponds to the guard evaluating to true and termination of the repetition. The second alternative corresponds to the guard evaluating to false and the iteration being repeated from the beginning.

Unfortunately, the above recurrence allows a single iteration of a repetition (for example of '**repeat skip until** *false*') to take zero time, or each successive iteration to take half the time of the previous (as in Zeno's paradox). To avoid this unrealistic behaviour, we define every iteration to take a minimum amount of time, d, which is strictly positive (1 attosecond will do). Hence a repetition is characterised by: there exists a strictly positive time, d, such that

$$\begin{aligned} R = {} & [\![\ \textbf{aux}\ s;\ s := \tau;\ \textbf{idle};\ C; \\ & \quad (\lceil B \mathbin{@} \tau \rceil \ [\!]\ (\lceil \neg\, B \mathbin{@} \tau \rceil\ ;\ \lceil s + d \leq \tau \rceil\ ;\ R)) \\ &]\!] \end{aligned}$$

where s is a fresh auxiliary variable, which captures the start time of an iteration. Before the repetition is restarted from the beginning there is a delay $\lceil s + d \leq \tau \rceil$ to ensure the time is at least d time units later than the start time of the iteration, s. This ensures that even if the guard is the constant *false* and the body is the null command **skip**, each iteration takes at least d time units and hence Zeno-like behaviour is avoided.

We give a rule for introducing a repetition with a body that terminates on every iteration. The predicate Q' acts as an invariant that is established at the end of the body on every iteration. Q' is not required to be idle-invariant. Hence we introduce a weaker predicate Q that is idle-invariant. Only Q can be assumed after the guard evaluation. If the repetition terminates both B and Q hold. If the guard evaluates to false, then at the start of the next iteration one can assume both $\neg\, B$ and Q. The body is executed initially when P is known to hold, or on a repetition when the guard is false, in which case $\neg\, B$ and Q hold. The body of the repetition establishes Q'.

The stronger (non-idle-invariant) predicate Q' is used in the case when the repetition does not terminate. If the repetition never terminates but the body always terminates then there is an infinite sequence of ever increasing times, corresponding to the times at which the end of the body is reached, at which both Q' and $\neg\, B$ hold for the current time and the current values of the state variables. This is captured by the predicate Q_∞ in the following law:

Law 6.4.1 (repetition) *Given idle-invariant, single-state predicates P and Q, a single-state predicate Q′ such that Q′ ⇛ Q, and an idle-stable, boolean-valued expression B,*

$$\begin{array}{l} \infty\, \vec{x}\colon \left[P,\ (B \mathbin{@} \tau \wedge Q \wedge \tau < \infty) \vee (Q_\infty \wedge \tau = \infty)\right] \\ \sqsubseteq \mathbf{repeat}\, \vec{x}\colon \left[P \vee (\neg B \mathbin{@} \tau \wedge Q),\ Q'\right] \mathbf{until}\, B \end{array}$$

where $Q_\infty \mathrel{\widehat{=}} (\forall t : \mathit{Time} \bullet (\exists \tau : \mathit{Time};\ \rho.v : T_v \bullet t \leq \tau \wedge \neg B \mathbin{@} \tau \wedge Q'))$. □

Note that the existential quantification in Q_∞ ensures that Q_∞ satifies Def. 6.2.2 (nontermination state independent).

For the train crossing example, a repetition can be used to test for the train passing the near sensor. We have weakened the assumption of the body (28) to *S*.

(19) ⊑ Law 6.4.1 (repetition); Law 6.2.3 (weaken assumption)

repeat

$$\mathit{sens},\ \mathit{before}\colon \left[S,\ \begin{array}{l} (\mathit{sens} \Rightarrow \mathit{near} \uparrow 0 \leq \tau) \wedge \\ (\neg\, \mathit{sens} \Rightarrow \mathit{before} \leq \mathit{near} \uparrow 0) \wedge \\ \tau \leq \mathit{near} \uparrow 0 + \mathit{err} \wedge S \end{array}\right] \tag{28}$$

until *sens*

The effect of the body corresponds to the loop invariant Q' of Law 6.4.1 (repetition). The conjunct $\tau \leq \mathit{near} \uparrow 0 + \mathit{err}$ is not idle-invariant, but the remaining conjuncts $(\mathit{sens} \Rightarrow \mathit{near} \uparrow 0 \leq \tau)$, $(\neg\, \mathit{sens} \Rightarrow \mathit{before} \leq \mathit{near} \uparrow 0)$ and *S* are idle-invariant. Hence these form the weaker condition *Q* in the law. If the repetition terminates both *Q* and the termination guard (*sens*) hold. Together these imply $\mathit{near} \uparrow 0 \leq \tau$, which along with termination ($\tau < \infty$), implies the effect of (19).

For nontermination, for any time *t* there exists a later time τ and corresponding values of the state variables, such that the invariant Q' and the negation of the termination guard ($\neg$ *sens*) hold. This implies the following.

$$\begin{array}{l} (\forall t : \mathit{Time} \bullet (\exists \tau : \mathit{Time};\ \mathit{sens} : \mathit{boolean};\ \mathit{before} : \mathit{Time} \bullet \\ \quad t \leq \tau \wedge \neg\, \mathit{sens} \wedge \mathit{before} \leq \mathit{near} \uparrow 0 \wedge \tau \leq \mathit{near} \uparrow 0 + \mathit{err})) \\ \Rrightarrow (\forall t : \mathit{Time} \bullet t \leq \mathit{near} \uparrow 0 + \mathit{err}) \\ \Rrightarrow \mathit{near} \uparrow 0 = \infty \end{array}$$

Along with $\tau = \infty$, this implies the effect of (19).

The refinement of the body (28) of the repetition depends on the assumption about the sensor behaviour (27). The sensor is sampled between the start time of the body of the repetition, which is captured in the time-valued auxiliary variable *before*, and time $\mathit{near} \uparrow 0 + \mathit{err}$. If the sampled value is false, the body must have begun execution before time $\mathit{near} \uparrow 0$, and if it is true, the finish time of the body must be after $\mathit{near} \uparrow 0$. Together these guarantee the effect of (28).

(28) ⊑ Law 6.2.4 (strengthen effect)

$$\mathit{sens}, \mathit{before}\colon \left[S,\ \begin{array}{l} \mathit{before} = \tau_0 \wedge \tau \leq \mathit{near} \uparrow 0 + \mathit{err} \wedge \\ \mathit{sens} \in \mathit{near}(\!|\tau_0 \mathinner{..} \mathit{near} \uparrow 0 + \mathit{err}|\!) \end{array}\right] \tag{12}$$

$$
\begin{array}{l}
[\![\ \textbf{var}\ \mathit{sens} : \mathit{boolean};\ \textbf{aux}\ \mathit{before} : \mathit{Time};\\
\quad \textbf{repeat}\\
\quad\quad F :: \mathit{before} := \tau;\\
\quad\quad\quad \mathit{sens} : \textbf{read}(\mathit{near});\\
\quad\quad G :: \textbf{deadline}\ \mathit{near} \uparrow 0 + \mathit{err};\\
\quad\quad\quad \left\{ \begin{array}{l} (\mathit{sens} \Rightarrow \mathit{near} \uparrow 0 \leq \tau) \wedge \\ (\neg\ \mathit{sens} \Rightarrow \mathit{before} \leq \mathit{near} \uparrow 0) \wedge \\ \tau \leq \mathit{near} \uparrow 0 + \mathit{err} \wedge S \end{array} \right\}\\
\quad \textbf{until}\ \mathit{sens}\\
]\!]
\end{array}
$$

Figure 6.3. Sensor detection repetition

$$
\begin{array}{l}
A :: \{\tau \leq \mathit{near} \uparrow 0\};\\
\quad \textbf{alloc var}\ \mathit{sens} : \mathit{boolean};\ \textbf{aux}\ \mathit{before} : \mathit{Time};\\
F :: \mathit{before} := \tau;\\
\quad \mathit{sens} : \textbf{read}(\mathit{near});\\
G :: \textbf{deadline}\ \mathit{near} \uparrow 0 + \mathit{err}
\end{array}
$$

Figure 6.4. Initial path entering sensor detection repetition

This specification is equivalent to (12), which has been refined earlier.

The complete repetition is given in Fig. 6.3. The deadline command in the repetition ensures that the high transition of the sensor is not missed.

6.5 Timing-constraint analysis

In order for compiled machine code to implement a machine-independent program it must guarantee to meet all the deadlines. The auxiliary variables introduced above aid this analysis. There is a deadline within the sensor detection repetition (Fig. 6.3) labelled G. It is reached initially from the entry to the repetition and subsequently on each iteration. The initial entry path (shown in Fig. 6.4) starts at the assumption A in Fig. 6.2 before entering the sensor detection repetition (which refines $[\mathit{SENS},\ \mathit{near} \uparrow 0 \leq \tau]$) in Fig. 6.3. The path allocates the local variable *sens*, extends the auxiliary variables with *before*, and follows the path into the repetition, assigning τ to *before* and reading *near* into *sens*, before reaching the deadline G. From the assumption at A, we know that the start time of the path is before $\mathit{near} \uparrow 0$ and the deadline on the path is $\mathit{near} \uparrow 0 + \mathit{err}$. If

$$
\begin{array}{l}
F :: \mathit{before} := \tau; \\
\quad \mathit{sens} : \mathbf{read}(\mathit{near}); \\
G :: \mathbf{deadline}\ \mathit{near} \uparrow 0 + \mathit{err}; \\
\quad \left\{ \begin{array}{l} (\mathit{sens} \Rightarrow \mathit{near} \uparrow 0 \leq \tau) \wedge \\ (\neg\ \mathit{sens} \Rightarrow \mathit{before} \leq \mathit{near} \uparrow 0) \wedge \\ \tau \leq \mathit{near} \uparrow 0 + \mathit{err} \wedge S \end{array} \right\}; \\
\quad [\neg\ \mathit{sens}];\ \text{-- repetition exit condition false} \\
F :: \mathit{before} := \tau; \\
\quad \mathit{sens} : \mathbf{read}(\mathit{near}); \\
G :: \mathbf{deadline}\ \mathit{near} \uparrow 0 + \mathit{err}
\end{array}
$$

Figure 6.5. Repetition path in sensor detection repetition

this path is guaranteed to execute in a time of less than *err* then the deadline is guaranteed to be met. Hence the timing constraint on the path is *err*.

For an iteration we consider the path (shown in Fig. 6.5) that starts at the assignment to *before* (*F*), reads the value of *near* into *sens*, passes through the deadline (*G*), branches back to the start of the repetition because *sens* is not true, performs the assignment to *before* (*F*), reads the value of *near*, and reaches the deadline (*G*). The guard evaluation is represented by $[\neg\ \mathit{sens}]$, which indicates that in order for the path to be followed, *sens* must be false. Using the loop invariant we can determine that the initial time assigned to *before*, i.e. the time at which the path begins execution, must be before $\mathit{near} \uparrow 0$ because the value of *sens* is false. The final deadline on the path is $\mathit{near} \uparrow 0 + \mathit{err}$. Hence, if the path is guaranteed to execute in less than time *err*, it will always meet its deadline.

If this path is guaranteed to reach its deadline then any path with this as a suffix is also guaranteed to meet the final deadline, and hence any number of repeated iterations will meet the deadline. The constraint on this path corresponds to a maximum time of *err* between successive reads of the sensor. Although the repetition is written as a busy wait, in a multi-tasking environment the repetition could be implemented by scheduling the body to execute so that the deadline is always met. For example, a common scheduling strategy is periodic scheduling in which a task is scheduled with a period of *P* seconds and has to complete within *D* seconds of the start of the period. In this case as long as $P + D$ is less than or equal to *err*, the requirements for meeting the deadline will be met.

The path shown in Fig. 6.6 starts from the deadline *G* within the body of the repetition, exits the repetition (because *sens* is true), deallocates *sens* and *before*, and sets the *gate* to *close*, before reaching the deadline at *C*. The initial deadline guarantees the start time of the path is less than or equal to $\mathit{near} \uparrow 0 + \mathit{err}$. The deadline on the path is $\mathit{near} \uparrow 0 + 200\,\mathrm{s}$. Therefore a suitable constraint on the path is $\mathit{near} \uparrow 0 + 200\,\mathrm{s} - (\mathit{near} \uparrow 0 + \mathit{err}) = 200\,\mathrm{s} - \mathit{err}$.

G :: **deadline** *near* $\uparrow 0 + err$;
  [*sens*]; -- exit repetition
  dealloc var *sens* : *boolean*; **aux** *before* : *Time*;
  gate : **write**(*close*);
C :: **deadline** *near* $\uparrow 0 + 200$ s

Figure 6.6. Exit path from sensor detection repetition

We have considered all the paths concerned with the repetition testing the near sensor. The remainder of the program which handles the sensor for the train leaving the crossing is treated in a similar manner.

In general, timing constraint analysis is undecidable because it encompasses the halting problem for a path containing a complete repetition without any internal deadlines. However, for restricted forms of programs automating timing constraint analysis is possible [4].

6.6 Conclusions

The real-time refinement calculus presented in this paper supports the development of *machine-independent* real-time programs. This has the advantage of decoupling the program development process from the timing analysis required for a particular machine. Timing constraints within the program are represented by deadline commands.

In this paper we have developed a predicative semantics for the calculus, in a style similar to that used by Hehner, and Hoare and He. A novel feature is that external inputs and outputs are represented by timed traces, and hence the values of such variables over time, and not just their initial and final values are significant. In addition, programs may be nonterminating. Commands in the language satisfy a number of healthiness properties: time cannot go backwards; the semantics of a nonterminating command is independent of the final values of the state variables; and all commands have no effect if 'executed' at time infinity.

In the laws for reasoning about compound commands we desired that reasoning about the behaviour of commands is independent of the time taken to execute components of the commands, such as guard evaluation. This is achieved by requiring predicates to be idle invariant. As to be expected the most interesting constructs to handle are sequential composition and repetitions. For a sequential composition, in order to model the reactive nature of real-time programs, we desired that the behaviour of the sequential composition over time be composed from the behaviour of the individual commands. Care needs to be taken with the case in which the first command in the composition does not terminate, and the behaviour of the sequential composition is the same as that of the first command.

For a nonterminating repetition, the values of the outputs are extended on each iteration. The law for reasoning about such repetitions relies on the loop invariant being repeatedly re-established at an infinite sequence of ever increasing times.

Acknowledgements

This research was supported by Australian Research Council (ARC) Large Grant A49801500, *A Unified Formalism for Concurrent Real-time Software Development*. I would like to thank Yifeng Chen, Colin Fidge, Karl Lermer, and Mark Utting for fruitful discussions on the topic of this paper, and the members of IFIP Working Group 2.3 on Programming Methodology for feedback on this topic, especially Eric Hehner for his advice on how to simplify our approach, and Cliff Jones for his insights into the relational approach. I would like to acknowledge the support of the University of Queensland Special Studies Program and thank the Department of Computer Science at the University of York (UK) for their hospitality.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[3] C. J. Fidge, I. J. Hayes, and G. Watson. The deadline command. *IEE Proceedings—Software*, 146(2):104–111, April 1999.

[4] S. Grundon, I. J. Hayes, and C. J. Fidge. Timing constraint analysis. In C. McDonald, editor, *Computer Science '98: Proc. 21st Australasian Computer Sci. Conf. (ACSC'98)*, Perth, 4–6 Feb., 575–586. Springer-Verlag, 1998.

[5] I. J. Hayes. Separating timing and calculation in real-time refinement. In J. Grundy, M. Schwenke, and T. Vickers, editors, *Int. Refinement Workshop and Formal Methods Pacific 1998*, 1–16. Springer-Verlag, 1998.

[6] I. J. Hayes. Real-time program refinement using auxiliary variables. In M. Joseph, editor, *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Comp. Sci.*, 170–184. Springer-Verlag, 2000.

[7] I. J. Hayes. Reasoning about non-terminating loops using deadline commands. In R. Backhouse and J. N. Oliveira, editors, *Proc. Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, 60–79. Springer-Verlag, 2000.

[8] I. J. Hayes. Reasoning about real-time repetitions: Terminating and nonterminating. Technical Report 01-04, Software Verification Research Centre, The University of Queensland, Brisbane 4072, Australia, February 2001.

[9] I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In D. J. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop (NFMW'96)*. Springer-Verlag, 1997.

[10] I. J. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385–448, 2001.

[11] E. C. R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.

[12] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[13] J. Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–825, 1994.

[14] J. Hooman and O. van Roosmalen. Formal design of real-time systems in a platform-independent way. *Parallel and Distributed Computing Practices*, 1(2):15–30, 1998.

[15] C. B. Jones. Program specification and verification in VDM. Technical Report UMCS-86-10-5, Department of Computer Science, University of Manchester, 1986.

[16] B. P. Mahony. *The Specification and Refinement of Timed Processes*. PhD thesis, Department of Computer Science, University of Queensland, 1992.

[17] B. P. Mahony and I. J. Hayes. Using continuous real functions to model timed histories. In P. A. Bailes, editor, *Proc. 6th Australian Software Engineering Conf. (ASWEC91)*, 257–270. Australian Comp. Soc., 1991.

[18] B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Trans. on Software Engineering*, 18(9):817–826, 1992.

[19] C. C. Morgan. *Programming from Specifications*, Second edition. Prentice Hall, 1994.

[20] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[21] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[22] M. Utting and C. J. Fidge. A real-time refinement calculus that changes only time. In He Jifeng, editor, *Proc. 7th BCS/FACS Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, July 1996.

[23] M. Utting and C. J. Fidge. Refinement of infeasible real-time programs. In *Proc. Formal Methods Pacific '97*, 243–262, Wellington, New Zealand, July 1997. Springer-Verlag.

7

Aspects of system description

Michael Jackson

Abstract

This paper discusses some aspects of system description that are important for software development. Because software development aims to solve problems in the world, rather than merely in the computer, these aspects include: the distinction between the hardware/software machine and the world in which the problem is located; the relationship between phenomena in the world and formal terms used in descriptions; the idea of a software model of a problem world domain; and an approach to the decomposition of problems and its consequences for the larger structure of software development descriptions.

7.1 Introduction

The business of software development is, above all, the business of making descriptions. A *program* is a description of a computation—or, perhaps, of a machine behaviour. A *specification* is a description of the input-output relation of a computation—or, perhaps, of the externally observable behaviour of a machine. A *requirement* is a description of some observable effect or condition that our customer wants the computation—or the machine— to guarantee. A *software design* is a description of the structure of the computation—or, perhaps, of a machine that will execute the computation.

In spite of its importance, we pay surprisingly little attention to the practice and technique of description. For the most part, it is treated only implicitly and indirectly, either because it is thought too trivial to engage our attention, or because we suppose that all software developers must already be fully competent practitioners. In the same way, the great universities in the eighteenth and early nineteenth century ignored the study of English literature. It was a truth universally acknowledged that anyone qualified to study Latin and Greek and mathematics in the university must already know everything worth knowing about the subject of English literature.

A. McIver et al. (eds.), *Programming Methodology*

But the discipline of description, like the study of English literature, is neither trivial nor universally understood. Many aspects of description technique are important in software development and merit explicit discussion. The following sections discuss particular aspects, setting them in the context of some simple problems. A concluding section briefly discusses the relationship between the view presented here and a narrower view of the scope of research, teaching and practice in software development.

7.2 Symbol manipulation

It has often seemed attractive to regard software development as a branch of pure mathematics. The computer is a symbol-processing machine. Each problem to be solved is formal, drawn from a pure mathematical domain. The development methods to be used are largely formal, with the addition of the intuitive leaps that are characteristic of creative mathematical work. And the criterion of success—correctness with respect to a precise program specification—is entirely formal.

This view has underpinned some notable advances in programming. It has led to the evolution of a powerful discipline based on simultaneous development of a program and its correctness proof, and a clear demonstration that, for some programs at least, correctness is an achievable practical goal. The class of such programs is large. It includes a repertoire of well-known small examples—such as GCD and searching or sorting an array— and many substantial applications—such as compiling program texts, finding maximal strong components in a graph, model-checking, and the travelling-salesman problem.

These are all problems with a strong algorithmic aspect. Their subject matter is abstract and purely mathematical, even when the abstraction and the mathematics have clear practical application. This is what allows the emphasis in software development to be placed on symbol manipulation. As Hermann Weyl expressed it [11]:

> "We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for. ... [The mathematician] need not be idle; there are many operations he may carry out with these symbols, without ever having to look at the things they stand for."

He might have gone further. We can't look at what the symbols stand for, because they don't stand for anything outside the mathematics: they are themselves the subject matter of the computation. The task of relating the mathematics to a practical problem is not part of the software developer's concern: it is someone else's business. Although our problem may be called *the Travelling Salesman problem* we are not really interested in the real salesmen and their travels, but only in the abstraction we have made of them.

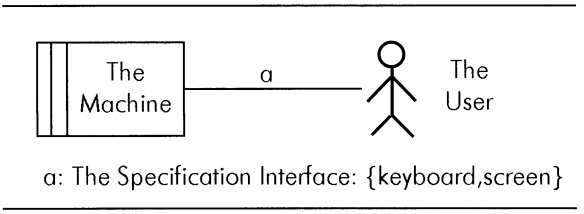


Figure 7.1. The machine and the user

7.2.1 *The specification firewall*

But even in the most formal problems an element of informality may intrude. A useful program must make its results visible outside the computer; most programs also accept some input. So questions of external representation and of data formats, at least, must be considered. How, for example, should we require our program's user to enter the nodes and arcs of the graph over which the salesman travels?

These less formal concerns arise outside the core computation itself, in the world of the software's users and the software developer's customers. In many cases they can relegated to a limbo beyond a *cordon sanitaire* by focusing on the *program specification*. As Dijkstra wrote [3]:

> "The choice of functional specifications—and of the notation to write them down in—may be far from obvious, but their role is clear: it is to act as a logical 'firewall' between two different concerns. The one is the 'pleasantness problem', i.e. the question of whether an engine meeting the specification is the engine we would like to have; the other one is the 'correctness problem,' ie the question of how to design an engine meeting the specification. ... the two problems are most effectively tackled by ... psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem."

Figure 7.1 pictures the situation. The specification interface *a* is an interface of shared physical phenomena connecting the customer to the machine. At this interface the customer enters input data, perhaps by keyboard, and receives output data, perhaps by seeing it displayed on the screen. The shared phenomena for the input are the keystrokes: these are shared events controlled by the customer. The shared phenomena for the output are the characters or graphics visible on the screen: these are shared states, controlled by the machine.

The specification firewall is erected at this interface. It enforces a fruitful separation of the 'hard' formal concerns of the software developer and computer scientist from the 'soft' concerns of the 'systems analyst', addressing informal problems in the world outside the computer. The software developers are relieved of responsibility for the world outside the computer: they need no more discuss

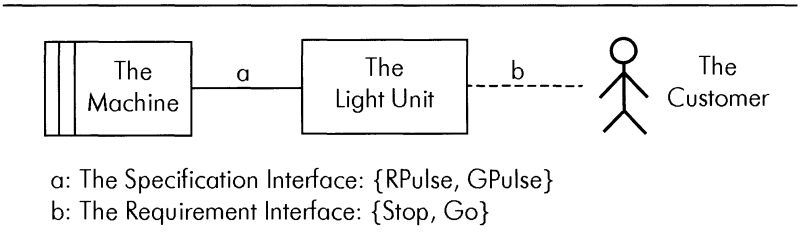


Figure 7.2. The machine, the world and the customer

the external data format for a graph than automobile engineers need discuss the range of paint colours for their cars' bodywork or the choice of upholstery fabric for the seats. The subject matter for serious attention and reasoning is restricted to the mathematics of the problem abstraction and of the computation that the machine will execute.

The 'soft' concerns, then, are relatively unimportant; they are relegated to a secondary place. The customer—who may well be the developer or another computer scientist with similar concerns and interests—may be slightly irritated by an inferior choice of input-output format at the specification interface, but is not expected to regard it as a crucial defect. The essential criterion, by which the work is to be judged, is the correctness and efficiency of the computation.

7.3 The Machine and the World

Not all customers will be so compliant. For most practical software development the customer's vital need is not to solve a mathematical problem, but to achieve specific observable physical effects in the world. Consider the very small problem of controlling a traffic light unit. The unit is placed at the gateway to a factory, and controls incoming traffic by allowing entry only during 15 seconds of each minute. The unit has a Stop lamp and a Go lamp. The problem is to ensure that the light shows alternately Stop for 45 seconds and Go for 15 seconds, starting with Stop. We can picture the problem as it is shown in Figure 7.2.

In addition to the machine, we now show the *problem domain*: that is, the part of the world in which the problem is located. There is no user: in this problem—as in many others—it is not clear who is the *user*, or even whether the notion of a user is useful. But there is certainly a *customer*: the person, or the group of people, who pay for the development work and will look critically at its results.

7.3.1 The specification interface

As before, the specification interface *a* is an interface of phenomena shared by the *machine domain* and the *problem domain*. Here the problem domain is the lights

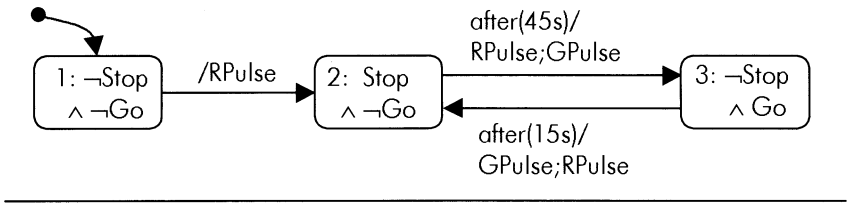


Figure 7.3. A system description

unit, and the shared phenomena are the signal pulses {*RPulse, GPulse*} by which the machine can cause it to switch on and off its Stop and Go lamps. The lights unit itself is on the other side of the specification interface.

7.3.2 *The requirement interface*

The customer is more remote from the machine than the user in a symbol manipulation problem. The customer's need is no longer located at the specification interface: the customer is interested in the regime of Stop and Go lamps, not in the signal pulses. So a new interface has appeared in the picture. The requirement interface *b* is a notional interface at which we can think of the customer as observing the world outside the machine. The phenomena of interest at this interface are the states of the Stop and Go lamps of the lights unit; these are, of course, quite distinct from the signal pulses at the interface with the machine.

The problem is about something physical and concrete. The externally visible behaviour of the machine, and the resulting behaviour of the lights unit, are not matters of pleasantness: they are the core of the problem.

7.3.3 *A system description*

Figure 7.3 is a description of the system as it might be described using a currently fashionable [10] diagrammatic notation derived from Statecharts [5]. In the transition markings the external stimulus, if any, is written before the slash ('/'), and the sequence of actions, if any, taken by the machine is written after it.

The initial state is 1, in which neither lamp is lit. Immediately the machine emits an RPulse, causing a transition to state 2, in which Stop is lit but not Go. 45 seconds after entering state 2, the machine emits an RPulse followed by a GPulse, causing a transition to state 3, in which Go is lit but not Stop. 15 seconds later the machine emits a GPulse followed by an RPulse, causing a transition back to state 2, and so on.

7.3.4 *Purposeful description*

It is always salutary in software development to ask why a particular description is worth making, and what particular purpose it serves in the development. In this tiny problem we can recognise three distinct roles that our system description is intended to play:

The requirement The *requirement* is a description that captures the effects our customer wants the machine to produce in the world. When we talk to the customer, we treat the description as a requirement. We ignore the actions that cause the pulses, and focus just on the timing events and the states. "To begin with," we say, "both lamps should be off; then, for 45 seconds, the Stop lamp only should be lit; then, for the next 15 seconds, the Go lamp only should be lit;" and so on. The requirement that emerges is:

```
forever {
  show only Stop for 45 seconds;
  show only Go for 15 seconds;
}
```

The machine specification The *specification* describes the behaviour of the machine in terms of the phenomena at the specification interface. It provides an interface between the problem analyst, who is concerned with the problem world, and the programmer, who is concerned only with the computer. When we talk to the programmer, we treat the description as a specification of the machine. We look only at the transitions with the timing events and the pulses. "First the machine must cause an RPulse," we tell the programmer, "then, after 45 seconds, an RPulse and a GPulse;" and so on. The Stop and Go states have no significance to the programmer, because they aren't visible to the machine; at best they are enlightening comments suggesting why the pulses are to be caused. The specification that emerges is:

```
{ RPulse;
  forever {
    wait 45 seconds; RPulse; GPulse;
    wait 15 seconds; GPulse; RPulse;
  }
}
```

The domain description The *domain description* bridges the gap between the requirement and the specification. The customer wants a certain regime of Stop and Go lamps, but the machine can directly cause only RPulses and GPulses. The gap is bridged by the properties of the problem domain. Here that means the properties of the lights unit. When we talk to the lights unit designer to check our understanding of the domain properties, we focus just on the pulses and the way they affect the states. "In the unit's initial state both lamps are off: That's right, isn't it? Then an initial RPulse turns the Stop lamp on; then an RPulse followed by a GPulse turns the Stop lamp

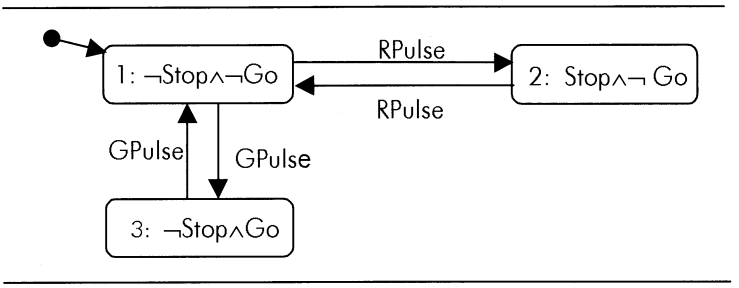


Figure 7.4. A partial domain description

off and lights the Go lamp, doesn't it?" and so on. The domain properties description that emerges[1] is shown in Figure 7.4.

7.3.5 *Why separate descriptions are needed*

Combining the three descriptions into one is tempting, but in a realistic problem it is very poor practice for several reasons. First, if the description were only slightly more complex it could be very hard to tease out the *projection* needed for each of the three roles.

Second, the adequacy of our development must be shown by an argument relating the three separate descriptions. Our goal is to bring about the regime of Stop and Go lamps that our customer desires. We must show that a machine programmed according to our specification will ensure this regime by virtue of the properties of the lights unit. That is:

$$\textit{specification} \wedge \textit{domain properties} \Rightarrow \textit{requirement}$$

In other words: if the machine meets its specification, and the problem world is as described in the domain properties, then the requirement will be satisfied[2]. The combined description does not allow this argument to be made explicitly.

Third, the single description combines descriptions of what we desire to achieve—the *optative* properties described in the requirement and specification—with a description of the known and given properties relied on—the *indicative* properties described in the domain description. It is always a bad idea to mix indicative and optative statements in the same description.

Fourth, the combined description is inadequate in an important way. Being based on a description of the machine behaviour, it can't accommodate a de-

[1] In fact, Figure 7.4 asserts much more than can be seen from the System Description given in Figure 7.3. For example: that it is possible to return to the dark state; that the first lamp turned on from the initial dark state may be the Go lamp; and that the RPulses affect only the Stop lamp and the GPulses only the Go lamp. Nothing in Figure 7.3 warrants these assertions.

[2] A fuller and more rigorous account of the relationship among the three descriptions is given in [4].

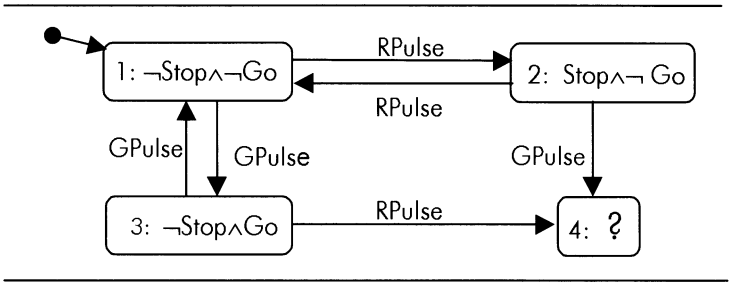


Figure 7.5. Lights-unit domain properties description

scription of what would happen if the machine were to behave differently—for example, by reversing the order of GPulse and RPulse in each pair. Figure 7.5 shows what a separate, full description of the domain properties might be.

Each lamp is toggled by pulses of the associated type: RPulse for Stop and GPulse for Go. The designer tells us that the unit can not tolerate the illumination of both lamps at the same time. We show state 4 as the *unknown state*, meaning that nothing is known about subsequent behaviour of the unit once it has entered state 4. Effectively, the unit is broken.

Fifth, the combined description isn't really re-usable. Because the embodied domain description, in particular, is merged with the requirement and the specification, it can't easily be re-used in another problem that deals differently with the same problem domain.

7.4 Describing the World

The three descriptions—requirements, domain properties and machine specification—are all concerned with event and state phenomena of the world in which the problem is located. But the first two are different from the third. The specification phenomena, shared with the machine, can properly be regarded as *formal*. Just as the machine has been carefully engineered so that there is no doubt whether a particular keystroke event has or has not occurred, so it has been carefully engineered to avoid similar doubt about whether an RPulse or a GPulse event has or has not occurred. The continuous underlying physical phenomena of magnetic fields and capacitances and voltages have been tamed to conform to sharply-defined discrete criteria.

But in general the phenomena and properties of the world have not been tamed in this way, and must be regarded as *informal*. The formalisation must be devised and imposed by the software developer. As W. Scherlis remarked [8] in his response to Dijkstra's observations [3] cited earlier:

> "One of the greatest difficulties in software development is formalization—capturing in symbolic representation a worldly computational

problem so that the statements obtained by following rules of symbolic manipulation are useful statements once translated back into the language of the world."

This task of formalization, along with appropriate techniques for its successful performance, is an integral, but regrettably much neglected, aspect of software development. Two important elements of this task are the use of *designations*, and the use and proper understanding of *formal definitions*.

7.4.1 Designations

Because the world is informal it is very hard to describe precisely. It is therefore necessary to lay a sound basis for description by saying as precisely as possible what phenomena are denoted by the formal terms in our requirements and domain properties descriptions. The appropriate tool is a set of *designations*. A designation gives a formal term, such as a predicate, and gives a—necessarily informal—rule for recognising instances of the phenomenon.

For example, in a genealogical system we may need this designation:

$$Mother(x, y) \approx x \textit{ is the mother of } y$$

Probably this is a very poor recognition rule: it leaves us in considerable doubt about what is included. Does it encompass adoptive mothers, surrogate mothers, stepmothers, foster mothers? Egg donors? Probably we must be more exact. Perhaps what we need is:

$$Mother(x, y) \approx x \textit{ is the human genetic mother of } y$$

Even this more conscientious attempt may be inadequate in a future world in which genetic engineering has become commonplace.

Adequate precision of the underlying designations is fundamental to the precision and intelligibility of the requirement and domain descriptions that rely on them. If it proves too hard to write a satisfactory recognition rule for phenomena of a chosen class, that chosen class should be rejected, and firmer ground should be sought elsewhere.

This harsh stipulation is less obstructive than it may seem at first. The designated terminology is intended for describing a particular part, or domain, of the problem world for a particular problem. As so often in software development, we may be tempted to multiply our difficulties a thousandfold by trying to treat the general case instead of focusing, as practical engineers, on the particular case in hand. The temptation must be resisted.

For example, in an inventory problem for the OfficeWorld Company, whose business is supplying office furniture, we may need to designate the entity class Chair. Perhaps we write this designation:

$$Chair(x) \approx x \textit{ is a single unit of furniture whose primary use is to provide seating for one person}$$

Philosophers have often cited 'chair' as an example of the irreducibly uncertain meaning of words in natural language. In the general case no designation of 'chair' can be adequate. Is a bar stool a chair? A bean bag? A sofa? A park bench? A motor car seat? A chaise longue? A shooting stick? These questions are impossibly difficult to answer: there are no right answers. But we do not have to answer them. The OfficeWorld Company has quite a small catalogue. It doesn't supply bar stools or park benches or bean bags. Our recognition rule is good enough for the case in hand.

7.4.2 *Using definitions*

Another factor mitigating the severity of the stipulation that designations must be precise is that the number of phenomenon classes to be designated usually turns out to be surprisingly small. Many useful terms do not denote distinctly observable phenomena at all, but must be *defined* on the basis of terms that do and of previously defined terms. For example:

$$\begin{aligned} &\mathit{Sibling}(a,b) \stackrel{\text{def}}{=} \\ &\quad a \neq b \wedge \exists\ p,q \bullet \mathit{Mother}(p,a) \wedge \mathit{Mother}(p,b) \\ &\qquad\qquad \wedge\ \mathit{Father}(q,a) \wedge \mathit{Father}(q,b) \end{aligned}$$

The difference between definition and designation is crucial. A designation introduces a fresh class of observations, and thus enlarges the scope of possible assertions about the world. A definition, by contrast, merely introduces more convenient terminology without increasing the expressive power at our disposal.

In an inventory problem, suppose that we have designated the event classes[3] *receive* and *issue*:

$$\begin{aligned} &\mathit{Receive}(e,q,t) \approx e \text{ is an event occurring at time } t \\ &\qquad\qquad\qquad \text{in which } q \text{ units of stock are received} \\ &\mathit{Issue}(e,q,t) \approx e \text{ is an event occurring at time } t \\ &\qquad\qquad\qquad \text{in which } q \text{ units of stock are issued} \end{aligned}$$

Then the definition:

$$\begin{aligned} &\mathit{ExpectedQuantity}(qty,tt) \stackrel{\text{def}}{=} \\ &\quad (\Sigma\ e \mid ((\mathit{Receive}(e,q,t) \vee \mathit{Issue}(e,-q,t)) \wedge t < tt) : q) = qty \end{aligned}$$

defines the predicate *ExpectedQuantity(qty,tt)* to mean that at time *tt* the number *qty* is equal to a certain sum. This sum is the total number of units received in *receive* events, minus the total number issued in *issue* events, taken over all events e occurring at any time t that is earlier than time *tt*. Being a definition, it says nothing at all about the world. By contrast, the designation and assertion:

[3]For uniformity, it is convenient to designate all formal terms as predicates. For any set of individuals, such as a class of events, the formal term in the designation denotes the characteristic predicate of the set.

$$InStock(qty, tt) \approx \textit{At time tt qty items are in the stock bin in the warehouse}$$

$$\forall\ qty, tt \bullet InStock(qty, tt) \Leftrightarrow (\Sigma e \mid ((Receive(e, q, t) \vee Issue(e, -q, t)) \wedge t < tt) : q) = qty$$

say that initially *InStock(0,t0)* and that subsequently stock changes only by the quantities issued and received. There is no theft, no evaporation and no spontaneous creation of stock. The definition of *ExpectedQuantity* expressed only a choice of terminology; the designation of *InStock*, combined with the accompanying assertion, expresses a falsifiable claim about the physical world.

7.4.3 *Distinguishing Definition From Description*

Many notations commonly used for description can also be used for definition, distinguishing the two uses by certain restrictions and by suitable syntactic conventions.

For example, it is often convenient to define terms for state components by giving a finite-state machine. Since mixing definition with description—like mixing indicative with optative—is very undesirable, the state-machine description should be empty *qua* description[4]. That is, in defining states it should place no constraint on the described sequence of events. Suppose, for example, that in some domain the sequence of events is

$$< a, b, a, b, a, \ldots >$$

and that we wish to define the state terms *After-a* and *After-b*. Figure 7.6 shows the definition: it avoids assuming that the sequence of events is as given above. *After-a* is defined to mean the state identified as state 2 in this state machine, and *After-b* is defined similarly. Of course, if the meanings are intended to include the clause "... and the given sequence of events has been followed so far", then a different definition is necessary.

7.5 Descriptions and models

An important aspect of description in software development is clarity in the distinction between a *description* and a *model*. Unfortunately, the word *model* is much overused and much misused. Its possible meanings[5] include:

- An *analytical model* of a domain: that is, a formal description from which further properties of the domain can be inferred. For example, a set of differential equations describing a country's economy, or a labelled transition diagram describing the behaviour of a vending machine.

[4] A term defined in a non-empty description is undefined whenever the description is false. It then becomes necessary either to use a three-valued logic or to prove at each of its occurrences that the term is well-defined.

[5] This distinction among the three kinds of model is due to Ackoff [1].

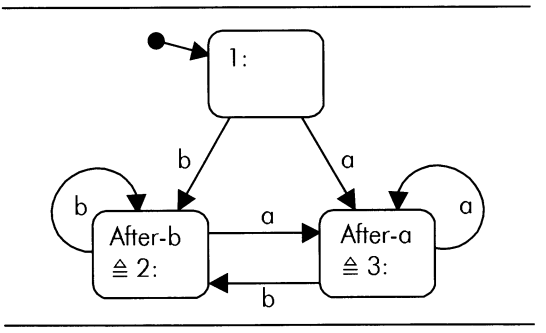


Figure 7.6. Defining states in a FSM

- An *iconic model* of a domain: that is, a representation that captures the appearance of the domain. For example, an artist's drawing of a proposed building.
- An *analogic model* of a domain: that is, another domain that can act as a surrogate for purposes of providing information. For example, a computer-driven wall display showing the layout of a rail network in the form of a graph, and the current train traffic on the network in the form of a blob for each train moving along the arcs of the graph.

Much difficulty arises from confusion between the first and third of these meanings. It is a common and necessary device in software development to introduce an analogic model, in the form of a database or other data structure, into the solution of an information problem or subproblem. Such an analogic model domain is to be regarded as an elaboration of a certain class of local variables of the machine. Descriptions of this model domain are often confused with descriptions of the domain for which it is a surrogate.

7.5.1 A model of a lift

A small hotel has an old and somewhat primitive lift. Now it is to be fitted with an information panel in the lobby, to show waiting guests where the lift is at any time and its current direction of travel, so that they will know how long they can expect to wait until it arrives.

The panel has a square lamp for each floor, to show that the lift is at the floor. In addition there are two arrow-shaped lamps to indicate the direction of travel. The panel display must be driven from a simple interface with the floor sensors of the lift. A floor sensor is on when the lift is within 6 inches of the rest position at the floor.

Figure 7.7 is the problem diagram. Here the customer manikin is replaced by the more impersonal dashed oval, representing the requirement. The requirement is that the lamp states of the lobby display (the phenomena *d*) should corre-

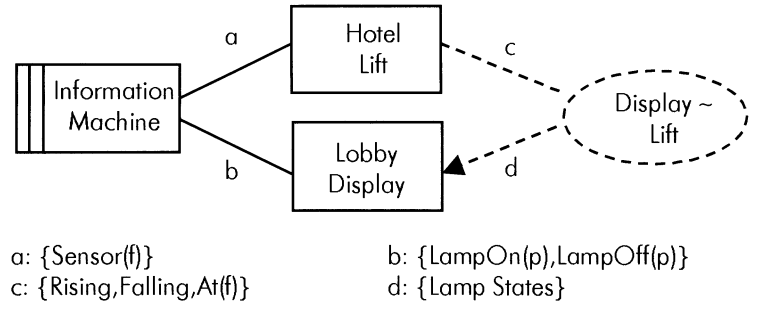


Figure 7.7. Lift position display problem

spond in a certain way to the states of the lift (the phenomena *c*). The arrow head indicates that the requirement constrains the display, but not the hotel lift itself.

This simple information problem presents a standard *concern* of problems of this class[6]. The information necessary to maintain the required correspondence is not available to the machine at the specification interface *a* at the moment when it is needed. The requirement phenomena include the current lift position and its current direction of travel; the specification phenomena include only the floor sensor states. To satisfy the requirement as well as possible, the machine must store information about the past history of the lift, and must interpret the current state and events in the light of this history.

The local phenomena of the machine in which this history is stored—perhaps in the form of program variables, or a data structure or small database—constitute an *analogic model domain*. If these local phenomena are not totally trivial it is desirable to decompose the original problem into two subproblems: one to build and maintain the model, and one to use the model in producing the lobby display. This problem decomposition is shown in Figure 7.8.

As the decomposed problem diagram shows clearly, the lift model and the hotel lift itself are disjoint domains, with no phenomena in common. In designing the lift model, the developer must devise model state phenomena *f* to correspond to the lift domain requirement phenomena *c*. These model phenomena might be called *MRising* and *MFalling*, corresponding to the lift states *Rising* and *Falling*, and *MAt(f)*, corresponding to *At(f)*.

The modelling subproblem is then to ensure that *MRising* holds in the model if and only if the lift is rising, that *MAt(f)* holds in the model if and only if the lift is at floor *f*, and so on. The model constructor operations—the phenomena *e*—will be invoked by the modelling machine when sensor state changes occur at its interface *a* with the hotel lift domain.

The display subproblem is much simpler: the display machine must ensure that the *Up* lamp is lit if and only if *MRising* holds; the floor lamp *f* is lit if and only if *MAt(f)* holds; and so on.

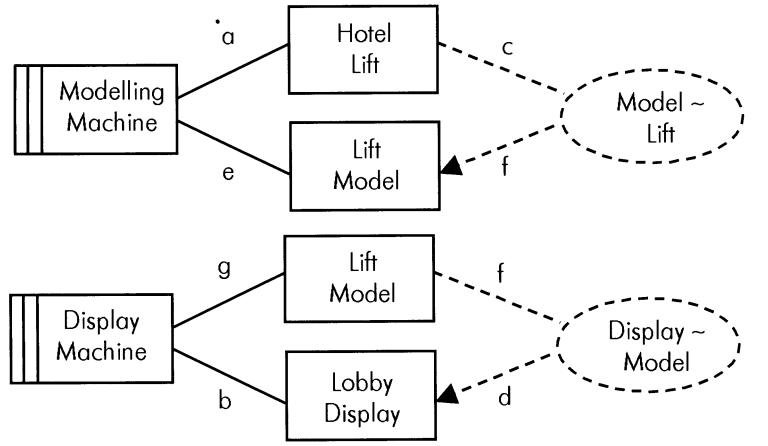


Figure 7.8. Lift position display problem decomposition

7.5.2 *The modelling relationship*

The desired relationship between a model domain and the domain it models is, in principle, simple. There should be a 1–1 correspondence between phenomena of the two domains and their values. For example, the lift has state phenomena *At(f)* for $f = 0 \ldots 8$ and the model has state phenomena *MAt(f)* for $f = 0 \ldots 8$. [6] For any f, *MAt(f)* should hold if and only if *At(f)* holds.

Because of this relationship it seems clear that a description that is true of one domain must be equally true of the other, with a suitable change of interpretation. For example, the description:

> "in any trace of values of *P(x)*, $0 \leq x \leq 8$ for each element of the trace, and adjacent values of x differ by at most 1."

is true of the lift domain if we take *P(x)* to mean *At(f)*, and must be true also of the model domain if we take *P(x)* to mean *MAt(f)*.

It therefore seems very attractive and economical to write only one description. In a further economy, even the work of writing the two interpretations can be eliminated by using the same names for phenomena in the lift and the corresponding phenomena in the model. Unfortunately, this is usually a false economy.

[6] Floor 0 is the lobby. In Europe floor 1 is the first above the ground floor; in the US the floors would be numbered 1 . . . 9.

Although almost universally attempted, both by practitioners and by researchers, it can work well only for an ideal model in which the desired relationship to the model domain is known to hold; practical models are almost never ideal.

7.5.3 *Practical models*

The lift domain phenomenon *At(f)* means that the lift is closer to floor f than to any other floor. However, it is not possible to maintain a precise correspondence between *At(f)* and the model phenomenon *MAt(f)*, because the specification state phenomena *Sensor(f)* do not convey enough information. The best that can be done is, perhaps, to specify the modelling machine so that *MAt(f)* is true if and only if *Sensor(f)* is the sensor that is on or was most recently on. So the correspondence between *At(f)* and *MAt(f)* is very imperfect. When the lift travels from floor 1 to floor 2, *MAt(1)* remains true even when the lift is six inches from the floor 2 home position and the state *Sensor(2)* is just about to become true.

The *Rising* and *Falling* phenomena are even harder to deal with. Once again, the modelling machine has access only to the *Sensor(f)* phenomena, and must maintain the model phenomena *MRising* and *MFalling* from the information they provide. Initially the lift may be considered to be *Rising*, because from the Lobby it can go only upwards; subsequently, when it reaches floor 8 (or floor 0 again) it must reverse direction. But it may also reverse direction at an intermediate floor, provided that it makes a service visit there and does not simply pass it without stopping.

Investigation of the lift domain shows that on a service visit the floor sensor is on for at least 4.8 seconds, allowing time for the doors to open and close. Passing a floor takes no more than 1 second. The model phenomena *MRising* and *MFalling* will be maintained as follows:

- Initially: $MRising \wedge \neg MFalling$
- Whenever *MAt(n+1)* becomes true when *MAt(n)* was previously true, for $(n = 0 \ldots 6)$: $MRising \wedge \neg MFalling$
- Whenever *MAt(8)* becomes true when *MAt(7)* was previously true: $MFalling \wedge \neg MRising$
- Whenever *MAt(n-1)* becomes true when *MAt(n)* was previously true for $(n = 2 \ldots 8)$: $MFalling \wedge \neg MRising$
- Whenever *MAt(0)* becomes true when *MAt(1)* was previously true: $MRising \wedge \neg MFalling$
- Whenever *MAt(n)* has been true for 2 seconds[7] continuously, for $(n = 1 \ldots 7)$: $\neg MFalling \wedge \neg MRising$

[7] A compromise between the limits of 1.0 and 4.8 seconds, affording an early but reasonably reliable presumption that the lift has stopped to service the floor.

These practical choices represent unavoidable departures from exact correspondence between the model and the lift domain. For example, during the first two seconds of a service visit either *MRising* or *MFalling* is true, although neither *Rising* nor *Falling* is true. Also, when the lift has reversed direction at an intermediate floor but has not yet reached another floor, either *Rising* or *Falling* is true, but neither *MRising* nor *MFalling* is true. Speaking anthropomorphically, we might say that the modelling machine is waiting to discover whether the next floor arrival will invite the inference of upwards or downwards travel.

7.5.4 *Describing the model and modelled domains*

Other factors that may prevent exact correspondence in a practical model include errors and delays in the interface between the modelling machine and the modelled domain, and the approximation of continuous by discrete phenomena. A further factor is the need to model the imperfection of the model itself. For example, NULL values are often used in relational databases to model the absence of information: a NULL value in a *date-of-birth* column indicates only that the date of birth is unknown. In the presence of such discrepancies it may still be possible to economise by using the same basic description for both domains and noting the discrepancies explicitly.

Another factor militating against a single description is that a model domain itself usually has additional phenomena that correspond to nothing in the modelled domain. The source of these phenomena is the underlying implementation of the model. A relational database, for example, usually has delete operations to conserve space, indexes to speed access to particular elements of the model, and ordering of tuples within relational tables to speed *select* and *join* operations. These discrepancies between the model and modelled domains can sometimes be regarded as no more than the difference between abstract and concrete views of the model. Introduction of the additional model phenomena is a refinement: the resulting implementation satisfies the model's abstract specification. This view applies easily to the introduction of tuple ordering and of indexing. It is less clear that it can apply to record deletion.

The use of only one description for the two domains fails most notably when the modelled domain has phenomena that do not and can not appear in the model. For example, the lift domain has the moving and stationary states of the lift car, and the opening and closing of the lift doors during a service visit to a floor. These phenomena can not appear in the model because there is not enough evidence of them in the shared phenomena of the specification interface. They are completely hidden from the modelling machine, and can enter into the model only in a most attenuated form—the choices based on assumptions about them. But they must still appear in any careful description of the significant domain properties.

In sum, therefore, it is essential to recognise that a modelled domain and its model are two distinct subjects for description. Confusion of the two results in importing distracting irrelevancies and restrictions into the problem domain description. For example, in UML [10] descriptions of a business domain must be

based on irrelevant programming concepts, such as *attributes*, *visibility*, *interfaces* and *operations*, taken from object-oriented languages such as C++ and Smalltalk. At the same time, UML notations provide no way of describing the syntax of a lexical problem domain, other than by describing a program to parse it.

This vital distinction between the model and the modelled domain is difficult to bear in mind if the verb *model* is used where the verb *describe* would do as well or better. The claim "We are modelling the lift domain" invites the interpretation "We are describing the lift domain", when often it means in fact "We are not bothering to describe the lift domain: instead we are describing a domain that purports to be an analogical model of it."

7.6 Problem decomposition and description structures

Realistic problems must be decomposed into simpler subproblems. Almost always, the subproblems are related by having problem domains in common: that is, they are not about disjoint parts of the world. The common problem domains must, in general, be differently viewed and differently described in the different subproblems. This section gives two small illustrations of this effect of problem decomposition.

7.6.1 An auditing subproblem

A small sluice, with a rising and falling gate, is used in a simple irrigation system. A control computer is to be programmed to raise and lower the sluice gate: the gate is to be open for ten minutes in each hour, and otherwise shut.

The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by *Clockwise*, *Anticlockwise*, *On* and *Off* pulses. There are sensors at the top and bottom of the gate travel; at the top the gate is fully open, at the bottom it is fully shut. The connection to the computer consists of four pulse lines for motor control and two status lines for the gate sensors.

The requirement phenomena are the gate states *Open* and *Shut*. The specification phenomena are the motor control pulses, and the states of the *Top* and *Bottom* sensors. A mechanism of this kind moves slowly and has little inertia, so a specification of the machine behaviour to satisfy the requirement is simple and easily developed. Essentially, the gate can be opened by setting the motor to run in the appropriate sense and stopping it when the *Top* sensor goes on; it can be closed similarly, stopping the motor when the *Bottom* sensor goes on.

The domain properties on which the machine must rely include:

- The behaviour of the motor unit in changing its state in response to externally caused motor control pulses;

- the behaviour of the mechanical parts of the sluice that govern how the gate moves vertically, rising and falling according to whether the motor is stopped or rotating clockwise or anticlockwise;
- the relationship between the gate's vertical position and its Open and Shut states; and
- the relationship between the gate's vertical position and the states of the Top and Bottom sensors.

To develop a specification of the control machine it is necessary to investigate and describe these domain properties explicitly.

7.6.2 *Fruitful contradiction*

Being physical devices, the sluice gate and its motor, on whose properties the control machine is relying, are not so reliable as we might wish. Power cables can be cut; motor windings burn out; insulation can be worn away or eaten by rodents; screws rust and corrode; pinions become loose on their shafts; branches and other debris can become jammed in the gate, preventing it from closing. The behaviour of the control computer should take account of these possibilities—at least to the extent of stopping the motor when something has clearly gone wrong.

Possible evidences of failure, detectable at the specification interface, include:

- the *Top* and *Bottom* sensors are on simultaneously;
- the motor has been set to raise the gate for more than m seconds but the *Bottom* sensor is still on;
- the motor has been set to lower the gate for more than n seconds but the *Top* sensor is still on;
- the motor has been set to raise the gate for more than p seconds but the *Top* sensor is not yet on;
- the motor has been set to lower the gate for more than q seconds but the *Bottom* sensor is not yet on.

Detecting these possible failures should be treated as a separate subproblem, of a class that we may call *Auditing problems*. The machine in this auditing subproblem runs concurrently with the machine in the basic control problem. The two subproblem machines are connected: the control machine, on detecting a failure, causes a signal in response to which the control machine turns the motor off and keeps it off thereafter.

The particular interest of this problem is that in a certain sense the domain property description of the auditing subproblem contradicts the description on which the solution of the control subproblem must rely. The indicative domain description for the control subproblem asserts that when the motor is set in such-and-such a state the gate will reach its *Open* state within p seconds; but

the description for the auditing problem contradicts this assertion by explicitly showing the possibility of failure.

At a syntactic level, this conflict can be resolved by merging the two descriptions to give a single consistent description that accommodates both the correct and the failing behaviour of the gate mechanism. This merged description might then be used for the control subproblem, the auditing subproblem being embedded in the control subproblem as a collection of local behaviour variants. But this merging is not a wise strategy. It is better to solve the control subproblem in the context of explicit appropriate assumptions about the domain properties, leaving the complications of the possible failures for a separate concern and a separate subproblem.

7.6.3 *An identities concern*

In the lift display problem it was necessary to pay careful attention to the gap between the requirement phenomena (the *At(f), Rising and Falling* states) and the specification phenomena (the *Sensor(f)* states) of the lift domain. But we were not at all careful about another phenomenological concern in the problem. We resorted—naturally enough—to the standard mathematical practice of indexing multiple phenomena: we wrote f for the identifier of a floor, and used that identifier freely in our informal discussion and—by implication—in our descriptions.

This was too casual. The use of 'abstract indexes' in this way is sometimes an abstraction too far: it throws out an important baby along with the bathwater. Essentially, it distracts the developer from recognising an important class of development concern: an *Identities* concern [6]. The potential importance of this concern can be seen from an anecdote in Peter Neumann's book about computer risks [7]:

> "A British Midland Boeing 737-400 crashed at Kegworth in the United Kingdom, killing 47 and injuring 74 seriously. The right engine had been erroneously shut off in response to smoke and excessive vibration that was in reality due to a fan-blade failure in the left engine. The screen-based 'glass cockpit' and the procedures for crew training were questioned. Cross-wiring, which was suspected—but not definitively confirmed—was subsequently detected in the warning systems of 30 similar aircraft."

'Cross-wiring' is the hardware manifestation of an archetypal failure in treating an identities concern.

7.6.4 *Patient monitoring*

In the well-known Patient Monitoring problem [9] the machine is required to monitor temperature and other vital factors of intensive-care patients according to parameters specified by medical staff. The physical interface between the machine

and the problem world of the intensive-care patients is essentially restricted to the shared register values of the analog-digital sensor devices attached to the patients. A significant concern in this problem is therefore to associate these shared registers correctly with the individual patients, and to describe how this association is realised in the problem domain. The complete chain of associations is this:

- each patient has a name, used by the medical staff in specifying the parameters of monitoring for the patient;
- each patient is physically attached to one or more analog-digital devices;
- each device is plugged into a port of the machine through which its internal register is shared by the machine;
- each port of the machine has a unique name.

To perform the monitoring as required, the machine must have access to a data structure representing these chains of associations. This data structure is a very specialised restricted *identities model* of the problem world of patients, devices and medical staff. It is, of course, quite distinct from any model of the patients that may be needed for managing the frequency of their monitoring and for detecting patterns in the values of their vital factors. The two models may be merged in an eventual joint implementation of the machines of the constituent subproblems, but they must be kept distinct in the earlier stages of the development process.

There is a further concern. Since neither the population of patients, nor the set of monitoring devices deemed necessary for each one, is constant, there must be an editing process in which the identities model data structure is created and changed. Concurrent access to this data structure by the monitoring and modifying processes therefore raises concerns of mutual exclusion and process scheduling. An excessively abstract view of the problem context will miss the existence of the data structure, and with it these important concerns and their impact on the Patient Monitoring system.

7.7 The scope of software development

The description concerns raised in this paper are primarily concerns about describing the problem world rather than designing the software to be executed by the machine. It's natural to ask again whether these description concerns are really the business of software developers at all. Perhaps the specification firewall does, after all, divide the business of software development from the business of the application domain expert.

Barry Boehm paints a vivid picture of software developers anxious to remain behind the firewall and not to encroach on application domain territory [2]:

> "I observed the social consequences of this approach in several aerospace system-architecture-definition meetings ("Integrating

> Software Engineering and System Engineering", Journal of INCOSE, pages 61-67, January 1994). While the hardware and systems engineers sat around the table discussing their previous system architectures, the software engineers sat on the side, waiting for someone to give them a precise specification they could turn into code."

It's clearly true that software developers can not and should not try to be experts in all application domains. For example, in a problem to control road traffic at a very complex intersection it must be the traffic engineer's responsibility to determine and analyse the patterns of incoming traffic, to design the traffic flows through the intersection, and to balance the conflicting needs of the different pedestrian and vehicle users. Software developers are not traffic engineers. But this is far from the whole answer.

There are several reasons why a large part of our responsibility must lie outside the computer, beyond the specification firewall. Here we will mention only two of them. First, the specification firewall usually cuts the development project along a line that makes the programming task unintelligibly arbitrary when viewed purely from the machine side: effectively, pure specifications are meaningless. And second, having created the technology that spawns huge discrete complexity in the problem domain, we have a moral obligation to contribute to mastering that complexity.

7.7.1 Meaningless specifications

In the problem of controlling traffic at a complex road intersection the pure specification is an I/O relation. Its domain is the set of possible traces of clock ticks and input signals at the computer's ports; its range is a set of corresponding traces of output signals. These trace sets may be characterised more or less elegantly, but, however described, they are strictly confined to these signals. The specification alphabet will be something like this—

$$\{clocktick,\ outsignal\_X1FF,\ ...,\ insignal\_X207,\ ...\ \}$$

— where the event classes in the alphabet are events occurring in the hardware I/O interface of the computer. Nothing is said about lights or push buttons, about the layout of the intersection, or about vehicles and pedestrians. These are all private phenomena of the problem domain, hidden from the machine because they are not shared at the specification interface.

It's clear that such a specification is unintelligible. A small improvement can be achieved by naming the signals at the specification interface to indicate the corresponding lights and buttons—

$$\{clocktick,\ outsignal\_red27,\ ...,\ insignal\_button8,\ ...\ \}$$

—but the improvement is very small. Further improvement would need additional descriptions, showing the layout of the intersection and the positions of the lights. Then the domain properties of vehicles and pedestrians, existing and desired

traffic flows, and everything else necessary to justify and clarify the otherwise impenetrable machine behaviour specification.

In short, the machine behaviour specification makes sense only in the larger context of the problem; and the problem is not located at the specification interface. If we restrict our work to developing software to meet given formal specifications, most of what we do will make no sense to us. We will be deprived of the intuitive understanding of the customer's problem that is essential both as a stimulus to creativity in program design and as a sanity check on the program we write.

7.7.2 *Discrete complexity*

Computers frequently introduce an unprecedented behavioural complexity into problem worlds with which they interact. This behavioural complexity arises naturally from the complexity of the software itself, and from its interplay with the causal, human and lexical properties of the problem domains.

In older systems behavioural complexity was kept under control by three factors. First, the software itself—whether in the form of a computer program or an administrators' procedure manual—was usually smaller and simpler than today's software by more than one order of magnitude. Second, there was neither the possibility nor the ambition of integrating distinct systems, and so bringing about an exponential increase in their combined behavioural complexity. Third, almost every system, whether a 'data-processing' or a 'control' system, relied explicitly on human cooperation and intervention. When inconvenient and absurd results emerged, some human operator had the opportunity, the skill and the authority to intervene and overrule the computer.

In many application areas we have gradually lost all of these safeguards. The ambitions of software developers increase to keep pace with the available resources of computational power and space. Systems are becoming more integrated, or, at least, more interdependent. And it is increasingly common to find levels of automation—as in flight control systems—that preclude human intervention to correct errors in software design or specification.

A large part of the responsibility for dealing with the resulting increased behavioural complexity must lie with computer scientists and software developers, if only because no other discipline has tools to master it. We can not discharge this responsibility by mastering complexity only in software: we must play a major role in mastering the resulting complexity in the problem world outside the computer.

7.8 Acknowledgements

Many of the ideas presented here have been the subject of joint work over a period of several years with Pamela Zave. They have also been discussed at length on

many occasions with Daniel Jackson. This paper has been much improved by his comments.

References

[1] R L Ackoff. *Scientific Method: Optimizing Applied Research Decisions*; Wiley, 1962.

[2] Barry W Boehm. Unifying Software Engineering and Systems Engineering; IEEE Computer Volume 33 Number 3, pages 114-116, March 2000.

[3] Edsger W Dijkstra. On the Cruelty of Really Teaching Computer Science; Communications of the ACM Volume 32 Number 12, page 1414, December 1989.

[4] Carl A Gunter, Elsa L Gunter, Michael Jackson, and Pamela Zave. A Reference Model for Requirements and Specifications; Proceedings of ICRE 2000, Chicago Ill, USA; reprinted in IEEE Software Volume 17 Number 3, pages 37-43, May/June 2000.

[5] David Harel. Statecharts: A visual formalism for complex systems; Science of Computer Programming 8, pages 231-274, 1987.

[6] Michael Jackson. *Problem Frames: Analysing and Structuring Software Development Problems*; Addison-Wesley, 2000.

[7] Peter G Neumann. Computer-Related Risks; Addison-Wesley, 1995, pages 44–45.

[8] W L Scherlis. responding to E W Dijkstra "On the Cruelty of Really Teaching Computing Science"; Communications of the ACM Volume 32 Number 12, page 1407, December 1989.

[9] W P Stevens, G J Myers, and L L Constantine; Structured Design. IBM Systems Journal Volume 13 Number 2, pages 115-139, 1974. Reprinted in Tutorial on Sofware Design Techniques; Peter Freeman and Anthony I Wasserman eds, pages 328-352, IEEE Computer Society Press, 4th edition 1983.

[10] James Rumbaugh, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Manual*; Addison-Wesley Longman 1999.

[11] Hermann Weyl. The Mathematical Way of Thinking; address given at the Bicentennial Conference at the University of Pennsylvania, 1940.

8

Modelling architectures for dynamic systems

Peter Henderson

Abstract

A dynamic system is one that changes its configuration as it runs. It is a system into which we can drop new components that then cooperate with the existing ones. We are concerned with formally defining architectures for such systems and with realistically validating designs for applications that run on those architectures. We describe a generic architecture based on the familiar registry services of CORBA, DCOM and Jini. We illustrate this architecture by formally describing a simple point-of-sale system built according to this architecture. We then look at the sorts of global properties that a designer of applications would wish a robust system to have and discuss variations on the architecture which make validation of applications more practical.

8.1 Introduction

The advent of ubiquitous computing, where everything is connected to everything else, has created a new challenge for Software Engineering and for Software Reuse in particular. It is now increasingly important that software components are designed for a life of constant change and frequent reuse.

With everything connected to everything else, systems are necessarily subject to dynamic change. You can't stop the whole world just to plug in a new component. Components need to be as nearly plug-and-play as possible. Flexible architectures such as Jini [35], are making the evolution of dynamic systems possible. The question is, how do you design for such architectures and how do you design components which will survive a lifetime of use and reuse even though their environment and the expectations which their users have of them, are constantly changing?

A. McIver et al. (eds.), *Programming Methodology*

8.1.1 *Dynamic systems*

In [16] dynamic systems are described as being built from components and having the property that a new component could be added to a running system at any time and the system would embrace its contribution without having to stop. It is the requirement that the system can evolve by accretion, without ever having to stop, that leads us to call the system"dynamic". The consequences for component reuse are dramatic. Components will be reused in ways that were not imagined by their original designers. In [16] we addressed the issue of who would be to blame if the consequence of adding a new component was that something broke.

In this paper, we formally describe some of the issues which arise for the developer of dynamic systems, not least of all the evolution of functionality in an incremental way. We do this by introducing an elementary architecture modelling language, ARC [17], which allows for experimentation with alternative architectural designs and for the validation of these designs using state-space search. In particular, ARC models can be compiled to run on the SPIN model-checker [21]. The ARC modelling paradigm, it is conjectured, is simple enough to allow many experiments to be performed quickly with modest cost and yet powerful enough to describe a range of practical architectures and generate valuable insight into their properties.

8.1.2 *The context of constant change*

We are concerned with dynamic systems in the context of constant change, where the system supports a business process which is constantly needing to be changed to match the rapidly moving marketplace. We wish to explore architectures which will allow the incremental enhancement of the system without having to be stopped for upgrade. Consequently we are concerned with issues of reconfigurability, where new components can be added to the system which then embraces the new services which they offer. We are less concerned with the removal of old components in that we anticipate architectures which will allow such components to gradually become obsolete and eventually retire.

However we are concerned with issues of survival. We will articulate scenarios in which the system can survive despite the fact that some components fail. One way of looking at this issue is to characterise the interaction between a system and its environment as a two-person game [1]. The moves made by the system are to maximise the number of components which can operate. The moves of the environment are to damage key components with the intention of preventing as many components as possible from operating successfully. We show how our modelling paradigm lends itself to this metaphor.

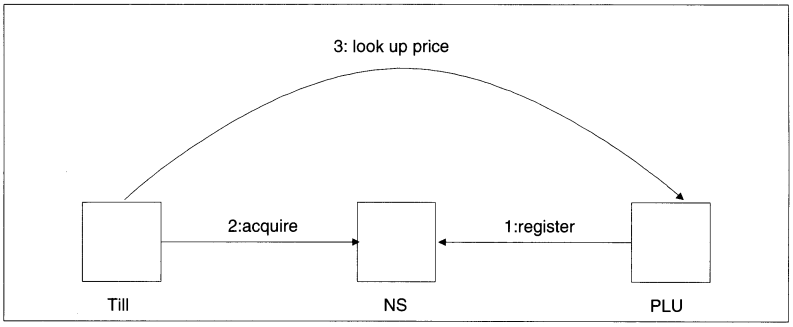


Figure 8.1. A UML Collaboration Diagram

8.2 Models of dynamic systems

In order to be able to make precise statements about alternative architectural proposals we need to use a language which has a precise meaning and which operates at a level of abstraction appropriate to the kinds of reasoning which we wish to perform. There is a choice of paradigms. Many architecture modelling languages base themselves on a message-passing, process-oriented view. Examples are Wright [2, 3, 32], Darwin [27, 28] and more recently FSP/LTSA [29]. Others, such as Rapide [24, 25, 26] take an event-based approach where events are specified by condition-action rules. This is the approach we will take in ARC. Architecture languages concern themselves with structure and behaviour [36]. We are, of course, concerned with both here. But, in a dynamic system, structure is dynamic and so structure merges into behaviour.

8.2.1 The ARC notation

Our conjecture is that our modelling language is appropriate to the design of reusable components for dynamic systems, because it operates at a level-of-abstraction that allows reasonably large systems to be modelled, but still allows a useful degree of validation of the models in a cost-effective manner.

The modelling paradigm is influenced by the collaboration diagrams of UML [9]. These diagrams are a variety of Object Interaction Diagram, where the behaviour of a (scenario) from a system is depicted. In collaboration diagrams (see Figure 8.1), objects (rather than classes) are shown along with the messages which pass between them. The objects are usually boxes and the messages are arrows. The sequencing is shown by numbering the messages. The reader can then follow a scenario by reading the messages in order. Designers use such a diagram to first convince themselves, then others, that they have a valid behaviour.

Figure 8.1 shows an example of a UML collaboration diagram and also serves to introduce the example which we will use throughout this paper both to introduce ARC and to consider alternative architectures. Figure 8.1 shows an EPOS (Electronic Point-of-Sale) system. It shows three objects: *Till* is the (hardware and

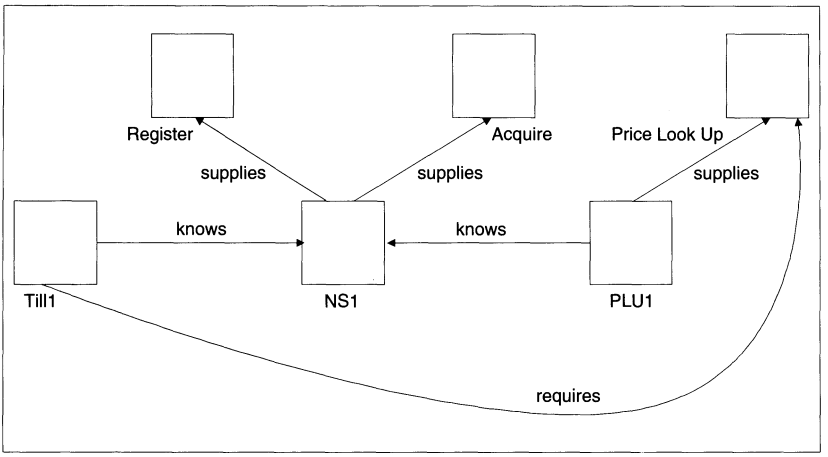


Figure 8.2. An ARC Diagram

software) component where customers' purchases are scanned and paid for; *NS* is the Name Server which (in this client-server architecture) acts as the registry for objects offering and requiring services; and *PLU* is the Price Look Up component which offers the service of supplying prices for purchased items.

The scenario depicted in the UML diagram of Figure 8.1 shows a sequence of three operations: first the *PLU* invokes a *register* operation on *NS* to register its availability as a supplier of the *PriceLookUp* service; then *Till* acquires from *NS* the name (*PLU*) of the supplier of this service; finally the *Till* invokes a look-up-price operation on the *PLU*.

Although inspired by the collaboration diagrams of UML, our paradigm uses a slightly higher level of abstraction. Rather than show messages, we show relationships or associations, between objects. The implication is that, if an appropriate relationship exists between two objects, one may have access to the services of the other. We will illustrate this in detail in what follows. The behavioural aspect of the system that we will then be able to illustrate is the configuration and reconfiguration of those relationships as, in a dynamic system, components first join and then acquire relationships with other components which they intend to use.

In ARC we use the terms component and object interchangeably. We think of objects or components as having state, behaviour and autonomy. That is, they are active, as if they were servers or clients.

Figure 8.2 shows the state of a system in ARC diagrammatic form. There are six components (*Till*1, *PLU*1, *NS*1, *Register*, *Acquire* and *PriceLookUp*) and three relationships (*knows*, *supplies* and *requires*). The diagram depicts the state in which, among other things, *Till*1 requires *PriceLookUp*, *PLU*1 supplies *PriceLookUp* and while *Till*1 does not yet know of *PLU*1, it does know *NS*1 which in turn knows *PLU*1. *NS*1 is, of course, the Name Server in this distributed system. *Till*1 will ask *NS*1 for the name of a component which supplies *PriceLookUp*, and as a con-

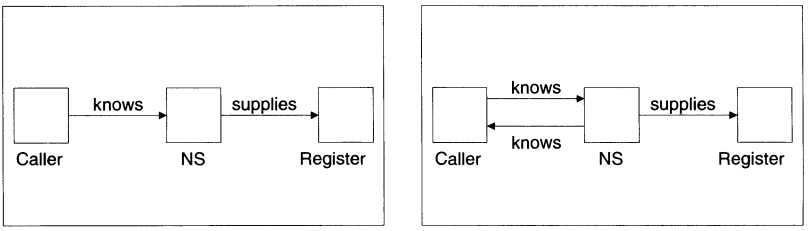


Figure 8.3. The Register Action

sequence the configuration will change dynamically to add the relationship that *Till*1 knows *PLU*1.

In practice, the ARC diagrams become too cluttered to express realistic scenarios, so we use them only to illustrate partial states. They come into their own when a group of engineers are designing a new solution on a whiteboard, because changes to the solution are quickly understood by all the participants. But for formal presentation we use a textual form to capture the full meaning of any situation. That is what we shall use here.

The ARC textual notation is based on logic, and in particular on the use of logic in Prolog strongly influenced by conceptual modelling [4]. A similar use of notation has recently been adopted in Alloy [22].

The state depicted in Figure 8.1 would be expressed by the conjunction

$$knows(Till1, NS1) \& knows(NS1, PLU1) \& supplies(NS1, Register) \& \\ supplies(NS1, Acquire) \& supplies(PLU1, PriceLookUp) \& \\ requires(Till1, PriceLookUp)$$

This is how we describe a state, as a conjunction of (usually) binary relations. Next we describe Actions which will enable us to move from state to state. We use Condition-Action rules. Figure 8.3 shows a diagrammatic form of a rule, with the condition to be met depicted in the left-hand box and the state to be moved to depicted in the right hand box. What the Action in Figure 8.3 depicts is the act of registering with a Name Server *NS*. The *Caller* knows *NS* initially, and in the eventual state *NS* knows the *Caller*.

Putting this Action into textual form, we have

$$register(Caller, NS) = \\ knows(Caller, NS) \& supplies(NS, Register) \\ \rightarrow +knows(NS, Caller)$$

Thus we define actions, which we give names to, which have a side-effect of adding and deleting relationships. Actions have parameters. The addition and deletion of relationships is denoted by $+$ and $-$ signs just in front of the relationship name. An example of relationship-deletion would be the reverse of the Register operation, shown in Figure 8.4.

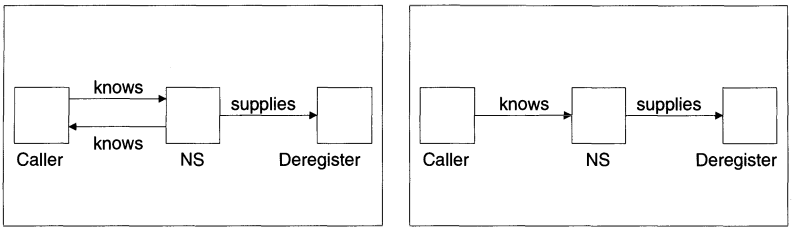


Figure 8.4. The deRegister Action

$$deregister(Caller, NS) =$$
$$knows(Caller, NS)\&knows(NS, Caller)\&supplies(NS, Deregister)$$
$$\rightarrow -knows(NS, Caller)$$

A more complex Action is shown in Figure 8.5. This is the Acquire action that also involves a Name Server. It is the way that components obtain knowledge of others that supply services which they require. The Action depicted in Figure 8.3 has the meaning

$$acquire(Caller, NS, Service) = \textbf{exists}\ Object.$$
$$knows(Caller, NS)\&knows(NS, Object)\&supplies(NS, Acquire)\&$$
$$supplies(Object, Service)\&requires(Caller, Service)$$
$$\rightarrow +knows(Caller, Object)$$

You can see how this would match a state in which, for example

$$knows(Till1, NS1)\&knows(NS1, PLU1)$$

and $NS1$, $Till1$ and $PLU1$ are as previously described. So if this action is performed on that state, we would move to a state in which $Till1$ knows $PLU1$, an obviously desirable state of affairs.

This is mostly all there is to ARC. In the formula for *acquire*, the component *Object* has a particular status. It is not a parameter of the operation. It is a local variable, which can match any component that satisfies the relational structure in which it is involved. In logical terms, it is existentially quantified with scope the condition and action parts of the rule. In addition to the logical structures which we have exhibited here, we allow explicit negation, disjunction, implication and universal quantification. Negation could have been used in the formula for *acquire*, for example, to strengthen the condition in such a way as to ensure that the *Caller* did not acquire something which supplied a service which was already supplied by some component which it already knew (add $\neg(knows(Caller, Object1)\&supplies(Object1, Service))$ to the condition).

8.2.2 *Validation of models*

The models we have made are particular forms of finite state machines, with the states represented by a particular edge-coloured graph, where the nodes are Components, the edges are Relationships and the colours are the actual Relations.

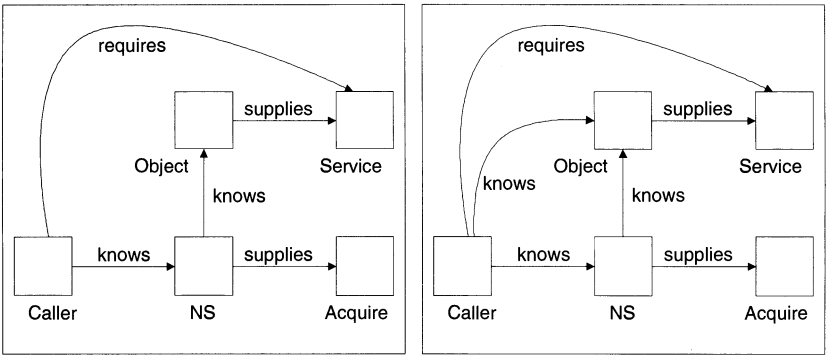


Figure 8.5. The Acquire Action

Transitions between states are accomplished by Actions, which add and remove edges from the graph.

Consequently, validation of the models can be accomplished by various finite-state-machine checking capabilities. In particular, model checking can be used [6, 13, 21, 22, 29]. It is also straightforward to build animations of the models, and this is an effective way for a group of engineers to persuade themselves that their design is complete and consistent, and to look at the consequences, for example, of component failure.

As an example of validating a model, consider the example we have used throughout this introduction to ARC. In the simplest scenario, we might begin in the state

$$knows(Till1, NS1)\&knows(PLU1, NS1)\&supplies(NS1, Acquire)\&$$
$$supplies(NS1, Register)\&requires(Till1, PriceLookUp)\&$$
$$supplies(PLU1, PriceLookUp)$$

Now the reader will realise that the sequence of Actions

$$register(PLU1, NS1);\ acquire(Till1, NS1, PriceLookUp)$$

will move us to a situation where, in addition to the above state, we also have the following relationships

$$knows(NS1, PLU1)\&knows(Till1, PLU1)$$

Figure 8.6 shows the ARC validation tool which supports various types of state space search. The model developed in this section has been presented to the tool which displays three panels each containing a list. The user chooses to instantiate a small number of objects of each type, in this case one Name Server (*NS*1), two Tills (*Till*1 and *Till*2) and two PLUs (*PLU*1 and *PLU*2). The list of Actions displays (in alphabetical order) just those which are effective in that their condition part evaluates to true and their action part will actually change the state. The State list comprises terms in the conjunction which describes part of the (graph representing the) current state in which we have expressed an interest. Selecting

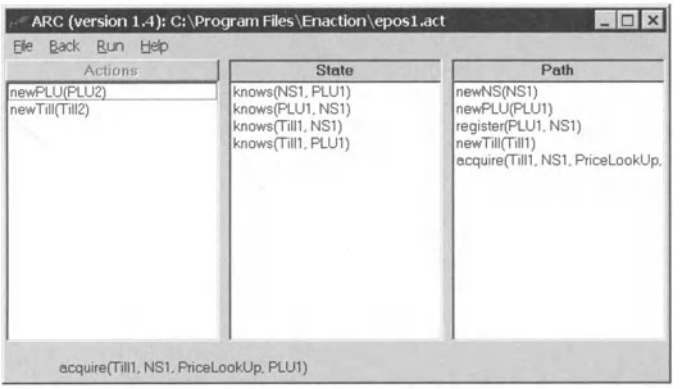


Figure 8.6. The ARC evaluation tool

an action from the Action list applies it to the current state and hence progresses to the next state. The actions which have been invoked so far are shown (this time in sequential order) in the Path list. Various methods of searching the state space are provided.

ARC models can also be translated into Promela [21] in a very straightforward way and executed on the SPIN model-checker. Every relationship of the form $rel(obj1, obj2)$ (that is, every edge potentially in the graph representing the state) is represented as a Promela (bit) variable. Adding the relationship to the state corresponds to setting this variable to true, removing the relationship to setting it to false. Experiments have shown that ARC and SPIN generate the same state machines. Translating to SPIN has the advantage that we can make use of SPIN's mature model-checking capabilities, particularly its performance and its ability to check temporal properties expessed in LTL.

8.3 Architectures for reuse

The client-server architecture which we have used to illustrate our modelling language is an example of a flexible architecture designed for reuse of (Services supplied by) Servers. We have shown that it is able to support the elementary kind of reconfiguration required by the initial marriage of clients to servers. And we can show that it is tolerant of some types of failure and incremental change.

8.3.1 Survival

Consider the following kind of attack on the client-server architecture

$$
\begin{aligned}
&break(Object) = \\
&\quad \textbf{all}\ Service.supplies(Object, Service) \rightarrow \\
&\quad -supplies(Object, Service)
\end{aligned}
$$

Clearly, if we execute $break(PLU1)$ then this can be fixed by the system performing $acquire(Till1, NS1, PriceLookUp)$ again, which will locate $PLU2$, assuming it has registered with $NS1$.

Breaking $NS1$ with $break(NS1)$ is a little more serious, but not immediately. The system continues to function. It runs into trouble after $break(PLU1)$, for now $Till1$ cannot find $PLU2$. Unless of course $Till1$ had had the foresight to prepare for this eventuality by acquiring $PLU2$ even though, having $PLU1$, it didn't strictly need it. But of course, eventually the loss of $NS1$ is more serious.

The semantics of creation of new objects gives a telling insight

$$newPLU(PLU) =$$
$$true \rightarrow +knows(PLU, NS1);\ +supplies(PLU, PriceLookUp)$$

The other object creating definitions are similar. In this system, every new object comes into existence knowing the name of the same single registry $NS1$. When $NS1$ dies, the system can only deteriorate.

But even here, there is a solution. It has to do with where the initiative for performing actions is assigned. In the model, we have purposely not assigned the actions to the objects. But we should, because we want objects to be autonomous and active. The reason we haven't done this in the model is that we don't want to decide early either who has the initiative or what their goal is. But suppose that all objects know how to invoke *acquire* on objects which supply that service and that their objective (goal) is to acquire as many instances of the objects which supply services which they may be able to use. Then, if $NS2$ is created and registers with $NS1$, all the objects which know $NS1$ can now acquire $NS2$ and thus increase their chances of survival.

Note that formally our architecture requires one of two changes. Either we weaken the condition on *acquire* to omit $requires(Caller, Service)$ so that objects can acquire anything, whether they need it or not. Or, we strengthen the requirements of all objects to include $+requires(Object, Acquire)$.

8.3.2 *Incremental change*

This leads to another consideration of how systems evolve, rather than just survive. Suppose that we plan to upgrade our EPOS system with a new service. For the sake of argument let us assume it is a Loyalty scheme whereby the system identifies the customer at the point-of-sale and offers bespoke services (such as targeted coupons). We will run through one scenario which illustrates this happening.

First we have a new Loyalty Server,

$$newLS(LS) =$$
$$true \rightarrow +knows(LS, NS1);\ +supplies(LS, Loyalty)$$

Then we have a new Till

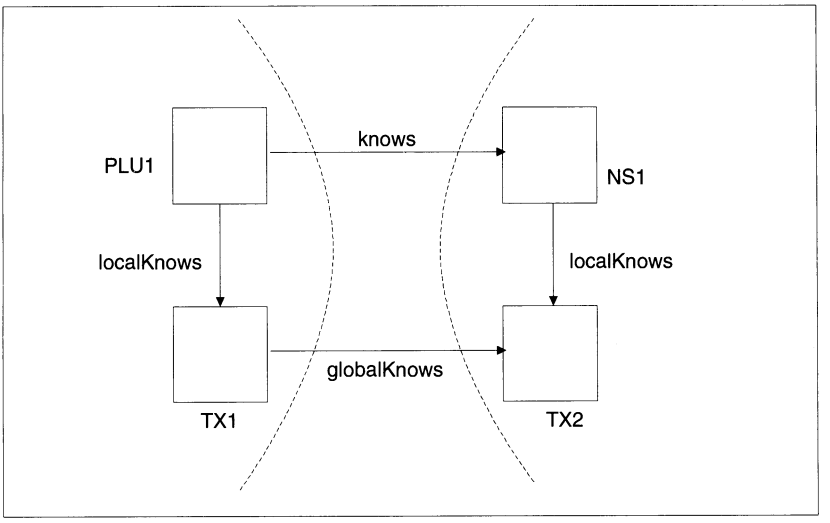


Figure 8.7. Separation of Levels

$$newLSTill(LSTill) =$$
$$true \rightarrow +knows(LSTill, NS1);$$
$$+requires(LSTill, PriceLookUp);\ +requires(LSTill, Loyalty)$$

The interesting question is, if we create $newLSTill(Till1)$ say, does it inherit the existing configuration of $Till1$. The formal model says it does. If that is not what we intended, then we need a tighter description.

Suppose we define

$$exists(Object) =$$
$$\textbf{exists}\ relationship.(relationship(Object, Something)\textbf{or}$$
$$relationship(Something, Object))$$

then we can strengthen the precondition on $newLSTill$ to be $\neg exists(LSTill)$.

If however, what we require is to genuinely model the fact that the old $Till1$ and the new $Till1$ actually share something other than a name, for example that they share the same hardware, then we need to separate the objects which represent the Till application from those that represent the Till hardware. This can be done, but we will not go into it here.

Of course, in all practical cases we must realise that systems are implemented at different levels and we will need to model components at different levels of abstraction. Figure 8.7 shows how this is done. In a high level model, the relationship *knows* will be stored explicitly. In a refinement of that model, the relationship *knows* will be derived from other relationships (stored or derived). Figure 8.7 shows how the $PLU1$ and the $NS1$, in separate environments (processes, name spaces, machines etc) come to know each other by a conjunction of relationships, set up presumably by more primitive actions than *acquire*.

$$knows(A, B) =$$
$$localKnows(A, TX1) \& localKnows(B, TX2) \&$$
$$globalKnows(TX1, TX2)$$

8.3.3 *Loosely-coupled components*

The architectures we are trying to describe to support reuse in the context of constant change, with its consequent need for dynamic reconfiguration, are leading us towards increasingly loosely-coupled components.

The architecture which we have used as our example, the client-server architecture, has some of this looseness of coupling. Dynamic binding is achieved through the use of registry services such as the Name Server which we have used.

As an example of something more loosely coupled consider the following architecture which is a development of the client-server. We don't have specialised Name Servers. Rather, every object is a Name Server. This is achieved by ensuring that every object supplies both *Register* and *Acquire*. Now, on creation, every object must know the name of some other object, but that doesn't have to be always the same object.

Given the initiative to seek out as many new objects as it can, a new object can increase substantially its chances of being able to survive and continue to function, notwithstanding attacks from elsewhere.

8.3.4 *It's all a game*

We can characterise the fight for survival of a system, or perhaps more accurately the components within the system as a 2-person game. Imagine that the two players are the System itself and the Environment. The System can make a move comprising a sequence of actions, thus moving to a desired state, whereupon it yields. The Environment can then make a move which we presume will break something. The Environment wins if the System gets into a position from which it can not recover to a position which it is required to achieve.

Restricting the Environment to a single break action is a modelling choice, but it does allow us now to specify an interesting property of a System state. The property is an integer which counts the number of moves the System is away from losing.

Consider Figure 8.8. This shows a common situation in the game of survival. Each node in the diagram represents a state of the system and the integer in the circle is the number of moves the System is away from disaster. The game starts at node A. If the System is astute enough always to move to C, whenever it is at A, then the System survives. If it ever moves to D, then there is the chance that the Environment will win by moving to B.

You can see how this metaphor reasonably captures the notation of survival for a dynamic system. We hope to show that it also reasonably captures the notion of incremental change and improvement as the System moves further away from zero. We expect that this will require a considerably more complex numbering

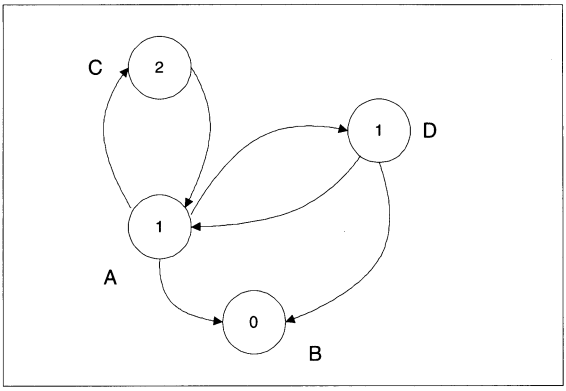


Figure 8.8. Positions in a game of Survival

scheme. We are investigating whether or not we can develop state models based on the game representation schemes devised by Conway [7].

8.4 Conclusions

We have concerned ourselves with the formalisation of dynamic systems, which we have characterised as systems of components that need to reconfigure themselves in order to respond to changes in the requirements upon them. We have shown how systems based on registry services are basically appropriate to this problem and have suggested some refinements to this architecture, specifically generalising the reasons why any component might store information about another. Another extension is to make every component (at a certain level) able to provide the capabilities of a registry.

We have introduced an architecture modelling language, ARC, which adopts the paradigm of modelling systems as objects and relations. This leads to an elementary behavioural description language which we have shown to be powerful enough to describe the systems which we wished to discuss. Validating the properties which we have proposed for solutions is possible using state-space search. We have a tool for doing this which we described briefly. More details, and the tool itself, are available on the web [17]

Further work on architecture modelling is ongoing. In particular we are building models of MQM [8], of Jini [35] and of the Ambients [5] paradigm as well as showing whether or not ARC can model most of the things that other architectural modelling languages can. Of course, theoretically it is possible to show that ARC can represent anything but we are more concerned with the practical use of the paradigm by software architects and software engineers in practice in real industrial scale tasks. We are confident that this objective will be achieved.

References

[1] Abramsky, S and G McCusker. Game Semantics, *see http://dcs.ed.ac.uk/abramsky* .

[2] R. J. Allen and D. Garlan, A Formal Basis For Architectural Connection, *ACM transactions on Software Engineering and Methodology*, July 97.

[3] R. J. Allen, Remi Douence, and David Garlan, Specifying Dynamism in Software Architectures *Workshop of Foundations of Component Based Systems*, Zurich, 1997, see http://www.cs.iastate.edu/ leavens/FoCBS/index.html

[4] M. Boman, J .A . Bubenko, P. Johannesson, and B Wangler, *Conceptual Modelling*, Prentice Hall, 1997.

[5] Luca Cardelli, Abstractions for Mobile Computation. *Microsoft Research Technical Report MSR-TR-98-34* available at research.microsoft.com.

[6] E. M. Clarke et al. Model Checking and Abstraction, *ACM Transactions on Programming Languages and Systems*, Sept. 94.

[7] J. Conway. *On Numbers and Games*, Academic Press, 1976.

[8] A. Dickman. *Designing Applications with MSMQ*, Addison Wesley, 1998.

[9] M. Fowler. *UML Distilled - Applying the standard Object Modelling language*, Addison Wesley, 1997.

[10] David Garlan et al. Architectural Mismatch, or, why it's hard to build systems out of existing parts. *ICSE*, 179–185, 1995.

[11] A. Gravell and P. Henderson. Executing formal specifications need not be harmful, *Software Engineering Journal*, vol. 11, num 2., IEE, 1996.

[12] David N. Gray et al. Modern Languages and Microsoft's Component Object Model. *Communications of the ACM*, Vol 41, No 5, 55–65, 1998.

[13] O. Grumberg and D. Long. Model Checking and Modular Verification, *ACM Transactions on Programming Languages and Systems*, 843–871, May 1994.

[14] M. Heimdahl and N. Leveson. Completeness and Consistency in hierarchical state-based requirements, *IEEE Transactions on Software Enginerring*, 22(6):363–377, 1996.

[15] P. Henderson and G. D. Pratten. POSD - A Notation for Presenting Complex Systems of Processes, *Proceedings of the First IEEE International Conference on Engineering of Complex Systems*, IEEE Computer Society Press, 1995.

[16] Peter Henderson. Laws for Dynamic Systems, *International Conference on Software Re-Use (ICSR 98)*, Victoria, Canada, June 1998, IEEE Computer Society Press.

[17] Peter Henderson. ARC: A language and a tool for system level architecture modelling, July 1999, see http://www.ecs.soton.ac.uk/ ph/arc.htm .

[18] Peter Henderson and Bob Walters. System Design Validation using Formal Models, *Proceedings 10th IEEE Conference on Rapid System Prototyping, RSP'99*, IEEE Computer Society Press, 1999.

[19] Peter Henderson and Bob Walters. Component Based systems as an aid to Design Validation, *Proceedings 14th IEEE Conference on Automated Software Engineering, ASE'99*, IEEE Computer Society Press, 1999.

[20] C. A. R. Hoare. How did Software get to be so reliable without proof? *Keynote address at the 18th International Conference on Software Engineering*. IEEE Computer Society Press, 1996. See also http://www.comlab.ox.ac.uk/oucl/users/tony.hoare/publications.html .

[21] G. J. Holtzmann. The Model Checker SPIN, *IEEE Transactions on Software Engineering*, Vol 23, No 5, 279–295, 1997.

[22] D. Jackson. Alloy: A lightweight Object Modelling Notation, available at http://sdg.lcs.mit.edu/ dnj/abstracts.html, July 1999.

[23] R. Kurki-Suoni. Component and Interface Refinement in Closed-System Specifications, *Proceedings of World Congress on Fomal Methods, Toulouse, September 1999*, LNCS 1709, 134–154.

[24] D. C. Luckham et al. Specification and Analysis of System Architecture using Rapide, *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[25] D. C. Luckham and J. Vera. An event-based architecture definition language, *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[26] D. C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events, http://pavg.stanford.edu .

[27] J. Magee and J. Kramer. Dynamic Structure in Software Architecture, *Proceeedings of the ACM Conference on Foundations of Software Engineering*, Software Engineering Notes, 21(6):3–14, IEEE Computer Society Press, 1996.

[28] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures, *Proceedings of 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, LNCS 989, 137–154, September 1995.

[29] J. Magee and J. Kramer. Concurrency: State Models and Java Programs, Wiley, 1999.

[30] Object Management Group. Common Object Request Broker: Architecture Specification, http://www.omg.com .

[31] M. Shaw et al. Abstractions for Software Architecture and Tools to Support Them, *IEEE Transactions on Software Engineering*, 21(4) 314–335, April 1995.

[32] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an emerging discipline*. Prentice Hall, 1996.

[33] K. Sullivan J. C. Knight. Experience Assessing an Architectural Approach to Large Scale Reuse, *Proceedings of ICSE-18*, 1996 IEEE Computer Society Press.

[34] K. Sullivan, J. Socha, and M. Marchukov. Using Formal Methods to Reason about Architectural Standards, *19th International Conference on Software Engineering*, Boston, IEEE Computer Press, 1997.

[35] Sun Microsystems. Jini Software Simplifies Network Computing, available at www.sun.com/jini .

[36] D. Wile. AML: an Architecture Meta-Language, *Proceedings ASE 1999*, IEEE Computer Society Press, pp 183-190.

9

"What is a method?" — an essay on some aspects of domain engineering

Dines Bjørner

Abstract

We discuss a concept of method in terms of its postulated principles, techniques and tools for the realm of software engineering. Software engineering is here seen as a confluence of *domain engineering*, *requirements engineering* and *software design*. Our scope is the concept of *domains* and *domain engineering*, and, our span is the concept of *domain facets*. We shall briefly contrast these with *domain attributes* such as for example put forward by Michael Jackson [1]. For the *domain facet* area of software development we then identify, exemplify and investigate, the latter rather briefly, a number of *domain facet* development principles and techniques. The main contributions of this essay are believed to be the identification of the *domain facet* concept, and the collection (including identification), classification, part investigation, and "fitting into a larger whole", of *domain facet* principles and techniques, as well as the thereby substantiated claim that these principles and techniques help characterise methods.

The essay has technical examples, but they are merely sketches. Had they been more substantial, the essay would not have been an essay. More substantial examples are given elsewhere[1]

9.1 Introduction

9.1.1 Domains, Requirements and Software Design

We assume the basic dogma: Before *software* can be *designed* it must be *requirements specified*. And before *requirements* can be expressed, an understanding of the world in which these *requirements* reside, the *domain*, must be formulated.

[1] See the author's lecture notes: `http://www.imm.dtu.dk/~db/setap/contents.ps`

A. McIver et al. (eds.), *Programming Methodology*

The *software design* describes how a computer (the hardware) is to proceed in order to achieve stated *requirements*. The *requirements* usually describe three things: (1) Which phenomena of the *domain* shall be supported by computing (the *domain requirements*); (2) which interface between the *machine* (*hardware + software*) and external phenomena — People, and other sensors and actuators — shall be provided (the *interface requirements*); and (3) what performance, dependability, maintenance, platform, and documentation measures are expected (the *machine requirements*).

Domain descriptions are indicative: Describe the "chosen world as it is", i.e. the *domain* — without any reference to *requirements*, let alone *software design*. *Requirements prescriptions* are putative: Prescribe what there is to be — properties, not designs, of the *machine*.

Domain descriptions must describe the chosen *domain* with its imperfections, not try to "paint a picture" of a "world as one would like it to be". In this essay we shall focus on such *domain descriptions*.

In this essay we shall not touch upon the methodological principles, techniques and tools that allow the software developer, based on formal descriptions of the domain to rigorously *project*, *instantiate*, *extend* and *initialise* a *domain description* "into" a *domain requirements definition*, and, from *domain* and *interface requirements definitions*, to similarly rigorously develop *software architecture designs*. We cover such principles, techniques and tools in other papers, e.g. [2, 3, 4, 5], and in our lecture notes.

We summarise:

- **Domains**
- **Requirements**
- **Software Design:**
  - **Software Architecture**
  - **Program Organisation**
  - **Etc. — Coding**

9.1.2 The Problem to be Addressed

In this essay we shall study some aspects of domain engineering only.

The overall problem that we are generally studying is that of methods for the development of large scale, typically infrastructure component software systems.

Excluded from our software development method concerns are therefore those related to the discovery, the invention of algorithms & data structures, for well-delineated problems such as sorting and searching, graph operations, fast Fourier transforms, parsing, *etc.* The borderline between infrastructure software systems and algorithms and & structures is indeed a fuzzy one — and one that we really do not wish to further investigate here. Suffice it to say that the infrastructure software systems we have in mind will indeed contain many examples of algorithms & data structures ! But as concerns the principles and

techniques of methods — we only claim that we investigate some that are deemed applicable to infrastructure software systems development.

9.1.3 Aspirations

The current author's ambition is to understand — in a comprehensive manner — suitable complements of principles and techniques for software development: *Where to start, how to proceed, and when to end.*

As forcefully pointed out by Jackson [1, 6, 7, 8], no one method suffices for all software development. Compilers seems best developed using one approach [9, 10, 11, 12], while real-time embedded and safety critical systems are perhaps best developed using an altogether different approach [13, 14, 15, 16].

Many software development principles and techniques transcend, however, their use in the development of individual, (frame) specific program packages and software systems. This essay is about such development issues.

9.1.4 Structure of Essay

In Section 9.2 we put forward a characterisation of what we consider to be a method, with its principles, teachniques, and tools, for (efficiently) analysing and synthesizing, i.e. constructing, (efficient, in this case) software.

The main section, Section 9.3, has two parts:

In Section 9.3.1 we look at problems of modelling the concerns of *stake-holders:* Their *perspective* on the domain[2].

In Section 9.3.2 we then look at a number of what we term *domain facets:* We currently list five such facets: *Intrinsics, support technology, management & organisation, rules & regulations* and *human behaviour.* Singling those out for individual, or otherwise clearly identified, modelling, we claim, satisfies an overall principle, that of separation of concerns, and seems to lead to more elegant descriptions.

Section 9.3 follows up on Section 9.2 in which we delineate what we, in general, see to be methods, methodology, principles, techniques, and tools.

9.1.5 Some Typographical Conventions

The text alternates between paragraphs which either contain plain text, or `characterises`, or `defines` a concept, which are then usually followed by paragraphs which `discuss` the concept, and paragraphs which state a `principle`, a `technique`, or a `tool`. We use the ▪ delimiter to show the end of the `specialised paragraphs`.

[2] As these stake-holders will also, later (but not to be covered in this essay) have a perspective on requirements

We make a distinction between characterisations and definitions: The former are (oftentimes necessarily) informal, the latter sometimes formalisable.

9.2 Method and Methodology

The notions of method and methodology are being "bandied about": "Some rules for engineering conduct", "some notation", or other, is claimed to be 'a method'. Some 'methods' are claimed to be 'formal'. In this section we take a first look at what might constitute a method. And we make a necessary distinction between method and methodology.

9.2.1 Method

Characterisation: *Method.* By a method we understand a set of principles for `selecting` amongst, and `applying`, a set of designated techniques and tools such which allow *analysis* and *construction* of artefacts. ▪

Discussion: The `selections` (of *analysis* and *synthesis* techniques and tools) and some of the *deployments* (of these techniques and tools) are to be carried out by people. The principles are usually of such a nature as to guide the developer, not to interfere with that person's possible ingenuity and creativeness, that person's ability to discover, to reflect and be skeptic. Hence we cannot ever expect to get anywhere near a formalisation of such principles. Therefore the term 'formal method' is unfortunate. Better would be formal techniques and formally based tools. Even better, to paraphrase Michel Sintzoff, would be to speak of logical or precise techniques and tools, as informal such are very much needed, but illogical or imprecise not. ▪

9.2.2 Methodology

Characterisation: *Methodology.* By methodology we understand the study of and knowledge about methods. ▪

Discussion: The two terms 'method' and 'methodology' are often used interchangeably — especially, it seems, in the US. ▪

9.2.3 Method Constituents

Discussion: The above 'method' characterisation identifies the following concepts: principle, *analysis,* construction, technique, tool, and 'artefact'. We need characterise these concepts. In the following we focus on domain descriptions as being the artefacts of interest. ▪

Principle

Characterisation: *Principle (I).* We quote from [17]: "An accepted or professed rule of action or conduct, ..., a fundamental doctrine, right rules of conduct, ...". ▪

Discussion: The concept of 'principle' is "fluid". Usually, by a method, some people understand an orderliness. Our 'definition' makes the orderliness part of the overall principles. Also: One usually expects analysis and construction to be efficient and to result in efficient artefacts. This too we relegate to be implied by some principles, techniques and tools. ▪

Characterisation: *Principle (II).* We make here the distinction between *d*evelopment principles (δ), and principles related to *c*oncepts (γ) of domain other than software development. We highlight the former by the texts "The Development Principle of δ", and the latter by the texts "The Principle of Modelling the γ Domain Concept". ▪

Analysis

Characterisation: *Analysis* is performed on descriptions. There seem to be three kinds of analysis. (i–ii) Informal validation or formal verification, including proof or model checking. This kind of analysis is performed, typically on narratives[3], respectively on formal texts. Such analyses lead to statements (i.e. meta-linguistic document texts) such as *"Such-and-such description text(s) denotes such-and--such properties"* ('is correct', or 'is not correct' [relating one part of the text to another], or 'denotes an NP-complete problem', etc.). (iii) Analysis performed on rough sketches, are not formalisable, but have the aim of forming concepts. ▪

Discussion: Descriptions describe some universe of discourse. We may claim that we are analysing that universe, but really, it is the model of that universe, in the form of some description, that we analyse. ▪

Construction [or: Synthesis]

Characterisation: *Construction* (or: *Synthesis*) means: The creation of a description, and thereby of a theory: A collection of properties that can be deduced

[3]We take it for granted that software development (in each (*domain, requirements* or *software design*) phase, and for each refinement or other development stage within phases, and for steps within stages) aims at constructing a number of `documents`: (a) `informative`, (b) `descriptive` — both `informal` and `formal` — and (c) `analytic`. Within informal descriptions we distinguish between those that are [non–deliverable] `rough sketches` — where `rough sketches`, often contain `rough formalisations`, cf. Example of Section 9.3.1 — and those that are `narratives` and `terminologies`. Informative documents inform about the development. Descriptions "inform" about (i.e. describe) a universe of discourse, as here: domain. And analyses "inform" about (i.e. analyse) descriptions; they are, in that sense, meta–linguistic.

from that description. The creation involves elicitation (acquisition), writing, analysis, rewriting, analysis again, rewriting, etc. ▪

Discussion: Writing informative or analytical documents may not be considered construction. They are necessary documents, but they do not describe manifest phenomena in the domain. ▪

Technique

Characterisation: *Technique.* [17]: "Method or technical skill, ...". ▪

Discussion: Already here we see a possible conflict: Our characterisation of 'method' involves the term 'technique' which by [17] is defined in terms of the term 'method'. We shall use the term 'technique' in the sense of the, or a specific 'procedure', 'routine' or 'approach' that characterises the technical skill. ▪

Tool

Characterisation: *Tool.* [17]: "An instrument for performing mechanical operations, a person used by another for his own ends, ..., to work or shape with a tool, ...". ▪

Discussion: We shall use the term tool in a wider sense: Any language is a tool, so is paper & pencil, blackboard & chalk, and so is any software package. Indeed, with language we shape concepts. ▪

9.2.4 The Method Principles

If, as we are now claiming, one can indeed identify a set of principles, techniques and tools that apply, conditionally, in a number of development situations, then these principles, techniques and tools ought probably also be deployed. Hence:

The Development Principle of *'Methodicity' — being Methodical* is now that of actually deploying relevant domain [and requirements] engineering [as well as software design] principles, techniques and tools during software development. ▪

Discussion: The hedge here is, obviously, the term 'relevant'. There is thus another meta-principle buried here. ▪

The Development Principle of *'Development Choice'* is a meta-principle, a 'conditional' that is part of every principle, technique and tool characterisation — is: Apply only a principle, a technique or a tool if its pre–conditions are met. ▪

The Meta–Technique of *'Methodicity'* expresses that, in respective phases of software development, one adheres to a list of (i) general abstraction & modelling, (ii) domain attribute, perspective and facet, (iii) domain requirements projection, instantiation, extension and initialisation, (iv) interface and (v) machine requirements, (vi) software architecture, (vii) program organisation — and many

other program design — principles, techniques and tools, ensuring that all due consideration is paid to these in the development. ▪

Discussion: In the current paper we shall only cover domain perspective and facet principles and techniques. In other papers and in lecture notes available over the net we cover many of the other principles and techniques mentioned above. ▪

The Meta–Technique of *'Development Choice'* expresses, relative to the previous 'methodicity' techniques, that for each of these one carefully writes down the assumptions upon which a choice of specific principle, technique or tool was deployed. ▪

Discussion: We have not, in this pape, for the sake of print space, enunciated these conditionals explicitly: They are, however, part of, and hence transpire indirectly from our coverage. ▪

9.2.5 Discussion

We have risked some debate as to whether the above delineations of what might constitute a method form a suitable basis.

Since 'methods' are to be deployed primarily by humans we prefer to characterise than to define. Definitions seem to have something more definite, more absolute about them. Characterisations seem more at ease.

Some may argue that the method principles, techniques and tool that we shall now endeavour to enumerate and investigate may unduly constrain the ingenuity of software developers: That having to follow these principles, to use those techniques and to deploy those tools may stifle their creativity. We believe the contrary: That the principles set the developer free, that having recognized techniques and tools allow the developer to focus on concepts, and put the mind to work on those: thinking, rather than "bureaucratic" labouring.

9.3 Domain Perspectives and Facets

We treat the subject of domain engineering in two parts. First we consider the plethora of stake-holders, that is: Individuals and institutions that are more-or-less interested in, or influenced by what goes on in the domain. Then we consider a concept of domain facets.

Thus we omit consideration of domain attributes ((i) *static* and *dynamic*, (ii) *tangible* and *intangible*, (iii) configuration spectra between contexts and states, (iv) time, space and space–time, (v) discreteness, continuity and chaos, (vi) hierarchies and compositions, and many others)[4] — *some of which*, (i–ii), *have been put forward by Michael Jackson* [1].

[4]This omission is due to page limitations. A proper study of 'methods', 'principles', 'techniques', and 'tools', would benefit, it is believed, from more comprehensive comparisons.

Domain attributes and domain facets are different: Attributes of different domain entities can be modelled more or less independently, i.e. "in parallel" ! In contrast, one usually tackles the description of a domain facet-by-facet. The domain attributes (tangibility, statics vs. dynamics, etc.) are not exclusively domain attributes: One may reasonably claim that during subsequent development phases (after domain engineering: Requirements engineering and software design) one may also reconsider (hence: New) deployments of attendant attribute principles and techniques.

Modelling stake-holder perspectives, domain attributes and domain facets otherwise takes place, during development, "concurrently": One alternates "to-and-from" iteratively.

There are additional description principles that we also do not cover: Property versus model oriented specification, representational and operational abstraction, denotational vs. computational models, etc. They "belong" in a class of modelling issues that we consider different from those of attributes and facets.

Our choice of the term 'facet' is just a choice. Whatever term was chosen, it had to be different from the term 'attribute'. Maybe for that other ("belong") class of modelling issues, just referred to above, we could then choose the term 'aspects' !

9.3.1 Stake–holders and Stake–holder Perspectives

Stake-holders

Characterisation: *Stake–holder.* By a stake–holder we mean a closely knit, tightly related group of either people or institutions, pressure groups — where the "fabric" that "relates" members of the group, "separates" these from other such stake-holder groups, and from non-stake-holder entities. ▪

Discussion: We shall not here try establish an ontology for the stake-holder concept. If one tried, that ontology would, on one hand, need to deal with issues of 'part of' and 'whole', as for system and component ontologies, and, on the other hand, since we are dealing mostly with human institutions, the ontology would probably have to incorporate a fuzzy membership notion. ▪

Examples: *Stake–holders* include enterprise staff: (i) owners, (ii) management (a) executive management (b) line management, and (c) "floor", i.e. operations management, and (iii) workers of all kinds, (iv) families of the above, (v) the customers of the enterprise, (vi) competitors, and the external, "2nd tier" stake–holders: (vii) resource providers (a) IT resource providers[5], (b) non–IT/non–finance [6], and (c) financial service providers, (viii) regulatory agencies

[5] Viz.: Computer hardware and other IT equipment, software houses, facilities management, etc.

[6] Viz.: Consumable goods, leasing agencies, etc.

who oversee enterprise operations[7], (ix) local and state authorities, (x) politicians, and the (xi) "public at large". They all have a perspective on the enterprise. . ▪

Discussion: It always makes good, commercial as well as technical, sense to incorporate the views of as many stake–holder groups as are relevant in the software development process. One need not refer to social, including so-called democratic, reasons for this inclusion. It is simply more fun to make sure that one has indeed understood as much of the domain (and, for that matter, as much of possible requirements) as is feasible, before embarking on subsequent, costly software development phases. ▪

The Principle of Modelling the *Stake–holder* Concept expresses that the developer and the client, when setting out on a domain description, clearly defines which stake-holders must be recognised and duly involved in the development. ▪

Technique of Modelling the *Stake–holder* Concept: Consider modelling each stake-holder group as a process, or a set of processes (i.e. behaviour[s]), or define suitable stake-holder specific context and/or state components and associated (observer and generator) functions. ▪

Stake-holder Perspectives

Characterisation: *Stake–holder Perspective.* By a stake–holder perspective we mean a partial description, a description which emphasises the designations, definitions and refutable assertions[8] that are particular to a given stake-holder group, or the interface between pairs, etc., of such. ▪

Discussion: Each perspective usually gives rise to a distinct view of the domain. These views share properties. A good structuring of the "totality" of perspectives can be helped by suitable, usually algebraic specification langauge constructs, such as possibly the class, scheme and object constructs of the RAISE [18] Specification Language RSL [19]. We shall not illustrate this point at present. ▪

The Principle of Modelling the *Stake–holder Perspective* Concept expresses that the developer and the client, when setting out on a domain description together, suitably as part of the contract, clearly defines which stake-holder perspectives must be recognised and duly included in the descriptions. ▪

Example: *Strategic, Tactical and Operations Resource Management.* We now present a rather lengthy example. It purports to illustrate the interface between a number of stake-holder perspectives. The stake-holders are here: An enterprises's top level, executive management (which plan, takes and follows up on strategic decisions), its line management (which plan, takes and follows up on tactical decisions), its operations management (which plan, takes and follows up on operational decisions), and the enterprise "workers" (who carry out decisions through

[7] Viz.: Environment bureaus, financial industry authorities (e.g.: The US Federal Reserve Board), food and drug administration (e.g.: The US FDA), health authorities (e.g.: The US HEW), etc., depending on the enterprise.

[8] Designations, definitions and refutable assertions are concepts defined in [1].

tasks). Strategic management here has to do with upgrading or downsizing, i.e. converting an enterprise's resources from one form to another — making sure that resources are available for tactical management. Tactical management here has to do with temporally scheduling and spatially allocating these resources, in preparation for operations management. Operations management here makes final scheduling and allocation, but now to tasks, in preparation for actual enterprise ("floor") operations.

Let R, Rn, L, T, E and A stand for resources, resource names, spatial locations, times, enterprises (with their estimates, service and/or production plans, orders on hand, etc.), respectively tasks (actions). SR, TR and OR stand for strategic, tactical and operational resource views, respectively.[9] SR expresses (temporal) schedules: Which sets of resources are either bound or free in which (pragmatically speaking: overall, i.e. "larger") time intervals. TR expresses temporal and spatial allocations of sets of resources, in certain (pragmatically speaking: mode finer "grained", i.e. "smaller") time intervals, and to certain locations. OR expresses that certain actions, A, are to be, or are being, applied to (parameter–named) resources in certain time intervals.

type R, Rn, L, T, E, A
  RS = R-**set**
  SR = (T×T) $\overrightarrow{m}$ RS, SRS = SR-**infset**
  TR = (T×T) $\overrightarrow{m}$ RS $\overrightarrow{m}$ L, TRS = TR-**set**
  OR = (T×T) $\overrightarrow{m}$ RS $\overrightarrow{m}$ A
  A = (Rn $\overrightarrow{m}$ RS) $\xrightarrow{\sim}$ (Rn $\overrightarrow{m}$ RS)
value
  obs_Rn: R → Rn
  srm: RS → E×E $\xrightarrow{\sim}$ E × (SRS × SR)
  trm: SR → E×E $\xrightarrow{\sim}$ E × (TRS × TR)
  orm: TR → E×E $\xrightarrow{\sim}$ E × OR
  p: RS × E → **Bool**
  ope: OR → TR → SR → (E×E×E×E) → E × RS

The partial, including loosely specified, and in cases the non-deterministic functions: srm, trm and orm stand for strategic, tactical, respectively operations resource management. p is a predicate which determines whether the enterprise can continue to operate (with its state and in its environment, e), or not. To keep our model "small", we have had to resort to a "trick": Putting all the facts knowable and needed in order for management to function adequately into E ! Besides

[9] In the formalisation, take for example that of OR, i.e.: OR = (T×T) $\overrightarrow{m}$ RS $\overrightarrow{m}$ A = defines OR to be the type of maps ($\overrightarrow{m}$) fro time periods (intervals (T×T)) into maps from sets of resources RS into actions (A) [to be performed on these resources during the stated time interval]. These actions are partial functions ($\xrightarrow{\sim}$) from argument (Rn) named sets of resources (RS) into similarly such named results.

the enterprise itself, E also models its environment: That part of the world which affects the enterprise.

There are, accordingly, the following management functions:

Strategic resource management, srm(rs)(e,e′′′′),let us call the result (e′,(srs,sr)) [see "definition" of the enterprise "function" below],proceeds on the basis of the enterprise (e) and its current resources (rs), and "ideally estimates" all possible strategic resource acquisitions (upgrading) and/or downsizings (divestmments) (srs), and selects one, desirable strategic resource schedule (sr). The "estimation" is heuristic. Too little is normally known to compute sr algorithmically. One can, however, based on careful analysis of srm's pre/post conditions, usually provide some form of computerised decision support for strategic management.

Tactical resource management, trm(sr)(e,e′′′′),let us call the result(e′′,(trs,tr)), proceeds on the basis of the enterprise (e) and one chosen strategic resource view (sr) and "ideally calculates" all possible tactical resource possibilities (trs), and selects one, desirable tactical resource schedule & allocation (tr). Again trm can not be fully algorithmitised. But some combinations of partial answer computations and decision support can be provided.

Operations resource management, orm(tr)(e,e′′′′),let us call the result(e′′′,or), proceeds on the basis of the enterprise (e) and one chosen tactical resource view (tr) and effectively decides on one operations resource view (or). Typically orm can be algorithmitised — applying standard operations research techniques.

We refer to [20] for details on the above and below model.

Actual enterprise operation, ope, enables, but does not guarantee, some "common" view of the enterprise: ope depends on the views of the enterprise, its context, state and environment, e, as "passed down" by management; and ope applies, according to prescriptions kept in the enterprise context and state, actions, a, to named (rn:Rn) sets of resources.

The above account is, obviously, rather "idealised". But we hope it is indicative of what is going on. To give a further abstraction of the "life cycle" of the enterprise, we "idealise" it, as now shown:

```
value
  enterprise: RS ~→ E ~→ Unit
  enterprise(rs)(e) ≡
    if p(rs)(e) then
      let (e′,(srs,sr)) = srm(rs)(e,e′′′′),
          (e′′,(trs,tr)) = trm(sr)(e,e′′′′),
          (e′′′,or) = orm(tr)(e,e′′′′),
          (e′′′′,rs′) = ope(or)(tr)(sr)(e,e′,e′′,e′′′) in
      let e′′′′′:E • p′(e′′′′,e′′′′′) in
      enterprise(rs′)(e′′′′′) end end
    else stop end

  p′: E × E → Bool
```

The enterprise re-invocation argument, rs′, a result of operations, is intended to reflect the use of strategially, tactically and operationally acquired, spatially and task allocated and scheduled resources, including partial consumption, "wear & tear", loss, replacements, etc.

The **let** e′′′′′:E • p′(e′′′′,e′′′′′) **in** ... shall model a changing environment.

Thus there were two forms of recursion at play here: The simple tail-recursion, and the recursive "build-up" of the enterprise state e′′′′. The latter is the interesting one. Solution, by iteration towards some acceptable, not necessarily minimal fix-point, "mimics" the way the three levels of management and the "floor" operations change that state and "pass it around, up-&-down" the management "hierarchy". The operate function "unifies" the views that different management levels have of the enterprise, and influences their decision making. Dependence on E also models potential interaction between enterprise management and, conceivably, all other stake-holders. ▪

Discussion: We remind the reader that — in the previous example — we are "only" modelling the domain ! That model is, obviously, sketchy. But we believe it portrays important facets of domain modelling and stake-holder perspectives. The stake–holders were, to repeat: Strategy ("executive") management (srm, p), tactical ("line") management (trm), operations ("floor") management (orm), and the workers (ope). The perpective being modelled focused on two aspects: Their individual jobs, as "modelled" by the "functions" (srm, p, trm, orm, ope), and their interactions, as "modelled" by the passing around of arguments (e, e′, e″, e‴, e′′′′) The **let** e′′′′′:E • p′(e′′′′,e′′′′′) **in** ... which "models" the changing environment is thus summarising the perspectives of "all other" stake–holders ! We are modelling a domain with all its imperfections: We are not specifying anything algorithmically; all functions are rather loosely, hence partially defined, in fact only their signature is given. This means that we model well-managed as well as badly, sloppily, or disastrously managed enterprises. We can, of course, define a great number of predicates on the enterprise state and its environment (e:E), and we can partially characterise intrinsics — facts that must always be true of an enterprise, no matter how.

If we "programme-specified" the enterprise then we would not be modelling the domain of enterprises, but a specifically "business process engineered" enterprise. Or we would be into requirements engineering — we claim. ▪

Technique of Modelling the *Stake–holder Perspective* Concept: Emphasize how the distinct stake-holders interact, which phenomena in the domian they generate, share, or consume. This 'technique' follows up on the 'Stake–holder' modelling technique. ▪

Discussion

The stake-holder example given above is "sketchy". It identifies, we believe, the most important entities and operations that are relevant to a small number of interacting stake-holders. We believe that "rough sketches" like the above are necessary in the iterative development of domains.

9.3.2 *Domain Facets*

We shall outline the following facets:

Domain intrinsics: That which is common to all facets.

Domain support technologies: That in terms of which several other facets (intrinsics, management & organisation, and rules & regulations) are implemented.

Domain management & organisation: That which primarily determines and constrains communication between enterprise stake-holders.

Domain rules & regulations: That which guides the work of enterprise stake-holders, their interaction, and the interaction with non-enterprise stake-holders.

Domain human behaviour: The way in which domain stake-holders despatch their actions and interactions with respect to enterprise: dutifully, forgetfully, sloppily — yes, even criminally.

We shall briefly characterise each of these facets. We venture to express "specification patterns" that "most closely capture" essences of the facet.

Separating the treatment of each of these (and possibly other) facets reflect a principle:

The Development Principle of *Separation of Concerns* expresses that when possible one should separate distinguishable concerns and treat them separately.▪

Discussion: We believe that the facets we shall present can be treated separately in most developments — but not necessarily always. Separation or not is a matter also of development as well as of presentation style.

The separation, in more generality, of computing systems development into the triptych of domain engineering, requirements engineering and machine (hardware + software) design, is also a result of separation of concerns — as are the separations of domain requirements, interface requirements and machine requirements (within requirements engineering), as well as the separations of software architecture and program organisation design [3]. ▪

Intrinsics

The Concept

Characterisation: *Intrinsics:* That which is common to all facets. ▪

An Example

Example: *Rail nets and switches.* We first give a summary view of a domain model for railway nets, first informally, then formally, leaving out axioms: A railway net consists of two or more stations and one or more lines. Nets, lines and stations consists of rail units. A rail unit is either a linear unit, or a switch unit, or a crossover unit, etc. Units have connectors. A linear unit has two connectors, a switch unit has three, a crossover unit has four, etc. A line is a linear sequence of connected linear units. A station usually has all kinds of units. A line connects exactly two distinct stations. A station contains one or more tracks (say, pragmatically, for passenger platforms or for cargo sidings). A path is a pair of connectors

of a unit, and pragmatically defines a way for a train to traverse that unit. A unit is at any one time in a state (σ), which we may consider a set of paths. Over a lifetime a unit may attain one or another state in that unit's state space (ω).

type
  N, L, S, U, C
value
  obs_Ls: N → L-**set**, obs_Ss: N → S-**set**,
  obs_Us: (N|L|S) → U-**set**, obs_Cs: U → C-**set**
  obs_Trs: S → Tr-**set**
type
  P′ = U × (C×C), Σ = P-**set**, Ω = Σ-**set**
  P = {| p:P′ • **let** (u,(c,c′))=p **in** (c,c′)∈ obs_Σ(u) **end** |}
value
  obs_Σ: U → Σ, obs_Ω: U → Ω

From the perspective of a train passenger or a cargo customer it is not part of the intrinsics that nets have units and units have connectors. Therefore also paths, states and state-spaces are not part of the intrinsics of a net as seen from such stake–holders.

From the perspective of the train driver and of those who provide the setting of switches and signalling in general, units, paths, and states are indeed part of the intrinsics: The intrinsics of a rail switch is that it can take on a number of states. A simple switch ($\overset{c_|}{\underset{c}{Y}}\overset{c_/}{}$) has three connectors: $\{c, c_|, c_/\}$. c is the connector of the common rail from which one can either "go straight" $c_|$, or "fork" $c_/$.

$$\begin{aligned}\omega_{g_s} : \{\ & \{\}, \\ & \{(c,c_|)\},\{(c,c_|),(c_|,c)\},\{(c_|,c)\}, \\ & \{(c,c_/)\},\{(c,c_/),(c_/,c)\},\{(c_/,c)\}, \\ & \{(c,c_/),(c_|,c)\},\{(c,c_/),(c_/,c),(c_|,c)\},\{(c_/,c),(c_|,c)\}\ \}\end{aligned}$$

ω_{g_s} ideally models a general switch. Any particular switch ω_{p_s} may have $\omega_{p_s} \subset \omega_{g_s}$. Nothing is said about how a state is determined: Who sets and resets it, whether determined solely by the physical position of the switch gear, or also by visible or virtual (i.e. invisible, intangible) signals up or down the rail away from the switch.

■

Methodological Consequences

The Principle of Modelling the *Intrinsics* Domain Facet expresses that in any modelling one first form and describe the intrinsic concepts. ■

Technique of Modelling the *Intrinsics* Domain Facet: The intrinsics model of a domain is a partial specification. As such it involves the use of well–nigh all description principles. Typically we resort to property oriented models, i.e. sorts and axioms. ■

Discussion

Thus the intrinsics become part of every one of the next facets. From an algebraic semantics point of view these latter are extension of the above.

Support Technologies

The Concept

Characterisation: *Support Technology* — that in terms of which several other facets (intrinsics, management & organisation, and rules & regulations) are implemented. ▪

An Example

Example: *Railway switches.* An example of different technology *stimuli:* A railway switch, "in ye olde days" of the "childhood" of railways, was *manually "thrown";* later it could be mechanically controlled from a distance by *wires and momentum "amplification";* again later it could be electro-mechanically controlled from a further distance by *electric signals that then activated mechanical controls;* and today switches are usually *controlled in groups that are electronically interlocked.*

An aspect of supporting technology includes the recording of state-behaviour in response to external stimuli. Figure 9.1 indicates a way of formalising this aspect of a supporting technology.

Figure 9.1. Probabilistic State Switching

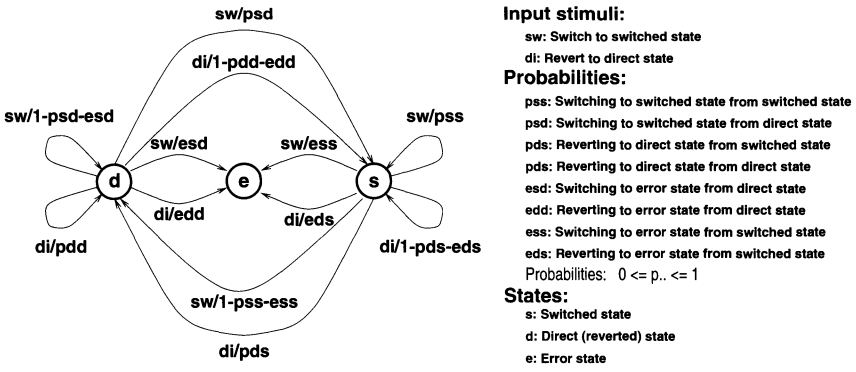


Figure 9.1 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and re–settings (to direct (d) state). A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd.. ▪

Another Example

Another example shows another aspect of support technology.

Example: *Air traffic radar.* Air traffic (iAT), intrinsically, is a total function over some time interval, from time (T) to monotonically positioned (P) aircraft (A).

A conventional air traffic radar "samples", at regular intervals, the intrinsic air traffic. Hence a radar is a partial function[10] from intrinsic to sampled air traffics (sAT).

type
  iAT = T → (A $\overrightarrow{m}$ P), sAT = T $\overrightarrow{m}$ (A $\overrightarrow{m}$ P)
value
  [radar] r: iAT $\xrightarrow{\sim}$ sAT, [close] c: P × P → **Bool**
axiom
  ∀ iat:iAT • **let** sat = r(iat) **in** ∀ t:T • t ∈ **dom** sat •
    t ∈ **dom** iat ∧ ∀ a:A • a ∈ **dom** iat(t) ⇒
      a ∈ **dom** sat(t) ∧ c((iat(t))(a),(sat(t))(a)) **end**

The axioms express a property that one expects to hold for a radar: That the radar-displayed aircraft positions are close to those of the aircraft in the actual world.

■

Methodological Consequences

Technique of Modelling the *Support Technology* Domain Facet: The support technologies model of a domain is a partial specification — hence all the usual abstraction and modelling principles, techniques and tools apply. More specifically: Support technologies (st:ST) "implements" intrinsic contexts and states: $\gamma_i : \Gamma_i, \sigma_i : \Sigma_i$ in terms of "actual" contexts and states: $\gamma_a : \Gamma_a, \sigma_a : \Sigma_a$

type
  Syntax,
  Γ_i, Σ_i, VAL_i, Γ_a, Σ_a, VAL_a,
  ST = Γ_i × Σ_i $\xrightarrow{\sim}$ Γ_a × Σ_a
value
  sts:ST-**set**
axiom
  ∀ st:ST • st ∈ sts ⇒ ...

Support technology is not a refinement, but an extension. Support technology typically introduces considerations of technology accuracy, failure, etc. Axioms

[10] This example is due to my former MSc Thesis student Kristian M. Kalsing.

characterise members of the set of support technologies sts. An example axiom was given in the air traffic radar example . ▪

The Principle of Modelling the *Support Technology* Domain Facet is a principle that is relative to all other domain facets. It expresses that one must first describe essential intrinsics. Then it expresses that support technology is any means of implementing concrete instantiations of some intrinsics, of some management & organisation, and/or of some rules & regulations. Generally the principle states that one must always be on the look-out for and inspire new support technologies. The most abstract form of the principle is: *"What is a support technology one day becomes part of the domain intrinsics a future day".* ▪

Discussion

[14, 13] exemplify the use of the Duration Calculus [21, 22, 23, 24, 25] in describing supporting technologies that help achieve safe operation of a road level rail crossing, and of a gas burner.

The support technology facet descriptions "re–appear" in the requirements definitions: Projected, instantiated, extended and initialised [3]. In the domain description we "only" record our understanding of all aspects of support technology "failures". In the requirements definition we then follow up and make decisions as to which kinds of "breakdowns" the computing system, the machine, is to handle, and what is to be achieved by such "handlings".

Management and Organisation

The Concept

Characterisation: *Management and Organisation:* That which primarily determines and constrains communication between enterprise stake-holders. ▪

Conceptual Examples — I

Discussion: People staff enterprises, the components of infrastructures with which we are concerned, for which we develop software. The larger these enterprises, these infrastructure components, are, the more need there is for management & organisation. The rôle of management is roughly, for our purposes, twofold: Firstly, to perform strategic, tactical and operational work (cf. example of Section 9.3.1), to make strategic, tactical and operational policies — including rules & regulations, cf. Section 9.3.2 — and to see to it that they are followed. The rôle of management is, secondly, to react to adverse conditions, unforeseen situations, and decide upon their handling, i.e. conflict resolution. Policy setting should help non-management staff operate in normal situations — for which no management interference is thus needed. And management "back--stops" problems: Takes these problems off the shoulders of non-management staff.

To help management and staff know who's in charge with respect to policy setting and problem handling, a clear conception of the overall organisation is

needed: Organisation defines lines of communication within management and staff and between these. Whenever management and staff has to turn to others for assistance they usually, in a reasonably well–functioning enterprise, follow the command line: The paths of organigrams — the usually hierarchical box and arrow/line diagrams. ▪

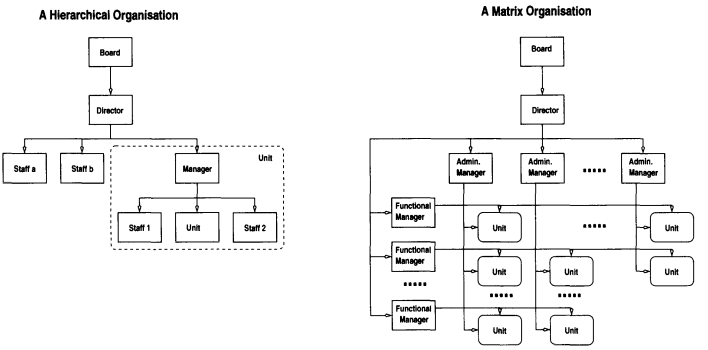
Methodological Consequences — I

Techniques of Modelling the *Management & Organisational* Domain Attributes Concepts: The management & organisation model of a domain is a partial specification — hence all the usual abstraction and modelling principles, techniques and tools apply. More specifically: Management is a set of predicates, observer and generator functions which either parameterise others, the operations functions, (that is, determine their behaviour), or yield results that become arguments to these other functions. We have indicated, in the example of Section 9.3.1, some of the techniques. Organisation is a set of constraints on communication behaviours. "Hierarchical", rather than "linear", and "matrix" structured organisations can also be modelled as sets (of recursively invoked sets) of equations. ▪

Conceptual Example — II

Examples: *Management & Organisation* To relate "classical" organigrams to formal descriptions we first show such an organigram, see Figure 9.2, and then we show schematic processes which — for a rather simple case (i.e. scenario) — model managers and managed !

Figure 9.2. Organisational Structures



type Msg, Ψ, Σ, Sx
channel { ms[i]:Msg | i:Sx }
value
sys: **Unit** → **Unit**

```
  mgr: Ψ → in,out { ms[i] | i:Sx } Unit
  stf: i:Sx → Σ → in,out ms[i] Unit
  sys() ≡ || { stf(i)(iσ) | i:Sx } || mgr(ψ)

value
  mgr(ψ) ≡
    let ψ′ = ... ;
      (|| {ms[i]!msg;f_m(msg)(ψ)|i:Sx } )
      □
      (□ {let msg′ = ms[i]? in g_m(msg′)(ψ) end|i:Sx}) in
    mgr(ψ′) end

  stf(i)(σ) ≡
    let σ′ = ... ;
      ((let msg = ms[i] ? in f_s(msg)(σ) end)
       □
       (ms[i] ! msg′ ; g_s(msg′)(σ))) in
    stf(i)(σ′) end

  f_m, g_m: Msg → Ψ → Ψ,
  f_s, g_s: Msg → Σ → Σ
```

Both manager and staff processes recurse (i.e. iterates) over possibly changing states. Management process non-deterministically, external choice, "alternates" between "broadcast"–issuing orders to staff and receiving individual messages from staff. Staff processes likewise non-deterministically, external choice, "alternates" between receiving orders from management and issuing individual messages to management. The example also illustrates modelling stake-holder behaviours as interacting (here CSP–like, [26, 27, 28]) processes. ▪

Methodological Consequences — II

Discussion: The *strategic, tactical and operations resource management* example of Section 9.3.1 (pages 183–186) illustrated another management & organisation description pattern. It is based on a set of, in this case, recursive equations. Any way of solving these equations, finding a suitable fixpoint, or an approximation thereof, including just choosing and imposing an arbitrary "solution", reflects some management communication. The syntactic ordering of the equations — in this case: a "linear" passing of enterprise "results" from "upper" equations onto "lower" equations — reflects some organisation. ▪

The Principle of Modelling the *Management & Organisation* Domain Facets expresses that relations between resources, and decisions to acquire and dispose resources, to de–, re– and schedule and de–, re– and allocate resources, and to de–, re– and activate resources, are the prerogatives of well-functioning management, reflect a functioning oranisation, and imply invocation of proce-

dures that are modelled as actions that "set up" and "take-down" contexts and change states. As such these principles tell us which sub-problems of development to tackle. ▪

Techniques of Modelling the *Management & Organisation* Domain Facet: We have already, under techniques for modelling 'Stake–holder' and 'Stake–holder Perspectives', mentioned some of the techniques. In this section we have used these techniques. Two "extremes" were shown: In Section 9.3.1 we modelled individual management groups by their respective functions (strm, trm, orm), and their interaction (i.e. organisation) by "solutions" to a set of recursive equations ! In this section we modelled management & organisation, especially the latter, by communicating sequential behaviours. ▪

Discussion

The domain models of management and organisation, of this section, as well as of the earlier section 9.3.1, eventually find their way into requirements, and, hence, the software design — for the cases that the requirements are about computing support of management and its organisation.

Support to solution of the recursive equations of the example of Section 9.3.1 may be offered in the form of constraint based logic solvers which may partially handle logic characterisations of the strategic and tactical management functions, and in the form of computerised support of message passing between the various management groups of the example of Section 9.3.1, as well as of the generic example of the present section.

Rules & Regulations

The Concept

Characterisation: *Rule.* That which guides the work of enterprise stake-holders as well as their interaction and the interaction with non-enterprise stake-holders. ▪

Characterisation: *Regulation.* That which stipulate what is to happen if a rule can be detected not to have been followed when such was deemed necessary. ▪

Rules & regulations are set by enterprises, enterprise associations, [government] regulatory agencies, and by law.

Three Examples

Examples: *Rail and Banking.* (i) Rail: Rule: *In China arrival and departure of trains at, respectively from railway stations are subject to the following rule: In any three minute interval at most one train may either arrive or depart.* Regulation: *Disciplinary procedures.* (ii) Rail: Rule: *In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving — obviously in the same direction — along the line, then there must be at least one free sector (i.e. without a train) between any two such trains.* Regulation: *Disciplinary procedures.* (iii) Banking: Rule: *In the United States of America personal checks issued in any one state of*

the Union must be cleared by the sending and receiving banks, if within the same state, then within 24 hours, and else within 48 or 72 hours, depending on certain further stipulated relations between the states. Regulation: *Fines and triple damages to affected clients.* ▪

Methodological Consequences

Technique of Modelling the *Rules & Regulations* Domain Facets: There are usually three kinds of syntax involved with respect to (i.e. when expressing) rules & regulations (resp. when invoking actions that are subject to rules & regulations: The syntaxes (Syntax_rul, Syntax_reg) describing rules, respectively regulations; and the syntax (Syntax_cmd) of [always current] domain external action stimuli. A rule, denotationally, is a predicate over domain stimuli, and current and next domain configurations ($\Gamma \times \Sigma$). A regulation, denotationally, is a state changing function over domain stimuli, and current and next domain configurations ($\Gamma \times \Sigma$). We omit treatment of [current] stimuli:

type Syntax_cmd, Syntax_rul, Syntax_reg, Γ, Σ
  Rules_and_Regulations = Syntax_rul × Syntax_reg
  RUL = $(\Gamma\times\Sigma) \to (\Gamma\times\Sigma) \to$ **Bool**,
  REG = $(\Gamma\times\Sigma) \to (\Gamma\times\Sigma)$
value
  interpret: Syntax_rul $\to \Gamma \to \Sigma \to$ RUL-**set**,
  interpret: Syntax_reg $\to \Gamma \to \Sigma \to$ REG

  valid: RUL-**set** $\to (\Gamma\times\Sigma) \times (\Gamma\times\Sigma) \to$ **Bool**
  valid(ruls)$((\gamma,\sigma),(\gamma',\sigma')) \equiv$
    $\forall$ rul:RUL • rul $\in$ ruls $\Rightarrow$ rul$(\gamma,\sigma)(\gamma',\sigma')$

  valid: REG $\to (\Gamma\times\Sigma) \to (\Gamma\times\Sigma) \to$ **Bool**
  valid(reg)(γ,σ) **as** (γ',σ')
    post reg$(\gamma,\sigma) = (\gamma',\sigma')$

axiom
  $\forall$ (ruls,reg):Rules_and_Regulations • $\exists$ $(\gamma,\sigma),(\gamma',\sigma'):\Gamma\times\Sigma$ •
    $\exists$ $\gamma'':\Gamma$, $\sigma'':\Sigma$
      • $\sim$valid(ruls)$((\gamma,\sigma),(\gamma'',\sigma''))$
        $\Rightarrow$ valid(reg)$(\gamma'',\sigma'') = (\gamma',\sigma')$

Rules & regulations are therefore modelled by abstract or concrete syntaxes of syntactic rules etc., by abstract types of denotations, and by semantics definitions, usually in the form of axioms or denotation–ascribing functions. ▪

The Principle of Modelling the *Rules & Regulations* Domain Facet expresses that domains are governed by rules & regulations: By laws of nature or edicts by humans. Laws of nature can be part of intrinsics, or can be modelled as rules & regulations constraining the intrinsics. Edicts by humans usually

change, but are usually considered part of an irregularly changing context, not a recurrently changing state. Modelling techniques follow these priciples. ▪

Rules & Regulation Scripts

We discuss an issue that arises with the above and which points to possible precautionary and/or remedial actions — as they would first be expressed in some reguirements:

Discussion: Domain rules & regulations are usually formulated in "almost legalese", i.e. in rather precise, albeit perhaps "stilted" subsets of the professional language of the domain in question. In cases such rules & regulation languages can be formalised, and we then call them script languages. A particular set of rules & regulations is thus a script. Such script languages can be mechanised: Making it "easy" for appropriate (rules & regulation issuing) stake-holders to script such scripts — and to have them inserted into their computing system: As predicates that detect rule violations, respectively suggest alternative actions (rather than causing a potentially violating action) or remedy an actual rule violation. ▪ The rules & regulations, that may be stipulated for a domain, can thus find their way into requirements that specify computerised support for their enforcement.

Human Behaviour

The Concept

Discussion: Some people try their best to perform actions according to expectations set by their colleagues, customers, etc. And they usually succeed in doing so. They are therefore judged reliable and trustworthy, good, punctual professionals (*b_p*) of their domain. Some people set lower standards for their professional conduct: Are sometimes or often sloppy (*b_s*), make mistakes, unknowingly or even knowingly. And yet other people are outright delinquent (*b_d*) in the despatch of their work: Couldn't care less about living up to expectations of their colleagues and customers. Finally some people are explicitly criminal (*b_c*) in the conduct of what they do: Deliberately "do the opposite" of what is expected, circumvent rules & regulations, etc. And we must abstract and model, in any given situation where a human interferes in the "workings" of a domain action, any one of the above possible behaviours. ▪

Characterisation: *Human Behaviour.* The way in which domain stake-holders despatch their actions and interactions with respect to an enterprise: professionally, sloppily, delinquently, yes even criminally. ▪

Methodological Consequences

Techniques of Modelling the *Human Behaviour (I–II)* Domain Facet: We often model the "arbitrariness" of human behaviour by internal non-determinism:

... b_p ⌈⌉ b_s ⌈⌉ b_d ⌈⌉ b_c ...

The exact, possibly deterministic, meaning of each of the b's can be separately described.

In addition we can model human behaviour by the arbitrary selection of elements from sets and of subsets of sets:

```
type
  X
value
  hb_i: X-set ... → ... , hb_i(xs,...) ≡ let x:X • x ∈ xs in ... end
  hb_j: X-set ... → ... , hb_j(xs,...) ≡ let xs′:X-set • xs′ ⊆ xs in ... end
```

The above shows just fragments of formal descriptions of those parts which reflect human behaviour. Similar, loose, descriptions are used when describing faulty supporting technologies, or the "uncertainties" of the intrinsic world. ▪

Technique of Modelling the *Human Behaviour (III)* Domain Facet: Commensurate with the above, humans interpret rules & regulations differently, and not always "consistently" in the sense of repeatedly applying the same interpretations. Our final specification pattern is therefore:

```
type
  RULS = RUL-set
  Action = Γ ~→ Σ ~→ (Γ × Σ)-infset
value
  interpret: Syntax_rul → Γ → Σ → RULS-infset

  human_behaviour: Action → Syntax_rr → Γ ~→ Σ ~→ Γ × Σ
  human_behaviour(α)(srr)(γ)(σ) as (γ′,σ′)
    post
      let γσs = α(γ)(σ) in
      ∃ (γ′,σ′):(Γ×Σ) • (γ′,σ′) ∈ γσs ∧
        let rules:RULS • rules ∈ interpret(srr)(γ)(σ) in
          ∀ rule:RUL • rule ∈ rules ⇒ rule(γ,σ)(γ′,σ′) end end
```

The above is, necessarily, sketchy: There is a possibly infinite variety of ways of interpreting some rule[s]. A human, in carrying out an action, interprets applicable rules and chooses a set which that person believes suits some (professional, sloppy, delinquent or criminal) intent. "Suits" means that it satisfies the intent, i.e. yields **true** on the pre/post state pair, when the action is performed — whether as intended by the ones who issued the rules & regulations or not. ▪

Discussion: Please observe the difference between the version of interpret as indicated in Section 9.3.2 and the present version: The former reflected the semantics as intended by the stake–holder who issued the rules & regulations. The latter reflects the professional, or the sloppy, or the delinquent, or the criminal semantics as intended by the similarly "qualified" staff which carries out the rule abiding

or rule violating actions. Please also observe that we do not here exemplify any regulations. ▪

The Principle of Modelling the *Human Behaviour* Domain Facet expresses what has now been mentioned several times, namely that some people are perfect: Follow rules & regulations as per intentions; other people are sloppy: Fail to follow the prescriptions; and yet other people are derelict or even criminal in the pursuit of their job: Deliberately flaunt rules & regulations. And the principle concludes that one must be prepared for the "worst". That is: Model it all. ▪

Discussion

The results of informal as well as formal domain descriptions of human shortcomings find their way into those requirements which define computerised support for taking precautionary actions should human errors be detected.

Discussion

We have covered a number of domain facets: *Intrinsics* ('the very basics'), *support technologies* (implementations of some parts of other facets), *management & organisation, rules & regulations,* and *human behaviour.* One can possibly think of other facets. With each domain facet the "full force" of all abstraction and modelling principles and techniques apply, and a careful "sequencing" ("fitting-in") of the treatment of 'that' facet with respect to other facets must be considered.

For each of the facets we have shown principles of and techniques for their modelling, and we have indicated that these facet models may eventually find their way into requirements models, and hence determine software designs.

9.3.3 Discussion

And we have covered, on a larger scale, the domain modelling of (domain) stakeholder perspectives and domain facets. The two concepts are not orthogonal. Their individual and combined treatment again demands judicious choice.

It has, throughout, been indicated how the domain model predicates the requirements, and hence the design.

One will never be able, it is conjectured, to achieve a complete domain model. But one can do far better than is practice today — where no such models are even attempted. Most claims of domain models are really biased towards contemplated software designs, embodying requirements, and are just covering at most the domain being projected, etc.

In the validation interaction between the software developers — who are major "players" in the development of both domain descriptions and requirements definitions — and the domain stake–holders, in that validation process, we claim, many errors — that before could, and hence would, creep unconsciously into the software development — can now be avoided. When indeed errors, i.e. "holes" in

the domain description, are still discovered, later, perhaps after final software delivery, then it is now easier, we claim, to pinpoint where these errors first occurred, and hence who were the perpetrators: The software, cum domain or requirements or design, developers, or the stake-holders, or both parties. On one hand it is now easier to resolve legal issues, and, as well, to repair malfunctioning software. The latter because, in its development, from domains via requirements to designs, we adhere to an unstated principle: That of homomorphic development: *If two or more algebraically independent ("orthogonal") concepts are expressed in the domain and are to be "found", somehow, also in the software, then their implementation must be likewise distinguishable.*

9.4 Conclusion

We have tried, more precisely, than what is normally experienced, to formulate a concept of method, in particular as it applies to a narrow part of domain engineering.

We have emphasised method principles and techniques, and we have proposed a number of domain perspective and facet modelling principles and related techniques.

We have only briefly referred to tools, and then only to linguistic tools such as natural language, the professional (i.e. subset natural) languages of specific universes of discourse, here almost exclusively domains, and the formal languages that "carry" formal techniques such as RSL, Finite State Machines and the Duration Calculi.

9.4.1 Discussion

Now: Have we achieved what we wished ?

To some extent, "Yes !" To some other extent, "No !"

As concerns the 'Yes', the essay speaks for itself: Presents our "Yes !".

As concerns the 'No', we discuss now some shortcomings, such as we presently see them.

Not all principles need or seem to need associated tecniques: 'Separation of Concern' appears to be a meta-principle that is then followed up by a choice between various techniques — but we cannot really say that these latter techniques are that intimately associated with the 'Separation of Concern' principle ?

Those principles, for which we have listed associated techniques, these techniques have be rather simple-mindedly expressed. We should like to see sharper characterisations — of a nature that sets them more apart, that distinguishes them more uniquely.

For some techniques we have achieved a formal characterisation, viz.: 'Support Technology', 'Rules & Regulations' and, partly, 'Human Behaviour'. We should

like to see these further elaborated; and we should like to see remaining facets characterised more formally.

Then the essay, as it stands, isolated from treatments of many other software development principles and techniques, risks being too narrow in its view of methods, their principles and techniques. We refer here to the obvious lack of the mentioning of principles and techniques for such general abstraction & modelling issues as *property vs. model-oriented descriptions, representation & operation abstraction, denotation, computation and process abstractions, time, space and time–space* concerns, *'hierarc[h]ality' vs. compositionality, configuration, context and state* modelling, etc.; to such domain attribute issues as *statics and dynamics* [1], *tangibility* [1], *dimensionality* [1], *discreteness, continuity and chaos,* etc.; to such domain requirements issues as *projection, instantiation, extension* and *initialisation,* etcetera, etcetera !

Since we can identify very many principles and techniques, some specific to distinct phases of development (to domains, or to requirements, or to software design), some generally applicable — since this is possible — it gives, we believe, strength to the argument that we must collect all these principles and techniques, we must investigate them individually and in relation to others, structure their presentation, and come up with such structured lists of principles and techniques as were referred to in the 'Methodicity' principles and its related technique, etc.

9.4.2 Future Work

The above discussion has pointed out some weaknesses, and has indicated additional work to be done: In meta–formalising some techniques, in collating "all so far identifiable" principles and techniques across at least the spectrum from and including domain engineering via requirements engineering [2] to inital parts of software design, notably software architecture and program organisation [3].

9.4.3 Acknowledgements

Acknowledgements are gratefully extended to Michael A. Jackson for his inspiring publications [29, 30, 31, 6, 32, 1, 7, 33, 34, 8], to the WG 2.3 membership for discussions of topics presented; to Hidetaka Kondoh of Hitachi Software Development Laboratories, Tokyo, for his thoughtful views on software development; and to my many, patient students, who have "suffered" lectures along the lines of this essay, and who have tested out their import in innumerable term and MSc projects.

It was at UNU/IIST[11] that we[12] systematically studied and applied, amongst may other programming and software engineering methodological issues, also the domain facets expounded in this chapter. My warmest acknowledgements goes to my colleagues during those years at UNU/IIST, 1991–1997, and beyond.

References

[1] Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*, Addison-Wesley, 1995.

[2] Dines Bjørner. Domains as Prerequisites for Requirements and Software *&c.* In M. Broy and B. Rumpe, editors, *RTSE'97: Requirements Targeted Software and Systems Engineering*, volume 1526 of *Lecture Notes in Computer Science*, pages 1–41. Springer-Verlag, Berlin Heidelberg, 1998.

[3] Dines Bjørner. Where do Software Architectures come from ? Systematic Development from Domains and Requirements. A Re–assessment of Software Engneering ? *South African Journal of Computer Science*, 22: 3–13, 1999.

[4] Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited plenum lecture.

[5] Dines Bjørner. Pinnacles of Software Engineering: 25 Years of Formal Methods. *Annals of Software Engineering*, 10:11–66, 2000. Eds. Dilip Patel and Wang Yi.

[6] Michael A. Jackson. Problems, Methods and Specialisation. *Software Engineering Journal*, pages 249–255, November 1994.

[7] Michael A. Jackson. Problems and requirements (software development). In *Second IEEE International Symposium on Requirements Engineering (Cat. No.95TH8040)*, pages 2–8. IEEE Comput. Soc. Press, 1995.

[8] Michael A. Jackson. The meaning of requirements. *Annals of Software Engineering*, 3:5–21, 1997.

[9] Dines Bjørner and O. Oest, editors. *Towards a Formal Description of Ada*, *LNCS*, vol. 98. Springer-Verlag, 1980.

[10] Dines Bjørner and M. Nielsen. Meta Programs and Project Graphs. In *ETW: Esprit Technical Week*, pages 479–491. Elsevier, May 1985.

[11] Dines Bjørner. Project Graphs and Meta-Programs: Towards a Theory of Software Development. In N. Habermann and U. Montanari, editors, *Proceedings Capri '86 Conference on Innovative Software Factories and Ada, Lecture Notes on Computer Science*. Springer-Verlag, 1986.

[11] UNU/IIST is a Research and Post-graduate & –doctoral Training Centre whose financial basis has been provided, 1992–1996, by The Republic of Portugal (US $ 5 mio), The [then] Portuguese administrated Territory of Macau (US $ 20 mio), and The People's Republic of China (US $ 5 mio).

[12] The author, as first and founding Director, Prof. Zhou Chaochen (then Principal Research Fellow, now Director), Søren Prehn, Chris W. George, Dr. Xu Qiwen, Dr. Richard C. Moore, Dr. Tomasz Janowski, and Dr. Cornelis A. Middelburg.

[12] Dines Bjørner. Software Development Graphs — A Unifying Concept for Software Development? In K.V. Nori, editor, *Vol. 241 of Lecture Notes in Computer Science: Foundations of Software Technology and Theoretical Computer Science*, pages 1–9. Springer-Verlag, Dec. 1986.

[13] A.P. Ravn and H. Rischel. Requirements capture for embedded real-time systems. In P. Borne, editor, *IMACS-IFAC Symposium MCTS, Villeneuve d'Ascq, France, May 1991*. IMACS Transaction Series, 1991.

[14] Jens U. Skakkebaek, Anders P. Ravn, Hans Rischel, and Zhou ChaoChen. Specification of Embedded, Real-time Systems. Technical report, Dept. of Computer Science, Technical University of Denmark, EuroMicro Workshop on Formal Methods for Real-time Systems, 1992 December 1991. The example: A railway road/rail crossing.

[15] Jens Ulrik Skakkebaek. Development of Provably Correct Systems. Technical report, Dept. of Computer Science, Technical University of Denmark, 30 August 1991 M.Sc. Thesis.

[16] Dines Bjørner. A ProCoS Project Description. *Published in two slightly different versions: (1) EATCS Bulletin, October 1989, (2) (Ed. Ivan Plander:) Proceedings: Intl. Conf. on AI & Robotics, Strebske Pleso, Slovakia, Nov. 5-9, 1989, North-Holland* , Dept. of Computer Science, Technical University of Denmark, October 1989.

[17] Jess Stein (Ed.). *The* Random House *American Everyday Dictionary*. Random House, New York, N.Y., USA, 1949, 1961.

[18] Chris George, Anne Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, 1995.

[19] Chris George, Peter Haff, Klaus Havelund, Anne Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[20] Dines Bjørner. Domain Modelling: Resource Management Strategics, Tactics & Operations, Decision Support and Algorithmic Software. In J.C.P. Woodcock, editor, Millennial perspectives in computer science, Palgrave, 2000.

[21] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.

[22] Zhou Chaochen and Li Xiaoshan. A Mean Value Duration Calculus. Research Report 5, UNU/IIST, P.O.Box 3058, Macau, March 1993. Published as Chapter 25 in *A Classical Mind*, Festschrift for C.A.R. Hoare, Prentice-Hall International, 1994, pp 432–451.

[23] Zhou Chaochen, Anders P. Ravn, and Michael R. Hansen. An Extended Duration Calculus for Real-time Systems. Research Report 9, UNU/IIST, P.O.Box 3058, Macau, January 1993. Published in: *Hybrid Systems*, LNCS 736, 1993.

[24] Zhou Chaochen. Duration Calculi: An Overview. Research Report 10, UNU/IIST, P.O.Box 3058, Macau, June 1993. Published in: *Formal Methods in Programming and Their Applications*, Conference Proceedings, June 28 – July 2, 1993, Novosibirsk, Russia; (Eds.: D. Bjørner, M. Broy and I. Pottosin) LNCS 736, Springer-Verlag, 1993, pp 36–59.

[25] Zhou Chaochen, Zhang Jingzhong, Yang Lu, and Li Xiaoshan. Linear Duration Invariants. Research Report 11, UNU/IIST, P.O.Box 3058, Macau, July 1993. Published in: Formal Techniques in Real-Time and Fault-Tolerant systems, LNCS 863, 1994.

[26] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), Aug. 1978.

[27] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[28] A.W. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[29] Michael A. Jackson. Description is Our Business. In *VDM '91: Formal Software Development Methods*, pages 1–8. Springer-Verlag, October 1991.

[30] Pamela Zave and Michael A. Jackson. Techniques for partial specification and specification of switching systems. In S. Prehn and W.J. Toetenel, editors, *VDM'91: Formal Software Development Methods*, volume 551 of *LNCS*, pages 511–525. Springer-Verlag, 1991.

[31] Michael A. Jackson. Problems, methods and specialisation. *Software Engineering Journal*, 9(6):249–255, November 1994.

[32] Michael A. Jackson. *Software Development Method*, chapter 13, pages 215–234. Prentice Hall Intl., 1994. Festschrift for C. A. R. Hoare: *A Classical Mind*, Ed. W. Roscoe.

[33] Pamela Zave and Michael A. Jackson. Where do operations come from? a multiparadigm specification technique. *IEEE transactions on software engineering*, 22(7), July 1996.

[34] Pamela Zave and Michael A. Jackson. Four dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.