# A Unifying Model for Specification and Design

J. Paul Gibson and Dominique Méry*
Université Henri Poincaré, Nancy 1
CRIN-CNRS URA 262, Nancy
email : {gibson, mery}@loria.fr

May 6, 1996

**Abstract**

The application of formal languages in the software development process is becoming more and more evident. Providing formal semantics and tools for the synthesis, analysis and transformation of behavioural models is usually the first step in the process of formal methods development. Many formal methods exist but, as yet, there is an absence of a meta-theory of *formal methods*. Such a meta-theory is the subject of this paper: we call it a *unifying framework*.

We present a generalisation of the software development model which reflects the standard approach of using different languages at different stages of development. A *unifying model* will give a better understanding of *why* and *how* this happens; together with strengthening the rigour of such standard *multi-semantic* approaches to software development.

## 1 Introduction

The purpose of this paper is to generalise the software development model to reflect the fact that different languages appear at different stages of development, to facilitate different levels of abstraction and different semantic capabilities. Clearly we need a better understanding of *why* and *how* this happens, in order to improve the rigour and formality in our development method. The paper introduces a unifying model for expressing three different types of development step: *specification refinement*, *program refinement* and *transformational refinement*. The model is shown to be generally applicable to any pair of *specification* and *implementation* languages, and, furthermore, can be extended to any number of development languages working in one coherent framework. We instantiate the unifying model in two distinct development frameworks, as a means of illustrating its utility. Firstly, an object oriented development method, in which ADTs and process algebras perform the roles of our different semantics, is examined. Secondly, we illustrate the model with respect to a development method which moves from temporal logic specifications to UNITY implementations.

The unifying model is schematically represented by the diagram below[1]:

The main idea of the paper is to present some initial theory which is shown to have a semantical basis in our different frameworks. We also discuss extensions to this work (which is at a preliminary stage) and their potential benefits to a formal development strategy. In large projects, in particular, where analysis, design and implementation are carried out concurrently, it is vital that we have a formal basis for relating changes in one level of development to levels of development further on in the process.

The paper is organized as follows. In the section that follows, we shall discuss the OO ADT approach with respect to the unifying schema. In the third section, UNITY and TLA like approaches are introduced and the

*This work is supported by the Institut Universitaire de France and the HCM Scientific Network MEDICIS (CHRX-CT92-0054). Surface mail : Bâtiment LORIA, BP 239, 54506 Vandœuvre-lès-Nancy, France. Phone : +33 83 41 30 79 and Fax : +33 83 41 30 79.

[1]The diagram is actually a simplification of the model as it represents only one branch of a *specification-implementation* pair-tree, where an implementation can act as the specification being passed to one single implementation framework. In the complete model, each specification framework may have more than one implementation framework at a lower level of abstraction.

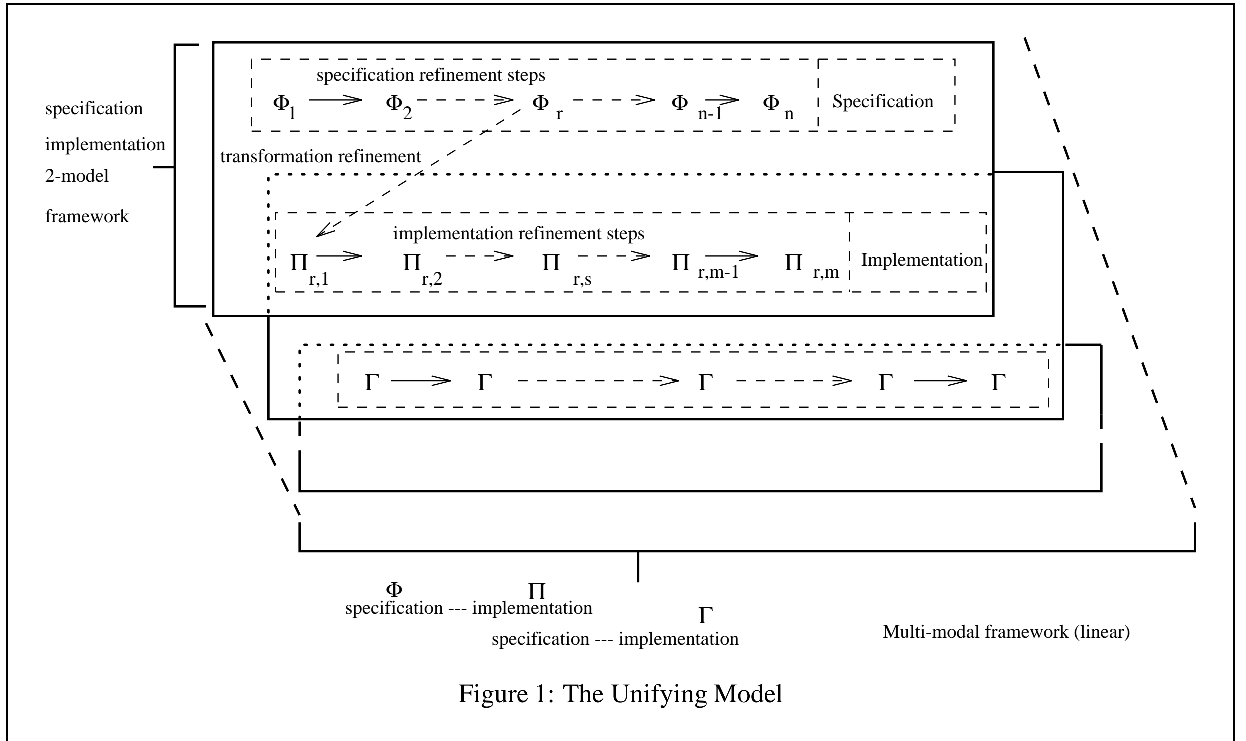


Figure 1: The Unifying Model

instantiation of the schema is outlined. The fourth and concluding section briefly discusses the general model, remarks on the design process and hints at future work within a service specification framework.

# 2 Object Oriented Frameworks for Requirements Capture and Design

## 2.1 Introducing FOOD(Formal Object Oriented Development)

[11] presents an Object-Labelled State Transition System (O-LSTS) model which can be used at all stages of software development (from analysis and requirements capture to design and implementation). The model is constructive and promotes the rigorous software engineering of abstract behaviour. What is interesting, from the point of view of finding a unification model for specification and design, is the way in which different languages are used to implement the O-LSTS models at different stages of the development. This is illustrated in the figure below:

The figure shows only that two different languages are used at different stages of development. We now present a short explanation of *why* and *how* this is done.

### Analysis and Requirements Capture

O-LSTS specifications are written in OO ACT ONE (a language whose syntax is similar to the ADT ACT ONE). Dynamic behaviour is defined by state transition diagrams. Analysis and execution of an O-LSTS requirements model is facilitated by a translation to ACT ONE [9], which is used to implement the O-LSTS semantics. The requirements model says *what* is required rather than *how* it is to be achieved in the final implementation. The notion of subclassing provides us with a formal relationship between models (class specifications). Subclassing is a *specification refinement* in our unifying model.

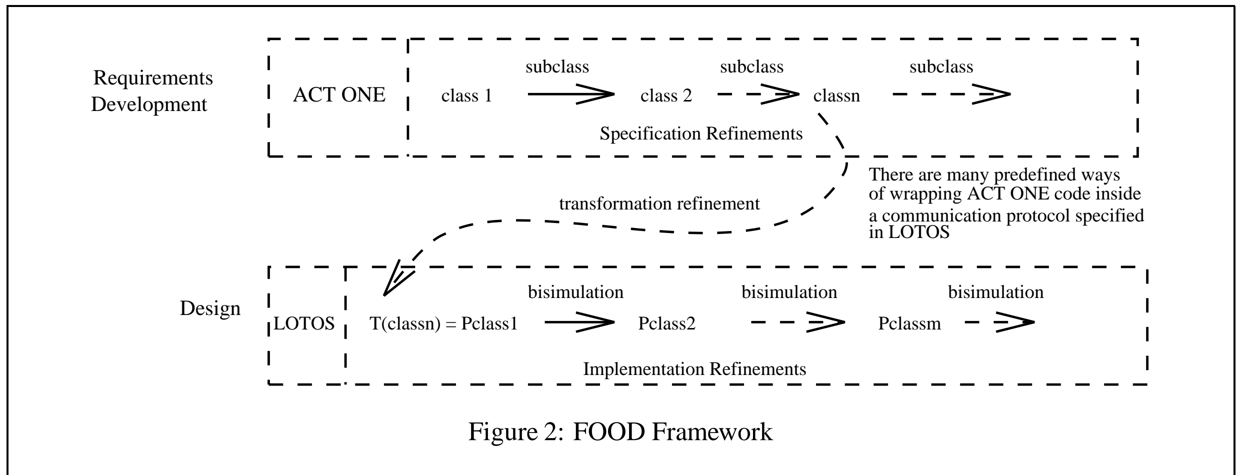


Figure 2: FOOD Framework

**Moving to Design**

The step from analyis to design requires an extension to the requirements model to incorporate semantics for object communication, concurrency and interaction, which are abstracted away from in the initial OO ACT ONE models. A process algebra provides a suitable formal model for the specification of these additional properties. LOTOS [21], which combines ACT ONE and a process algebra in one coherent semantic model, provides a means of constructing object oriented design semantics. The first step from an OO ACT ONE requirements model to a full LOTOS design can be done in many ways. This initial step is used to specify the protocol semantics of the objects in our requirements model (i.e. how we communicate with objects in order to use the functionality they provide). This first design step corresponds to a *transformational refinement* in our unifying model.

**Moving to Implementation**

Design must be targetted towards a particular implementation environment. With LOTOS, this is done by refining the specifications so that each step in the design process is *correctness preserving*. There are a number of useful formalisms of *correctness* which are based on the notion of equivalence (strong bisimulation equivalence, weak bisimulation equivalence, testing equivalence etc ...). Within the O-LSTS model of FOOD, there are a number of predefined *correctness preserving transformations* which can be used to step design towards particular resources in the implementation environment[2]. These CPTs are *implementation refinements* in our unifying model.

## 2.2 A simple FOOD example

Consider the specification of a simple Stack providing LIFO behaviour on a store of Natural numbers. This is specified using OO ACT ONE and results in the following ACT ONE code:

```
Type Stack USING Nat Is Stack
OPNS empty:  -> Stack (* LITERAL *)
Str:  Stack,Nat -> Stack (* STRUCTURE *)
head:  Stack -> Nat, tail:  Stack -> Stack (* ACCESSORS *)
add:  Stack, Nat -> Stack (* TRANSFORMER *)
EQNS forall Stack1:Stack, Nat1:Nat
head(Empty) = ErrorNat; head(Str(Stack1,Nat1)) = Nat1;
tail(empty)= empty; tail(Str(Stack1,Nat1))= Stack1;
add(Stack1, Nat1)= Str(Stack1,Nat1);
ENDCLASS Stack
```

[2] *Correctness* in this case implies a weak bisimulation equivalence between models.

Consider now a step in the requirements process in which the customer expresses the need for additional behaviour so that the Stack can return the number of elements presently stored (we shall call this new service `size`). This is easily done in OO ACT ONE by defining the equations:

```
size(empty) = 0; size(Str(Stack1,Nat1)) = 1+Stack1.size.
```

Now, we have a subclassing relationship between a `Stack` and a `SizedStack`. We return to this in the next section, which considers the `Stack` design.

### LOTOS: ACT ONE + process algebra

Given the `Stack` requirements, how do we move to the initial design? There are many different translation mechanisms for producing full LOTOS from our OO ACT ONE models [12]. The simplest is the transformation which produces a *remote procedure call* protocol semantics. The full LOTOS thus produced contains the ACT ONE code together with the following process algebra code:

```
PROCESS Stack[head,tail,add](Stack1:Stack):noexit:=
(head; head!head(Stack1); Stack[...](Stack1) )[]
(tail; tail!tail(Stack1); Stack[...](Stack1) )[]
(add?Nat1:Nat; Stack[...](add(Stack1,Nat1)) )
ENDPROC (* Stack *)
```

The same *translation refinement* applied to the `SizedStack` ACT ONE specification produces exactly the same process algebra specification, as the `Stack`, except that there is an additional choice of events to include the `size` service.

```
...)[] (size; size!size(Stack1); Stack[...](Stack1))[]( ...
```

A simple design step (or *implementation refinement* in the unifying model) is to introduce a variable which remembers the number of elements on the `SizedStack` instead of counting them every time a `size` request is receive.

### Concurrent Development

Given the `SizedStack` requirements, the implementation can procede at the same time as the requirements are refined. Typically, we reach a stage where we must integrate *specification refinements* and *implementation refinements* in a meaningful (and *correct*) way. This is illustrated in the diagram below:

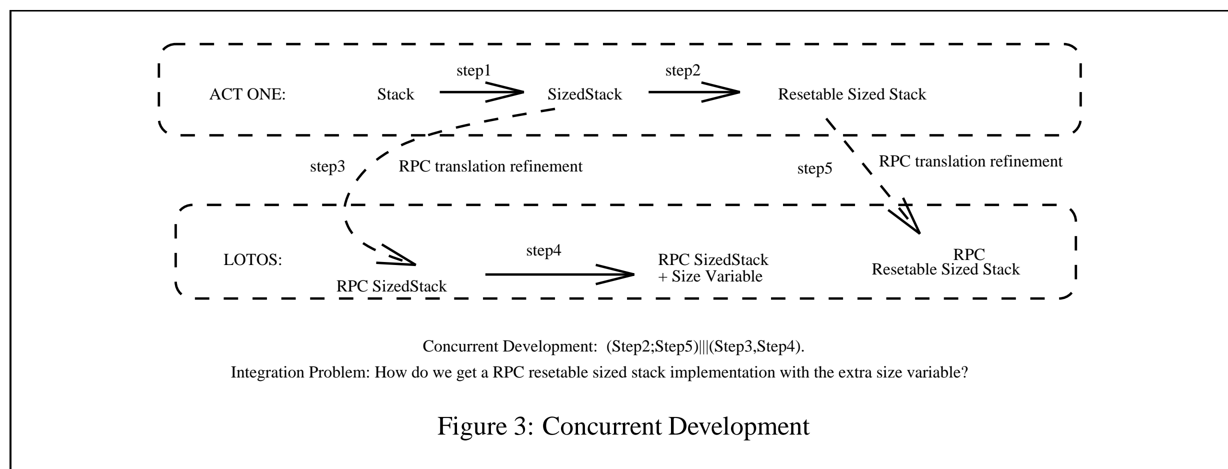


Figure 3: Concurrent Development

The question is one of incremental development and is particularly important in systems where requirements are continually being added in a modular fashion and implementations exist in many different forms. Feature interaction in telecommunication systems is one such problem domain which would clearly benefit from a *unifying theory*.

# 3 Temporal Frameworks for Program Specification and Program Design

Algebraic approaches, such as the OO LOTOS method seen in section2, are based on bisimulation relationships that allow one to show that terms are equivalent according to some correctness criterium. If we consider a program we would like to compare it with another program that is *better* with respect to some implementation environment. We have to define a relationship over programs and more generally we have to include specifications in the same framework. Category theory for studying combination of temporal theories, and its techniques of superposition are useful for transforming programs by preserving correctness properties. The seminal work of Back [2, 3] has founded the refinement calculus that is a transformational calculus over action systems in which the transformation preserves total correctness. The UNITY [8] approach is based on a mixed calculus that includes refinement calculus over action systems under weak fairness and refinement calculus over a temporal specification (the refinement preserves safety properties and eventuality properties under the weak fairness assumption). In the case of TLA [13], refinement is expressed as logical implication between TLA formulae. A general language expresses development steps and a correctness criterium is based on the relationship between proofs and programs: such a relationship defines a means of constructing proofs which address *invariance* and *eventuality* properties. The underlying operational model does not support any notion of *time* or *efficiency*.

The instantiation of the unifying model is a way to explain UNITY [8] but also TLA [13] and Action systems [2, 3, 5]. The reasoning exploits two different languages : a (temporal) specification language and a programming language based on action systems. The diagram of the figure 4 describes the relationship between the different languages.

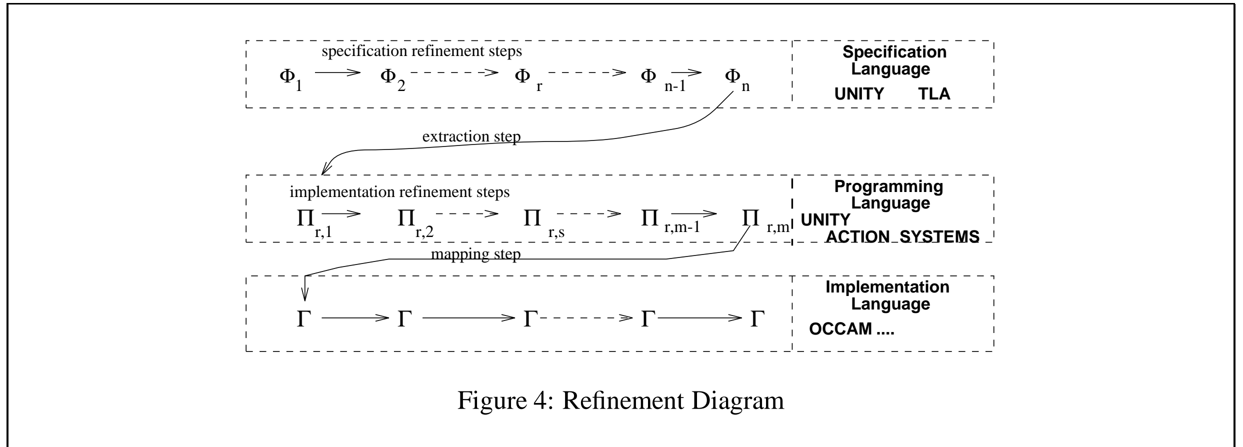


Figure 4: Refinement Diagram

- a purely logical refinement step in a temporal language (as TLA, UNITY) : the refinement process is defined as a deduction process and uses temporal rules.
- a purely transformational (or programming) refinement step in an action-based programming language (as action systems) : the refinement process transforms action systems and preserves properties as invariance, total correctness; the semantics of this level is related to the previous one by the notion of predicate tranformer but technical problems appear when dealing with fairness; superposition is a technique in this level.
- a mapping refinement relates the programming level and the implentation level : in fact, the third level gathers techniques of compiling and is the way to get real programs.
- Finally, there are steps that allow one to transit from one level to another one : an extraction step produces an action system from a formal specification and a mapping step produces a code from a more abstract programming language as action systems.

UNITY covers the different levels of refinement and it requires different semantics. Superposition techniques can be improved and enriched but we think that they are domain-driven.

## Refining in a purely logical language

The refinement of a temporal specification $\varphi_1$ into another temporal specification $\varphi_2$ is defined as an implication, namely $\varphi_2 \Rightarrow \varphi_1$. TLA and UNITY use the logical refinement, and the coding of a proof system for TLA [10] or UNITY [6] produces a refinement tool. In fact, the logical refinement is defined as a goal-directed proof process [15, 18].

**Property 1** *If $\varphi_1$ is refined into $\varphi_2$, then any property of $\varphi_1$ is a property of $\varphi_2$.*

A property of $\varphi_1$ is a formula $\varphi$ that is derived from $\varphi_1$ and it is clear that the logical refinement preserves properties. Different points are to be addressed. A first point is to get a useful proof system that is really useful in the refinement process. A second point is related to the power of the specification language:

- UNITY expresses safety properties but also eventuality properties under justice assumption.
- TLA expresses safety and eventuality properties, and supports the specification of different types and combinations of fairness.

The semantics of the refinement can be defined as the preservation of properties and provide a way to unify the logical refinement and the action refinement.

The logical (or specification) refinement relationship is the goal-directed proof method: with theorem-provers, such as Isabelle [20], implement our specification relationship. We have experimented with Isabelle to implement a proof system [18] for a kernel of SDL [7] and our tool can be useful to derive solutions described as a set of actions under nondeterministic fair choice. The crucial quality of the specification refinement is that it is based on a well-known and well-mastered technique (the deduction relation ).

## Refining actions and action systems

A program is a set of actions executed under some scheduling policy ( weak or strong fairness or general fairness); such a program is called an action system. The refinement calculcus of Back defines a refinement preserving total correctness of refined programs. The refinement is simply defined as follows: a program $\pi$ is refined into a program $\pi'$, if any correctness property holding for $\pi$ holds also for $\pi'$. Let us give several examples of this idea.

**Example 3.1 Action systems [2, 3, 5, 4, 16, 15, 17, 19]**
$\pi \sqsubseteq_{total-correctness} \pi'$ means that, for any $\phi$ of $\mathcal{L}$, $WP(\pi)(\phi) \Rightarrow WP(\pi')(\phi)$. If $\pi$ is totally correct with respect to *pre* and *post* conditions, $\pi'$ is totally correct with respect to *pre* and *post* conditions.
■

**Example 3.2 AMN [1, 14]**
$\pi \sqsubseteq_{invariant} \pi'$ means that any invariant of $\pi$ is an invariant of $\pi'$. As a first consequence, the strongest invariant of $\pi$ is weaker than the strongest invariant of $\pi'$. $\pi$ and $\pi'$ can contain different kinds of actions and actions may be fully different. Yet, practical refinement of programs is carried out by modifying one action at every step of refinement. A formal defintion of the invariant-preserving refinement is:

$$\pi \sqsubseteq_{invariant} \pi' \iff \left\{ \begin{array}{ll} \forall I: & ([Init_\pi \Rightarrow I] \wedge [\forall \alpha \in \pi : I \wedge cond(\alpha) \Rightarrow \alpha \bullet I]) \\ & \Rightarrow \\ & ([Init_{\pi'} \Rightarrow I] \wedge [\forall \alpha \in \pi' : I \wedge cond(\alpha) \Rightarrow \alpha \bullet I]) \end{array} \right\}$$

The transformations on programs leads to increase the number of actions or to strengthen actions of a program.
■

**Example 3·3 Unity [8]**
Unity exploits the superposition techniques to help the user to transform programs. A superposition rule adds a new action that does not modify the variables modified by the other previous actions : for instance, a timestample is added to the program but does not change other variables and the action is added as a concurrent action to every action. The fact that no previous variable is modidied guarantees that the previous properties are still holding.
■

### Relating specification and programs in our Unifying Model

When a program $\pi$ is developed from a specification $\varphi$, the relation can be verified by a proof or can be obtained by construction. The main problem is to identify a set of actions from a formal specification. This step is called an extraction and corresponds to a *transformation refinement* in our unifying model. If we consider a UNITY specification, we can find a specification of an action. The foundation of the relation is derived from the predicate transformers.

**Property 2**
1. *$\pi$ is refined into $\pi'$, if any property of $\pi$ is a property of $\pi'$ with respect to predicate transformers.*
2. *$\varphi_1$ is refined into $\varphi_2$, if $\varphi_2 \Rightarrow \varphi_1$.*
3. *$\pi$ $sat$ $\varphi$, if any model of $\pi$ is a model of $\varphi$.*

In fact, the predicate transformers are used to build proof systems for invaraiance and eventuality properties. In this way we can show that the three refinements are equivalent with respect to the preservation of correctness properties. Moreover, proof systems are used to refine in the logical language [17].

**Example 3·4**
$\pi \sqsubseteq_{invariant,eventuality} \pi'$ means that any invariant of $\pi$ is an invariant of $\pi'$ and any eventuality of $\pi$ is an eventuality of $\pi'$. We have not yet mentioned that this refinement is based on predicate transformer properties. In fact, if a predicate transformer $WP(\pi)$ is defined for every $\pi$, then the preservation of the eventuality properties is ensured by stating: $WP(\pi) \Rightarrow WP(\pi')$ and $I_{\pi'} \Rightarrow I_\pi$ states the preservation of invariants.
■

# 4 Conclusion

We have presented only an introduction to our *unifying model* and commented on the need for a meta-model of software development. Two different formal development frameworks have been shown to be particular instantiations of our model. This is just a beginning.

We aim to continue the work in two orthogonal directions. Firstly, we must build a sound theory for the *unifying model* (we believe that category theory provides a suitably powerful semantic basis upon which to start constructing a theory of relations between languages and language refinements). Secondly, we must perform an analysis of different software development methods to identify common patterns of refinement which occur in *multi-modal* approaches. The formulation of a theory of *refinement patterns* is a long term goal.

Already we have identified the need for a unifying theory in the domain of telephone service (and feature) development. The specifications and implementations are arried out in different semantics frameworks and requirements are continually incremented. We hope that the unifying model will provide us with a better understanding of the feature interaction problem which is currently being addressed [22].

# References


[1] J. R. Abrial. *The B book*. Cambridge University Press, 1996. to appear.

[2] R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.

[3] R. J. R. Back. Correctness preserving programs refinements: proof theory and applications. Mathematical Centre Tracts 131, Mathematical Centre, Amsterdam, 1980.

[4] R. J. R. Back and K. Sere. Deriving an occam implementation of action systems. In C. Morgan and J. C. P. Woodcock, editors, *4rd Refi nement Workshop*. Springer-Verlag, january 1991. BCS-FACS, Workshops in Computing.

[5] R. J. R. Back and J. von Wright. A lattice-theoretical basis for a specification language. In J. L. A van de Snepscheut, editor, *Mathematics for Program Construction*, pages 139–156. Springer-Verlag, june 1989. LNCS 375.

[6] N. Brown and D. M´ery. A proof environment for concurrent programs. In J. C. P. Woodcock, editor, *FMÉ93: Industrial-Strength Formal Methods*, pages 196–215. IFAD, Springer-Verlag, 1993. LNCS 670.

[7] CCITT. Recommendation z. 100 specification and description language sdl. Note, 1988.

[8] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[9] H. Ehrig and Mahr B. *Fundamentals of Algebraic Specifi cation I*. Springer-Verlag, Berlin, 1985. EATCS Monographs on Theoretical Computer Science (6).

[10] U. Engberg. *TLP Manual-(release 2. 5a)*-PRELIMINARY. Department of Computer Science, Aarhus University, May 1994.

[11] J. Paul Gibson. Formal object based design in LOTOS. Tr-113, University of Stirling, Computing Science Department, Stirling, Scotland, 1994.

[12] J.Paul Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS*. Tech. report csm-114, Stirling University, August 1993.

[13] L. Lamport. A temporal logic of actions. *Transactions On Programming Languages and Systems*, 16(3):872–923, May 1994.

[14] K. Lano. *The B Language and Method: A Guide to Practical Formal Development*. FACIT. 1996.

[15] D. M´ery. The $\backslash\sqcap$ system as a development system for concurrent programs: $\delta\backslash\sqcap$. *Theoretical Computer Science*, 94(2):311 – 334, march 1992.

[16] D. M´ery. Proving and developing concurrent programs: a small system. In C. M. I. Rattray and R. G. Clark, editors, *Proceedings of the IMA conference on the Unifi ed Computation Laboratory*. The IMA, OXFORD UNIVERSITY PRESS, 1992.

[17] D. M´ery. *Une Méthode de Raffi nement et de Développement pour la Programmation Parallèle*. Doctorat ès sciences math´ematiques, Universit´e de Nancy 1,UFR STMIA, DFD Informatique, F´evrier 1993.

[18] D. M´ery and A. Mokkedem. Crocos: An integrated environment for interactive verification of sdl specifications. In G. Bochmann, editor, *Computer-Aided Verifi cation Proceedings*. Springer Verlag, 1992.

[19] C. Morgan. *Programming from Specifi cations*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[20] L. Paulson. The Isabelle Reference Manual. Technical report, University of Cambridge, Computer Laboratory, 1992.

[21] K.J.T. Turner. *Using FDTS: An Introduction To ESTELLE, LOTOS and SDL*. John Wiley and Sons, 1993.

[22] P. Zave. Feature interactions and formal specifications in telecommunications. *Computer*, August 1993.