A Formal Approach to Determining Parallel Resource Bindings

Experience Report

John A. Keane,
Centre for Novel Computing,
Department of Computer Science,
The University, Manchester, UK.

Walter Hussak,
Department of Computer Studies,
University of Technology,
Loughborough, UK.

Abstract

This paper investigates the nature of the design process for parallel operating systems. It proposes a temporal logic-based formal methodology addressing the high-level design of such systems. In operating systems design much use is made of the informal notion of resource bindings. *A way of improving the high-level design of parallel systems is proposed by providing a formal language for enumerating the design space and thus enabling all high-level design alternatives to be represented. A design process to be used with this language is given, the aim being to establish the* most appropriate *binding. The process is temporal logic-based and permits high-level design of parallel systems to be analysed, tested and, in certain cases, formally verified before implementation is embarked upon.*

1 Introduction

Successful industrial applications of formal methods, such as the IBM CICS project using Z [7], have increased interest in applying formal methods in the development cycle of real-life systems. Various uses of formal methods have been proposed for different stages of the development process. With parallel systems, the added complexity and design options have resulted in a greater demand for some sort of assistance. The required assistance is twofold. Firstly, to be made more aware of the design alternatives and secondly, to obtain some sort of verification or assurance that the design is viable before embarking on the costly task of implementation. Formal techniques have been used in a limited way to offer verification in real-life parallel systems; see for example [3], [6], [11] and [9]. These rarely address the concern of giving guidance as to the design alternatives. Furthermore, some only address a fairly low level of design.

In this paper, a formal design process for high-level design is proposed which enables the high-level design space to be enumerated, and which provides limited verification of chosen design options.

In section 2 the experiences of two parallel operating systems projects, both collaborative with industry, are drawn upon to extract the essential features of high-level design. From this, a design space language is formulated in section 3. A design process to be put in place at the high-level design phase of system development is given in section 4. Section 5 gives examples of use of the design process including use of temporal logic to provide partial verification of the design. Some conclusions are drawn in section 6.

2 Operating Systems Design

The behaviour of an operating systems is described, at a high-level by systems architects, in terms of an intuitive notion of *bindings*[1]; see for example [13], [15], [14], [16], [4] and [12]. An operating system can be viewed as handling an infinite set of requests. These requests take the form of "provide access to resource(s)". The system ensures that this request is satisfied by providing an "association" between the request and the resource(s). In informal design, such an association is termed a *binding* between the request and the resource(s). It is the set of such permissible associations/bindings that can occur that define the behaviour of a system, and indeed that indicate the possible design structuring for that system.

The design phase of the operating systems projects that initiated this work was based around what bindings were appropriate. The designers had little specific

[1] The term *binding* pervades the computer world and assumes a slightly special meaning in each case. This ranges from its usage in the purest sense, as in the binding of names to values in the λ-calculus, to its usage in the present context of operating system design.

0270-5257/94 $3.00 © 1994 IEEE

formal methods experience. Consequently the discussion was informal and assumed an intuitive notion of what was meant by binding.

The Flagship operating system [10] was decomposed into a four layer model of *abstract machines (am)* such that level am_n, above, depended upon, the services of am_{n-1}. The levelled composition of the abstract machines was expected to provide the required behaviour of the system. In turn, the abstract machines were decomposed informally into *subsystems*. In the kernel *am*, these subsystems were *resource managers*, corresponding to the resources of the machine.

In the kernel, the building block for the resource managers was the Flagship ADT (FADT) which provided serialisable operations on its state. A call to an operation of an FADT required a binding of five components. The ordering in which these bindings were to occur would have an impact on the design and decomposition of the resource management components. It is precisely this sort of issue that would benefit most from early formalisation of design alternatives. This example is returned to in section 5 where a formal design language is used to express the intuitive notion of the *most appropriate* binding strategy.

So far, in the Flagship operating system, the system specification had been informal. At the level of the subsystems formality was introduced by specifying behaviour in a VDM-like relational model. Whilst such specification proved useful it did not involve any formal correspondence to higher levels as no formal definition of the overall system behaviour was available[2].

The system design of the EDS operating system project began to address some of the remaining inadequacies that were present in the Flagship project. Temporal logic was introduced to express required properties such as liveness. A temporal description of a scheduler was produced [8], and an informal proof that the implementation of the scheduler satisfied its specification was also carried out.

It was clear that a formal design process had to be put in place that would also provide some sort of formal test of the viability of the high-level design options. The process, described below, is aimed primarily at the high-level design phase of parallel systems. Notation that allows the intuitive notions of bindings to be placed in a formally enumerated design space is introduced and a temporal logic-based process for testing the design is proposed.

[2] An attempt at a formal argument for the system's behaviour is given in [9].

3 Binding Strategy Language

Given identified resource types, a notation is needed which will indicate all the relevant ways that the resources can be bound, in other words the *binding strategies*. In view of the remarks above, a function application style of notation might be appropriate. Thus, below,

$$R_1(R_2)$$

has the intuitive meaning of R_1 *being bound to* R_2 *applicatively*. There is to be a distinction between

$$R_1(R_2) \text{ and } R_2(R_1).$$

With parallel systems, the notion of the order in which resources are bound is important. To this end,

$$R_1; R_2$$

suggests R_1 *followed by* R_2, and

$$R_1 \| R_2$$

indicates R_1 *and* R_2 *in parallel.*

The binding strategies given below are very general. The form of the binding strategies are encapsulated in the *Binding Strategy Language (BSL)*. The BSL has the following additional characteristics:

1. There is a notion of resource type

   $$R_1,\ R_2,\ \ldots,$$

   which are sets of instances of the respective resource types. Thus, R_1 might be the set of processes, and R_2 might be the set of parts of state which can be accessed individually. Lower-case letters will denote particular instances of resource types, such as an instance of a printer resource type. Thus,

   $$R_i = \{r_{ij} : r_{ij} \in R_i\}.$$

2. The bindings should be able to cope with both static and dynamic binding of resources. If R_j is a resource type then, for each instance $r_{jk} \in R_j$, there may be many instances of it

   $$r_{j1},\ r_{j2},\ \ldots,\ r_{jl}$$

   say. Then, if a request to bind

$$R_i(R_j)$$

is received, this means that any of

$$r_i(r_{j1}),\ r_i(r_{j2}),\ \ldots,\ r_i(r_{jl})$$

is acceptable. It is also possible to bind to a dynamic resource type statically, say if in the above the user had specifically requested

$$R_i(r_{j3}).$$

The difference between the static and dynamic cases is that in the static case the instance of the resource type required is known at the outset.

The formal syntax of the BSL is as follows:

Syntax

Symbols

1. Sequence of symbols denoting resource types: $R_1,\ R_2,\ \ldots$.
2. For each resource type R_i, symbols denoting the resource instances: $r_{i1},\ r_{i2},\ \ldots$
3. Parentheses ( ), serial composition ;, parallel composition $\|$.

Expressions

The language BSL comprises applicative bindings, *bdapp*, bindings in serial, *bdser* and parallel bindings, *bdpar*. It is argued here that a series-parallel language of applicative bindings is sufficiently expressive for all sound designs of parallel systems. Below, the syntax of the language is given in BNF with non-terminals enclosed in angled brackets <, >.

$$\begin{aligned}
&\textbf{< bd >} && ::= \textit{< bdapp >} \mid \textit{< bdser >} \mid \\
&&& \quad \textit{< bdpar >} \mid \textit{< bdatom >} \\
&\textbf{< bdapp >} && ::= (\textit{< bd >}\textit{< bd >}) \\
&\textbf{< bdser >} && ::= (\textit{< bd >}; \textit{< bd >}) \\
&\textbf{< bdpar >} && ::= (\textit{< bd >} \| \textit{< bd >}) \\
&\textbf{< bdatom >} && ::= (R_1) \mid (R_2) \mid \ldots \mid (r_{11}) \mid (r_{12}) \mid \ldots \mid \\
&&& \quad (r_{21}) \mid (r_{22}) \mid \ldots
\end{aligned}$$

Thus, a typical expression might be

$$(r_{12})((R_2)\|(R_3))$$

where R_2 and R_3 are bound to dynamically.

Design Options Language

A given design options language corresponds to an integer n, to a finite subset of resource types $\mathcal{S}$

$$R_1,\ R_2,\ \ldots,\ R_n$$

and, for each such resource type R_i, a finite subset $\mathcal{S}_i$ of instances

$$r_{i1},\ r_{i2},\ \ldots,\ r_{ik_i}$$

The *Design Options Language*, $DOL_{\mathcal{S},\mathcal{S}_i}$, is defined to be the sublanguage of the overall BSL comprising expressions which have only one occurrence of the symbols belonging to the set of resource types $\mathcal{S}$ and each of the sets of resource instances $\mathcal{S}_i$, no symbols from any other resource type and only n resource type or resource instance symbols in total.

Design Options

A *design option* is a set of expressions in a given Design Options Language. A given DOL enables a representation of all the design options to be listed. Thus all the possible high-level designs can be given consideration at an early stage. The nature of the meaning attached to 'binding' will differ from application to application. For each particular application, this meaning will be given precisely by a temporal logic semantics. The intention of the DOL is not to provide such a semantics that applies strictly in all situations, but rather to ensure that, in the particular application, all design options are given some denotation and thus made apparent at an early stage. It is believed that in most cases a temporal semantics will be possible, and analysis, testing or even verification will be facilitated. This was found to be the case in the design of the EDS operating system. At the very least, it will provide testing of high-level design before the costly process of development is embarked on.

4 Design Process

BSL is to be used in the context of the following temporal logic-based design process. It is given in terms of an eight stage procedure to be put in place at the high-level design phase of parallel system development.

1. Identify the resource types in the system.
2. For each resource type determine the resource instances.
3. List the expressions in the corresponding DOL.
4. Conduct a preliminary and informal assessment of the design options and select a small subset of viable options.
5. Produce a corresponding temporal semantics $\mathcal{TL}_{op}$ for each design option *op* in this subset of the design options.
6. Give a temporal logic formula $\mathcal{TL}_{prop}$ for the required properties of the system.
7. Express any architectural constraints (e.g. hardware) as a temporal logic formula $\mathcal{TL}_{constr}$.
8. Test the design by proving absolutely, or proving with chosen finiteness assumptions, the formula: $\mathcal{TL}_{op} \wedge \mathcal{TL}_{constr} \Rightarrow \mathcal{TL}_{prop}$

5 Examples of using the design process

The following illustrates the use of the design process with real systems. As the temporal logic stages of the process are substantial only the first example covers these. The rest focus on BSL and how different strategic design decisions can be represented initially by very simple expressions.

Monolithic store management

This example is of a system with a monolithic virtual store shared by all the processes in the system. User processes read and write to virtual store by first accessing a store map to find which real address corresponds to a virtual address, and then to access the physical store at that real address.

The resource types are user processes $U = \{u(n) : u(n) \in U\}$, a single store map sm and a physical store ph. The viable designs are, after informal consideration of the possible BSL expressions,[3]:

- $((U)(sm))(ph)$[4] ~users access the store map first and then the physical store in two stages,
- $(U)((sm)(ph))$ ~ users access the store map and physical store in one go.

The temporal semantics of the two designs corresponding to the two BSL expressions are given by specifying each designed system as the sequence of events in time that can occur in that system.

The events that may or may not be occurring at a given point in time correspond to the following predicates:

- $sm(v, r)$: r is the real address of the virtual address v in the store map;
- $ph(r, x)$: x is the value contained at the location with address r in physical store;
- $u\_sm(n, v)$: user n is accessing the store map to get the real address of the virtual address v;
- $u\_ph(n, r)$: user n is accessing the physical store at real address r;
- $s\_sm(v)$: the system process (possibly hardware) accesses the store map at virtual address v for store management purposes;
- $s\_ph(r)$: the system process (possibly hardware) accesses the physical store at real address r for store management purposes;
- $u(n)$: user n is accessing either the store map or physical store.

It is clear that the value of the predicates sm, ph, u_sm, u_ph, s_sm, s_ph and u are required to change over time. For this reason, the temporal language to be used is first-order linear discrete time temporal logic with equality where the predicates are 'flexible' in the terminology of [1]; see that reference for the full semantics. A pleasant feature of temporal logic is that specifications can be appreciated with only the following brief description of the temporal operators:

1. $\Box\phi$ ("always ϕ") is true if the formula ϕ is true now and in every future moment.
2. $\bigcirc\phi$ ("at the next moment ϕ") is true now if the formula ϕ is true at the next instant.

[3] The informal idea of 'binding' in this case is seen applicatively and hence the BSL expressions given. It is possible to see the idea of 'binding' here as serial association and hence use BSL expressions with ; instead. Whatever scheme is used in any particular application does not matter as all possible designs will be accounted for by *some* expression. It is a lack of awareness of the possibilities that leads to bad designs initially. With BSL the options are made apparent early on.

[4] Strictly speaking the corresponding design option is the singleton set $\{((U)(sm))(ph)\}$.

3. $\phi \mathcal{U} \psi$ ("ϕ until ψ") is true if ψ becomes true at some point in time and *until* that happens ϕ will be true.

The temporal semantics of the two chosen design alternatives are:

- $\mathbf{bind}_{((U)(sm))(ph)} \stackrel{def}{=} \forall n \forall v \forall r.$
  $\Box\,(u\_sm(n,v) \wedge sm(v,r) \Rightarrow \bigcirc(\neg u(n)\,\mathcal{U}\,u\_ph(n,r)))$

This states that for any user n, any virtual address v and any real address r, it is always the case that if user n accesses the store map and finds that the virtual address v is at real address r, then the next access that this user performs is to the physical store at that real address. The other binding has the temporal expression:

- $\mathbf{bind}_{(U)((sm)(ph))} \stackrel{def}{=} \forall n \forall v \forall r.$
  $\Box\,(u\_sm(n,v) \wedge sm(v,r) \Rightarrow \bigcirc u\_ph(n,r))$

A user accesses physical store as soon as (conceptually at the next instant in time) the real address of the physical address is obtained. The architectural constraints are broken down into *constr*1, *constr*2 and *constr*3 which, respectively, allow access to the store map and physical store by only one process at a time (*constr*1), do not allow a simultaneous access by a user process and the system process to either the store map or physical store (*constr*2) and which permit changes to the store map by the system process for store management purposes (*constr*3).

- $\mathbf{constr1} \stackrel{def}{=} \forall m \forall n \forall v_1 \forall v_2 \forall r_1 \forall r_2.$
  $(m \neq n) \Rightarrow \Box\,(\neg(u\_sm(m,v_1) \wedge u\_sm(n,v_2)) \wedge$
  $\neg(u\_ph(m,r_1) \wedge u\_ph(n,r_2)))$
- $\mathbf{constr2} \stackrel{def}{=} \forall n \forall v_1 \forall v_2 \forall r_1 \forall r_2.$
  $\Box(\neg(u\_sm(n,v_1) \wedge s\_sm(v_2)) \wedge \neg(u\_ph(n,r_1) \wedge s\_ph(r_2)))$
- $\mathbf{constr3} \stackrel{def}{=} \forall v \forall r_1 \forall r_2.$
  $(r_1 \neq r_2) \Rightarrow \Box\,((sm(v,r_1) \wedge \bigcirc sm(v,r_2)) \Rightarrow \bigcirc\, s\_sm(v))$

The last constraint reads: "whenever the store map changes, the system process must have accessed it". Then,

- $\mathbf{arch\_constr} \stackrel{def}{=} constr1 \wedge constr2 \wedge constr3$

The required property of the system is that the store is kept consistent. This means that if a user process accesses the store map to obtain the real address r of a virtual address v, then when that user accesses physical store at that r it must still be the real address of v:

- $\mathbf{prop} \stackrel{def}{=} \forall n \forall v \forall r.\ \Box\,(u\_sm(n,v) \wedge sm(v,r) \Rightarrow$
  $\bigcirc(\neg u(n)\,\mathcal{U}\,(u\_ph(n,r) \wedge sm(v,r))))$

The proof obligations for the two design options are:

- $\mathbf{bind}_{((U)(sm))(ph)} \wedge \mathbf{arch\_constr} \Rightarrow \mathbf{prop}$

and

- $\mathbf{bind}_{(U)((sm)(ph))} \wedge \mathbf{arch\_constr} \Rightarrow \mathbf{prop}$

Unfortunately, proofs are not possible in the temporal logic used. However, with suitable finiteness restrictions on the set of users, virtual addresses and real addresses, the formulae reduce to formulae in propositional temporal logic which certainly admit proofs, indeed the logic is decidable and is the focus of much active research into efficient theorem provers, e.g. [2]. With the inclusion of additional temporal formula that have been omitted here, giving fairly obvious relationships between the predicates, it is possible to formally verify that the second bindings option has the correct properties. Although, with the finiteness assumptions, this would not be an absolute proof, nevertheless it does give some indication of the correctness of the high-level design as was sought.

Distributed Store Management

The monolithic store example hardly requires a temporal verification to show that the second binding alternative is the one to choose. The following, more interesting distributed store example can be dealt with in exactly the same way though at greater length.

It comprises virtual storage distributed between several processors. The resources in the system are user processes $P = \{p_n : p_n \in P\}$, a (global) table PE indicating on which processor each p_n's store is located, for each process p_n a store map sm_n giving the virtual to real address map on the processor that p_n is on and, for each processor pr a physical address to value map ph_{pr}. One possible design option for a process updating its virtual store, is that the process p_n first accesses PE to find which processor it is on, and then accesses sm_n to obtain the real address; in BSL: $((p_n)(PE))(sm_n)$.

An alternative binding is if the process first accesses the virtual to real address map sm_n and then finds which processor this relates to; in BSL: $((p_n)(sm_n))(PE)$.

If the hardware moves processes from one processor to another so that the virtual to real address map does not change, then the second bindings option would be the one giving a consistent store. Bindings options other than the ones shown are possible.

Flagship ADTs

In section 2, the Flagship operating system was discussed and in particular the FADT components of the kernel. It was essential to ensure serialisability of the state of these FADTs in the presence of parallelism. A call to an operation of an FADT required a binding of five components: a *stateholder*, *sh*, which was bound to a *processor*, PE_i, and could not be copied (but could, conceptually, be moved to another processor); a *guardian*, *g*, that ensured only one operation of the FADT ever has access to the stateholder at any instant; an operation, *op*, called by the user, and partial parameters to the operation, *pp*, supplied by the user. As mentioned earlier, the order in which these bindings are carried out has an impact on system structuring. It is precisely the type of intuition involved in choosing the *best* binding order that the BSL attempts to make formal.

The following conditions were considered to be needed for serialisability:

1. the presence of the guardian, that ensures only one operation has access to the stateholder at any instant,

2. the stateholder is bound to only one processor at any instant, which ensures there is only ever one copy of the stateholder extant.

The operation, *op*, and the partial parameters, *pp*, are only bound to the binding of (PE_i, *sh*, *g*) on each call. So the major concern is the order of binding of the three components, (PE_i, *sh*, *g*).

Two possible bindings seem intuitively to be worthy of consideration. The BSL framework allows these alternatives to be investigated more formally.

The stateholder must exhibit serialisable behaviour, enforced by the guardian. A single PE must be bound in order to achieve this. The two alternatives of interest are as follows:

1. The binding of a processor is of secondary importance, because, as noted, conceptually, the stateholder could be re-bound to another processor at some instant and the same *abstract* behaviour will ensue. Thus, the binding of longest duration is between the stateholder and the guardian.

   This intuition can be formalised using BSL: a stateholder and a guardian are bound in any order. The design options are: $\{((g);(sh)), ((sh);(g))\}$. This binding holds for the lifetime of the FADT.

   A processor, PE_i, is then bound, for example, $(PE_i)((g);(sh))$. This binding may be changed during the lifetime of the FADT.

2. The *closest* binding is between the processor and the stateholder. The guardian, to queue operations on the stateholder, is of secondary importance and can be bound later.

   This intuition can be formalised using BSL: a stateholder and a processor are bound in any order. The design options are: $\{((PE_i);(sh)), ((sh);(PE_i))\}$.

   A guardian *g*, is then bound, for example, $(PE_i)((sh);(g))$.

   The difficulty that can now be seen with this approach is that conceptually the stateholder can be moved to another processor during it's lifetime. To achieve this type of re-binding dynamically is made far more difficult because it also involves unbinding the guardian. This becomes far more apparent by considering the *BSL* expression.

Alternative 1 corresponds more to a top-down abstract model of *what* is to be achieved. Alternative 2 corresponds more to a bottom-up view of *how* the behaviour will be achieved.

Alternative 1 is considered to be the *most appropriate* binding strategy. This example shows one of the benefits of the *BSL* is allowing such design alternatives to be considered.

For completeness, the full binding with alternative 1 is as follows. A user, *u*, passes the partial parameters of the call, *pp*, to the required FADT operation, *op*, thus $(u)((pp)(op))$, and when the guardian allows this operation to proceed, the following binding holds:

$$(u)((pp)((op)((PE_i)((g);(sh))))),$$

which is sufficient for the operation to proceed *and* be serialisable. This larger binding only holds for the duration of an operation call. The corresponding design option is a set of such expressions for the various PE's and the *g* and *sh* in possible orders.

File Management

A traditional view of file management is that a user requests access to a named file, implicitly the file is to be accessed in some fashion. This access is obtained by associating a file manager with the file. To handle these requests the system recognises the request, determines its validity, identifies the physical file from

the name provided, and ensure that the file can be accessed in the way requested. Following this, a binding must occur between three resources: *user*, *file* and *file manager* to provide the required behaviour. The issue at this stage is how to bring about this binding.

Given a user u, a file f, and a file manager fm, (all are instances of a resource type as defined earlier), the following four bindings occurrences are possible, and are denoted using suitable sets of BSL expressions:

- All three resources are bound together at what appears to be the same instant in time; the design option is:

  $\{(u)\|(f)\|(fm)\}$.

- The user and file manager are bound together (in either order), and, at some subsequent instant, the file is bound. The binding of the user and the file manager is of the longer duration. This order may be required, for example, if the user wishes to access all files in the same manner, perhaps sequentially so as to print them all; the design option is:

  $\{(f)((u);(fm)),(f)((fm);(u))\}$

- The user and file are bound together (in either order), and, at some subsequent instant, the file manager is bound. The binding of the user and the file is of the longer duration. This order may be required, for example, if the user wishes to access the file in a number of different ways, perhaps sequentially for a report of the whole file, and then access it randomly for interactive reads and updates; the design option is:

  $\{(fm)((u);(f)),(fm)((f);(u))\}$

- The file and file manager are bound together (in either order), and at some subsequent instant the user is bound. The binding of the file and the file manager is for a longer duration. This order, essentially gives the user only one means of accessing the file, i.e. via its bound file manager; the design option is:

  $\{(u)((f);(fm)),(u)((fm);(f))\}$

Printer Access

Finally, consider a request from a user u, to print a file f on a printer. This request could be to print to a designated printer p_i, in which case the binding is to a specific instance of a dynamic resource type P (the set of all printers). Thus, $(p_i)((u)(f))$ However, if all printers have the same capability and are in the same location then the request would be to bind to the dynamic resource type P, and some printer scheduler can determine which printer instance should be used. Thus, $(P)((u)(f))$.

6 Conclusions and Future Work

The objective of this paper has been to place the informal, but common and useful, notion of *resource bindings* into a formal framework, to improve the high-level design of parallel operating systems. It has drawn on experiences of high-level design in two parallel operating systems projects.

The main benefits of the resulting design process (section 4), and from which the process derives its uniqueness, was use of the BSL to give a formal representation of the design space. This enabled a better analysis of the design alternatives and improved the design of, for example, Flagship ADTs where the advantage of binding the *guardian* and *stateholder* for the lifetime of a Flagship ADT thus allowing the processor to change during the lifetime was indicated when the BSL expressions were written down. Flagship ADTs were the building blocks of the Flagship kernel, and the BSL analysis presented in section 5 led to various re-designs of the prototype kernel.

The verification parts of the design process - stages 5 to 8 of section 4 - were based on temporal logic mainly because of early success with using the formalism in the EDS project [8]. They required considerably more input of effort and at best succeeded in providing fairly rigorous though informal arguments of the viability of certain designs. A lack of availability of appropriate tools was also a drawback of these stages. An alternative design process could replace these stages by use of other tools such as the Concurrency Workbench [5]. So, for example, stages 5 and 7 might mean providing CCS agents $\mathcal{CCS}_{op}$ and $\mathcal{CCS}_{constr}$, and stage 6 a specification $\mathcal{ML}_{prop}$ in the modal mu-calculus offered by the workbench. The verification at stage 8 would then involve running the workbench model checker to check that the composition of $\mathcal{CCS}_{op}$ and $\mathcal{CCS}_{constr}$ satisfied $\mathcal{ML}_{prop}$. It is likely that finiteness assumptions needed for provability in the temporal logic case would also be needed when using the workbench. However, it would nevertheless be worth comparing use of verification tools such as those offered by the workbench, in the context of appropriate stages 5 to 8 in the design process, with temporal logic tools used in the context of the design process given here.

Acknowledgements

Thanks to the Flagship and EDS groups, particularly Steve Leunig, Ken Mayes and Brian Warboys, who have contributed to our understanding of the design of parallel operating systems. Thanks also to the anonymous referees whose comments have improved the presentation of the work. This work was partly supported by UK Alvey project, IKBS 049 SERC grant GR/E 21070, and by ESPRIT II grant, EP2025.

References

[1] *Temporal-Logic Theorem Proving*, M. Abadi, PhD Thesis, Stanford University, Report No. STAN-CS-87-1151, 1987.

[2] MetateM: A Framework for Programming in Temporal Logic, H. Barringer, M. Fisher, D. Gabbay, G. Gough and R. Owens, In *REX Workshop on Stepwise Refinement of Distributed Systems*, LNCS-430, Springer-Verlag, 1989.

[3] The Use of VDM within the Alvey Flagship Project, G.S. Boddy, in *VDM - The Way Ahead*, LNCS-328, Springer-Verlag, 1988.

[4] Names and Name Resolution, D.E. Comer and L.L. Peterson, in *Concurrency Control and Reliability in Distributed Systems*, B.K. Bhargava (Ed.), van Nostrand Reinhold, 1987.

[5] The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems, R. Cleveland, J. Parrow and B. Steffen, *ACM TOPLAS 15(1)*, 1993.

[6] Specification of an Operating System Kernel: Forest & VDM compared, S.J Goldsack, in *VDM - The Way Ahead*, LNCS-328, Springer-Verlag 1988.

[7] *Specification Case Studies*, I. Hayes, Prentice-Hall International, 1987.

[8] Specification of a Distributed Operating System Environment, W. Hussak, EDS Project Document, Dept. of Computer Science, University of Manchester, 1989.

[9] The Use of Formal Methods in Parallel Operating Systems, J.A.Keane and W. Hussak, *Proc. of COMPSAC 92*, IEEE Press, 1992.

[10] Levels of Atomic Actions in the Flagship Parallel System, K.R. Mayes and J.A. Keane, *Concurrency: Practice and Experience 5(3)*, 1993.

[11] Specifying and Refining Concurrent Systems, R.E. Milne, RAISE/STC/REM/13/V2.STC, RAISE Project Document, 1990.

[12] Names, R.M. Needham, in *Distributed Systems*, S. Mullender (Ed.), ACM Press Addison-Wesley, 1989.

[13] Naming and Binding of Objects, J.H. Saltzer, in *Operating Systems: An Advanced Course*, LNCS-60, Springer-Verlag, 1978.

[14] On the Naming and Binding of Network Destinations, J.H. Saltzer, *Proc. Int. Symp. on Local Computer Networks*, IFIP/T.C.6, Florence, Italy, 1982.

[15] VME/B A Model for a Realisation of a Total System Concept, B.C. Warboys, *ICL Technical Journal 2 (2)*, 1980.

[16] VME Nodal Architecture: a Model for the Realisation of a Distributed System Concept, B.C. Warboys, *ICL Technical Journal 4 (3)*, 1985.