A Method of Verification in Design: an operating system case study

John A. Keane
Department of Computation
UMIST
Manchester, UK

Walter Hussak
Department of Computer Science
University of Loughborough
UK

Abstract

This paper reports a study of verification in the high-level design phase of operating system development in which both rigorous and formal verification are used, where the rigorous argument is used to determine a manageable formal proof to be carried out. A 2-sorted first order temporal language is used to express several possible high-level designs and the required properties of an operating system store manager. The case of large system limits is reduced to a case of small system limits by use of a rigorous argument. Corresponding propositional temporal logic (PTL) formulae are then verified using a PTL theorem prover.

Keywords

High-level design, Operating Systems, Rigorous Proof, Temporal Logic Theorem Provers.

1 Introduction

Software development is recognised as being highly complex and thus error-prone. The general aim of software engineering techniques is to ensure the production of correct and reliable software that meets requirements. The validation of these requirements is a major aspect of software engineering. There are many approaches to validation, one of which being the use of formal methods to provide some form of verification.

Formal methods have long been proposed as an antidote to some of the complexity of system design and validation. The proposed use of formality ranges from specification of requirements through to full-blown verification of systems with increasing cost, time and effort. For safety-critical systems there has been a recognition of the value, indeed necessity, of formality at various stage of development. Increasingly as computer systems become mission-critical for business, it is being recognised that `24 hours, 7 days` a week up-time is required. Such up-time mean impacts the underlying hardware and systems software. Generally for hardware, such availability is ensured by techniques such as duplication of component. In software a number of approaches are available at implementation level such as triple modular redundancy. Nevertheless such does not detract from the need for better methods of validation and verification.

In terms of operating system development of enterprise servers, it is becoming increasingly necessary to provide some formal treatment of validation and verification. Such systems play a critical role in the provision of service, and are highly complex: they often have real-time components, deal with an unpredictable load, handle security of information, tend to have very long installation lives (30+ years), and address a large and, at design-time, unknown set of application requirements. Perhaps most of all, they exhibit as an integral design rationale the presence of concurrency: many of the techniques and notations for reasoning about concurrency (semaphores, conditional critical regions, monitors, CSP etc) arose in the context of operating system design.

The authors' role in the development of two industrial-scale operating systems was to consider the most cost-effective and useful role of formal methods, given a small amount of human resource. In such a system, whilst it is potentially desirable to provide full-blown verification to the level of source code, to meet the cost of developing theorem provers or proof assistants that would be practical for use with the particular system is simply impossible due to resource constraints. The emphasis of the formal process therefore should be on the formal specification of requirements and the rigorous verification of design properties. A *rigorous* verification uses arguments that a mathematically educated person could understand and accept as constituting a valid proof. It need not be presented entirely symbolically with every minute

step of an argument justified by use of a computer, as is the case with a formal proof.

An issue arises as to how to validate high level design: one approach is to test the developed system to see if the system (and thus the high-level design to which the implemented system should correspond) has certain required properties. The approach here is to provide proofs for certain cases so that the high-level design can be evaluated *at* design time. Rather than argue whether the approach here constitutes testing or verification, we suggest that the common aim of both testing and the approach here is to validate that the developed system satisfies requirements.

This paper discusses an approach to formal specification and verification derived from work on these industrial-scale projects. In the context of the operating system design and development, the approach is cost-effective at providing rigorous analysis of design properties before embarking upon costly and difficult-to-change implementation. The method allows a first-order temporal language to be used abstractly at the specification stage, yet requires only a propositional temporal logic theorem prover to be used to verify properties of the system.

The paper is structured as follows: §2 describes the approach within the context of operating system design and development; in §3 store manager designs are specified in a 2-sorted first order temporal language; the reduction of these specifications to simple propositional temporal formulae is given in §4; in §5 the verification of the PTL properties using temporal logic theorem provers is discussed; Finally, further work and conclusions are discussed in §6.

2 A Practical Approach to Specification and Verification of Design

As with other software developers, operating system designers express and document requirements using natural language. It is important that when these high-level design requirements are expressed formally the correspondence between informal and formal is transparent to the designers.

A typical such specification involves processes and various shared resource types, as seen in [4] and [7]. The system is naturally specified in a first order temporal language representing a large number of processes and resource instances corresponding to the system limits. Following this specification activity, there is a need for verify properties of the system design. In view of the problem of using theorem provers at the actual system limits, the alternatives to full-blown formal verification appear to be:

A: (i) `specify for actual system limits`
   (ii) `verify directly by rigorous argument`

B: (i) `specify for actual system limits`
   (ii) `give temporal formulae for a small number of processes and resource instances`
   (iii) `formally verify formulae in B(ii)`

`C: perform both A and B`

The style of verification using a rigorous argument involves the use of much less detailed proof and provides much of the benefit of formal proof gained at a much lower cost. From this point of view the first approach might be advised.

However, there are two reasons why a formal element to the proof should be considered at the high-level design phase:

1. A greater assurance of correctness may be desired at the high-level design phase of system development because of the greater cost of incorrectness at this level.
2. The smaller amount of detail at the higher level actually makes it more practically viable for automated formal proof than lower levels of design.

If greater assurance is required then approach `C` may be considered, where a rigorous proof is accompanied by a formal proof of the temporal formulae resulting from giving small values to the number of processes and resource instances. The question is which small values should be given for the number of processes and resource instances if the formal proof is to *reinforce* the rigorous proof? The proof with the chosen values is useless if it does not reflect the case of large values.

As a consequence, the following approach, `D`, is proposed which combines the benefits of both rigorous and formal proof, and where the formal proof is a *logical extension* of the rigorous argument.

D: (i) `specify first order`
   (ii) `rigorous argument reduction to a small number of processes and resource instances`
   (iii) `give (propositional) temporal formulae for this small number of processes and resource instances`
   (iv) `formally verify formulae in D(iii)`

0-7695-0493-0/00 $10.00 (c) 2000 IEEE

A practical case study is given here to demonstrate how this approach can be used at the high-level design phase of operating system development. The context is the design methodology in [7] which analysed the informal design processes that took place in the high-level design of two industrially linked parallel operating systems projects - the UK Alvey Flagship project [6] and the Esprit EDS project [9]. As a result of this a design process to be put in place at the high-level design phase of system development was formulated. A feature of this design process is the use of a design space language BSL ('Binding Strategy Language'), which aims to provide some sort of guidance as to the design options. The later stages of the design process suggest formal verification of required properties of viable designs before the costly task of implementation is embarked upon. Precisely, the design process means formally specifying each *design* (the 'binding'), the *system* that it will run on (the 'architectural constraints') and the required *properties*, and then verifying that the overall *design+system* has the required *properties*.

The case study is of the design of a store manager. This will involve:

1. Specifying the design, system and properties for different store managers in a first-order temporal language;
2. Showing that validity for temporal formulae corresponding to actual system limits is equivalent to validity for valuations in a certain small case;
3. Verifying the formulae for the small case using a theorem prover.

It should be noted that the approach described does not help with finding required properties but rather verifying and validating them, and in a wider context achieving these properties in a developed system. The assumption is that the designers (by experience and consideration) have expressed the required properties - of course consistency of store is a major property for an operating system.

Related work on reducing large system limits to small values includes [2] and [8]. The contribution here is doing so in the context of the use of temporal logic in the high-level design of operating systems.

3 Formal Specification of Store Manager Designs

The first order temporal language used in this section, FLTL-2, is a 2-sorted language with time dependent predicates; for further language details the interested reader is directed to [5]. The language is easily understood by operating system designers in that FLTL-2 formulae correspond closely to the informal English description. At a high-level, store management comprises:

- user processes
- a system process
- a store map
- a physical store

The idea is that there is a shared virtual address space such that each user process accesses virtual store by first accessing the store map to find which physical address corresponds to a virtual address, and then the user process accesses the physical store.

The system process also accesses the store map and physical store in order to reorganise the store. This might be an access by the system process to change the store map, later followed by an access by the system to physical store, so that the new physical addresses of virtual addresses, receive the data that the old physical addresses of those virtual addresses contained.

The temporal specification of the store manager designs, the architectural constraints and the required properties involve specifying the sequence of events in time that each allows to occur. The events which are accesses to the store map and physical store are assumed to last the duration of a single point in time. Overall, the events that may or may not be occurring at a given point in time break down into statements about processes and the store map. These are treated as two sorts N and M respectively in the use of FLTL-2. The M-sorted predicate sm, the N-sorted predicates usm and uph, and the propositions ssm and sph have the following meaning:

- $usm(n)$: user process n is accessing the store map
- $uph(n)$: user process n is accessing physical store
- $sm(m)$: the m-th store map is in operation
- ssm: the system process is accessing the store map
- sph: the system process is accessing physical store

A pleasant feature of temporal logic is that specifications can be appreciated with only the following brief description of the temporal operators[1]:

[1] Equivalence of informal and formal specification cannot be proved, however, it is possible to consider subjectively how clear

0-7695-0493-0/00 $10.00 (c) 2000 IEEE

1. $\Box p$ (*always p*) is true if the formula p is true now and in every future moment;
2. $\bigcirc p$ (*at the next moment p*) is true now if the formula p is true at the next instant;
3. $p \mathcal{U} q$ (*p until q*) is true if q becomes true at some point in time and *until* that happens p will be true.

First of all, two different system architectures (given abstractly[2] in terms of 'architectural constraints'), that the store manager might run on, are specified.

3.1 System 1 Architecture

3.1.1 Store map properties

`At every point in time, there is some store map in operation`

- **constr1** $\stackrel{\text{def}}{=} \Box \exists m.sm(m)$

`Only one store map is in operation at any instant`

- **constr2** $\stackrel{\text{def}}{=} \forall m_1 \forall m_2. \Box (sm(m_1) \wedge sm(m_2) \Rightarrow m_1 = m_2)$

`A store map can only change when the system process accesses it`

- **constr3** $\stackrel{\text{def}}{=} \forall m_1 \forall m_2. \neg(m_1 = m_2) \Rightarrow \Box (sm(m_1) \wedge \bigcirc sm(m_2) \Rightarrow \bigcirc ssm)$

3.1.2 Concurrent Access

`A user cannot be accessing the store map and physical store at the same time`

- **constr4** $\stackrel{\text{def}}{=} \forall n. \Box \neg(usm(n) \wedge uph(n))$

`The system can only access the store map and physical store one at a time`

- **constr5** $\stackrel{\text{def}}{=} \Box \neg(ssm \wedge sph)$

`Only one user can access the store map at any given time`

- **constr6** $\stackrel{\text{def}}{=} \forall n_1 \forall n_2. \Box (usm(n_1) \wedge usm(n_2) \Rightarrow n_1 = n_2)$

`Only one user can access the physical store at any given time`

- **constr7** $\stackrel{\text{def}}{=} \forall n_1 \forall n_2. \Box (uph(n_1) \wedge uph(n_2) \Rightarrow n_1 = n_2)$

`The system cannot access the store map if a user is accessing the store map (and vice-versa)`

- **constr8** $\stackrel{\text{def}}{=} \forall n. \Box \neg(usm(n) \wedge ssm)$

`The system cannot access the store map whilst a user is accessing physical store`

- **constr9** $\stackrel{\text{def}}{=} \forall n. \Box \neg(uph(n) \wedge ssm)$

`The system cannot access physical store whilst the user is accessing physical store`

- **constr10** $\stackrel{\text{def}}{=} \forall n. \Box \neg(uph(n) \wedge sph)$

Notice that it is allowed for the system to access physical store when the user is accessing the store map. This might occur if the system is completing its latest reorganisation of store by performing the necessary physical store updates corresponding to changes it made to the virtual to real address store map earlier on. It is safe for a user to access the new store map at this point as a subsequent access by the user to physical store will occur after the system has finished with the physical store and will pick up the corresponding new state of the physical store.

3.2 System 2 Architecture

System 2 has all the architectural constraints **constr1** to **constr10** and has the additional constraint that,
`after a change to the store map by the system, a user cannot access physical store until that user has observed (accessed) this store map`

- **constr11** $\stackrel{\text{def}}{=} \Box (ssm \Rightarrow \forall n.(\neg uph(n) \mathcal{U} usm(n)))$

3.3 First Design Option

A design option comprises a specification of the initial conditions and the 'binding' in the terminology of [7]. The initial conditions for the first design are that the first two users access the store map without a need for an intervention by the system

is the correspondence between informal and formal. To assess this, a formalism needs to be tried: here the correspondence of the formal (temporal logic) to the informal (English) "looks good".

[2] An architecture of a system can be defined abstractly in terms of the properties that the system should exhibit. In this context the system is the store manager.

0-7695-0493-0/00 $10.00 (c) 2000 IEEE

- **init1** $\stackrel{\text{def}}{=}$ $\exists n_1 \exists n_2.\ \neg(n_1 = n_2) \wedge usm(n_1) \wedge \bigcirc usm(n_2)$

The 'binding' condition states that if the user accesses the store map, the next access by the user will be to physical store at some later point in time

- **bind**$_{((U)(sm))(ph)}$[3] $\stackrel{\text{def}}{=}$ $\forall n.\ \Box\ (usm(n) \Rightarrow \bigcirc(\neg usm(n) \mathcal{U} uph(n)))$

3.4 Second Design Option

In the second design, the initial conditions are that first a user accesses the store map, and then the second user accesses the store map at some later point in time (not necessarily immediately)

- **init2** $\stackrel{\text{def}}{=}$ $\exists n_1 \exists n_2.\ \neg(n_1 = n_2) \wedge usm(n_1) \wedge \Diamond usm(n_2)$

For the binding, the user accesses physical store as soon as (at the very next instant in time) the physical address of the virtual address is obtained

- **bind**$_{(U)((sm)(ph))}$[4] $\stackrel{\text{def}}{=}$ $\forall n.\ \Box\ (usm(n) \Rightarrow \bigcirc uph(n))$

3.5 Required Properties

The required (safety) property of the overall system is that the store is kept consistent. In other words, if a user accesses a store map, then the next access to physical store cannot occur when a different store map is in operation if the user has not accessed the store map in the meantime. This is expressed by the following equation

- **prop** $\stackrel{\text{def}}{=}$ $\forall n \forall m.\ \Box\ (usm(n) \wedge sm(m) \Rightarrow \bigcirc\neg((\neg usm(n) \wedge \neg uph(n)) \mathcal{U} (uph(n) \wedge \neg sm(m))))$

3.6 Proof Obligations

The three proof obligations that will be considered are as follows. Firstly, to see if the first design running on System 1 satisfies the required properties. For this, it is necessary to show that

- **eq1** $\stackrel{\text{def}}{=}$ $(init1 \wedge bind_{((U)(sm))(ph)} \wedge \bigwedge_{i=1}^{10} constr_i) \Rightarrow prop$

is valid. Secondly, to determine whether the second design running on System 1 is correct the validity of

- **eq2** $\stackrel{\text{def}}{=}$ $(init2 \wedge bind_{(U)((sm)(ph))} \wedge \bigwedge_{i=1}^{10} constr_i) \Rightarrow prop$

needs to be demonstrated. Finally, to see if the first design running on System 2 is correct, the validity of

- **eq3** $\stackrel{\text{def}}{=}$ $(init1 \wedge bind_{((U)(sm))(ph)} \wedge \bigwedge_{i=1}^{11} constr_i) \Rightarrow prop$

needs to be shown.

4 Rigirous Argument

The next stage of the process involves:

1. producing a rigorous argument that reduces the specification to one with a small number of processes and resource instances;
2. providing the (propositional) temporal logic formulae corresponding to these small number of processes and resource instances.

4.1 Reduction to Small System Limits

For our case study, this involves showing that that the proof obligations of §3- eq1, eq2 and eq3 - are each valid for the actual system limits if and only if they are valid for the case of 2 store maps and 2 processes. Precisely, this means proving (rigorously):

(i) $\models_{(i_M, i_N)}\ eq1 \Leftrightarrow \models_{(2,2)}\ eq1$

(ii) $\models_{(i_M, i_N)}\ eq2 \Leftrightarrow \models_{(2,2)}\ eq2$

(iii) $\models_{(i_M, i_N)}\ eq3 \Leftrightarrow \models_{(2,2)}\ eq3$

where $\models_{(\alpha,\beta)}\ eq$ denotes validity of the FLTL-2 formula eq interpreted over domains of cardinality α and β for the two sorts M and N respectively. The integers i_M and i_N are the actual system limits in our case study. The rigorous argument for (i), (ii) and (iii) is too large to include here but represents 3-4 pages of hand-proof. The interested reader is referred to [5] for the full details.

4.2 Propositional Formulae for Small System Limits

This involves listing the propositional temporal formulae corresponding to the formulae in §3 for the case of 2 users and 2 store maps.

[3] The subscript uses the notation of BSL, a language for generating design options. By a design option we mean a plausible design of a system. The role of a design option is to direct the designer at the next level of design [7].

[4] Notice that, here, the subscript associates to the right, whereas in the first design option the subscript associated to the left.

0-7695-0493-0/00 $10.00 (c) 2000 IEEE

4.2.1 Propositional Variables

- $sm1$, $sm2$, $usm1$, $usm2$, $uph1$, $uph2$, ssm, sph

4.2.2 System 1 and System 2 Constraints

- **constr1** $\stackrel{\text{def}}{=} \Box(sm1 \vee sm2)$
- **constr2** $\stackrel{\text{def}}{=} \Box\neg(sm1 \wedge sm2)$
- **constr3** $\stackrel{\text{def}}{=} \Box(((sm1 \wedge \bigcirc sm2) \Rightarrow \bigcirc ssm) \wedge ((sm2 \wedge \bigcirc sm1) \Rightarrow \bigcirc ssm))$
- **constr4** $\stackrel{\text{def}}{=} \Box(\neg(usm1 \wedge uph1) \wedge \neg(usm2 \wedge uph2))$
- **constr5** $\stackrel{\text{def}}{=} \Box\neg(ssm \wedge sph)$
- **constr6** $\stackrel{\text{def}}{=} \Box\neg(usm1 \wedge usm2)$
- **constr7** $\stackrel{\text{def}}{=} \Box\neg(uph1 \wedge uph2)$
- **constr8** $\stackrel{\text{def}}{=} \Box(\neg(usm1 \wedge ssm) \wedge \neg(usm2 \wedge ssm))$
- **constr9** $\stackrel{\text{def}}{=} \Box(\neg(uph1 \wedge ssm) \wedge \neg(uph2 \wedge ssm))$
- **constr10** $\stackrel{\text{def}}{=} \Box(\neg(uph1 \wedge sph) \wedge \neg(uph2 \wedge sph))$

4.2.3 System 2 Additional Constraint

- **constr11** $\stackrel{\text{def}}{=} \Box((ssm \Rightarrow (\neg uph1 \mathcal{U} usm1)) \wedge (ssm \Rightarrow (\neg uph2 \mathcal{U} usm2)))$

4.2.4 First Design Option

- **init1** $\stackrel{\text{def}}{=}$

$$(usm1 \wedge \bigcirc usm2)$$

$$\vee$$

$$(usm2 \wedge \bigcirc usm1)$$

- **bind**$_{((U)(sm))(ph)}$ $\stackrel{\text{def}}{=}$

$$\Box((usm1 \Rightarrow \bigcirc(\neg usm1 \mathcal{U} uph1))$$

$$\wedge$$

$$(usm2 \Rightarrow \bigcirc(\neg usm2 \mathcal{U} uph2)))$$

4.2.5 Second Design Option

- **init2** $\stackrel{\text{def}}{=}$

$$(usm1 \wedge \Diamond usm2)$$

$$\vee$$

$$(usm2 \wedge \Diamond usm1)$$

- **bind**$_{(U)((sm)(ph))}$ $\stackrel{\text{def}}{=} \Box((usm1 \Rightarrow \bigcirc uph1) \wedge (usm2 \Rightarrow \bigcirc uph2))$

4.2.6 Required Properties

- **prop** $\stackrel{\text{def}}{=}$

$$\Box(\neg(usm1 \wedge sm1 \wedge \bigcirc((\neg usm1 \wedge \neg uph1) \mathcal{U} (uph1 \wedge \neg sm1)))$$

$$\wedge$$

$$\neg(usm1 \wedge sm2 \wedge \bigcirc((\neg usm1 \wedge \neg uph1) \mathcal{U} (uph1 \wedge \neg sm2)))$$

$$\wedge$$

$$\neg(usm2 \wedge sm1 \wedge \bigcirc((\neg usm2 \wedge \neg uph2) \mathcal{U} (uph2 \wedge \neg sm1)))$$

$$\wedge$$

$$\neg(usm2 \wedge sm2 \wedge \bigcirc((\neg usm2 \wedge \neg uph2) \mathcal{U} (uph2 \wedge \neg sm2))))$$

5 Formal Verification of the Designs

At this stage suitable PTL formulae have been established, based on the rigorous argument. Hence this stage is concerned with a formal verification of the PTL formulae.

The three equations **eq1**, **eq2**, and **eq3** were run on the `dp` temporal theorem prover [3]. `dp` is a decision procedure for a future linear time temporal logic over infinite time models. In the way it is used here `dp` attempts to show that a given formula is valid, i.e. that its negation has no model. Readers interested in the detailed output of `dp` are referred to [5], here a part of the verification is given.

Using `dp` involves having to prepare the temporal logic formulae in an input file according to the `dp` syntax as follows:

- `G` corresponds to *always*, i.e. $\Box$,
- `U` corresponds to *until*, i.e. $\mathcal{U}$,
- `F` corresponds to *sometimes*, i.e. $\Diamond$,
- `X` corresponds to *next*, i.e. $\bigcirc$,
- `&` corresponds to *and*, i.e. $\wedge$,

0-7695-0493-0/00 $10.00 (c) 2000 IEEE

- | corresponds to *or*, i.e. $\vee$,
- ~ corresponds to *not*, i.e. $\neg$,
- > corresponds to *implies*, i.e $\Rightarrow$.

For example, the formula **constr8**

$$\Box(\neg(usm1 \wedge ssm) \wedge \neg(usm2 \wedge ssm))$$

is translated to be

```
G(~(usm1 & usm) & ~(usm2 & ssm))
```

Given this translation, **dp** informs the user as to the validity of the entered formula.

The output from `dp` has been laid out slightly differently for appearance purposes. In addition, `dp` inserts left associating brackets. Thus

```
(((A&B)&C)&D)
```

is displayed instead of

```
A&B&C&D
```

Apart from this, the correspondence between formulae in `dp` and those given in §4 should be obvious.

5.1 First Design/System 1

eq1, i.e. $(init1 \wedge bind_{((U)(sm))(ph)} \wedge \bigwedge_{i=1}^{10} constr_i) \Rightarrow prop$, was shown to be invalid. Recall this is the design that states that if the user accesses the store map, the next access by the user will be to the physical store at some later point in time. System 1 is defined as the set of constraints, [**const1**,...,**constr10**]. The interaction with the theorem prover was as follows:

```
dp> ((((((((((
((usm1 & Xusm2) | (usm2 & Xusm1))
& G((usm1 > X(~usm1 U uph1))
    & (usm2 > X(~usm2 U uph2))))
& G(sm1 | sm2))
& G~(sm1 & sm2))
& G(((sm1 & Xsm2) > Xssm)
    & ((sm2 & Xsm1) > Xssm)))
& G(~(usm1 & uph1) & ~(usm2 & uph2)))
& G~(ssm & sph))
& G~(usm1 & usm2))
& G~(uph1 & uph2))
& G(~(usm1 & ssm) & ~(usm2 & ssm)))
& G(~(uph1 & ssm) & ~(uph2 & ssm)))
& G(~(uph1 & sph) & ~(uph2 & sph)))

G(((((usm1 & sm1) > X~((~usm1 & ~uph1)
    U (uph1 & ~sm1)))
& ((usm1 & sm2) > X~((~usm1 & ~uph1)
    U (uph1 & ~sm2))))
& ((usm2 & sm1) > X~((~usm2 & ~uph2)
    U (uph2 & ~sm1))))
& ((usm2 & sm2) > X~((~usm2 & ~uph2)
     U (uph2 & ~sm2))))
Invalid
(142.4s)
```

5.2 Second Design/System 1

eq2, i.e. $(init2 \wedge bind_{(U)((sm)(ph))} \wedge \bigwedge_{i=1}^{10} constr_i) \Rightarrow prop$, was shown to be valid. Recall this is the design that states that the user accesses physical store as soon as (at the very next instant in time) the physical address of the virtual address is obtained.

5.3 First Design/System 2

eq3, i.e. $(init1 \wedge bind_{((U)(sm))(ph)} \wedge \bigwedge_{i=1}^{11} constr_i) \Rightarrow prop$, was also shown to be valid. Recall this is the same design as **eq1** specifies, except there is an additional architectural constraint, **constr11**, which serves to make the design valid.

5.4 Summary

It has thus been shown that the *first* design option is invalid with System 1 but is valid with System 2, where System 2 is equivalent to System 1 with the addition of **constr11**. The *second* design option has been shown to be valid for System 1.

6 Conclusions and Further Work

Within the context of industrial-scale operating system development, the use of a formal treatment is increasingly considered essential. The question is more where can the necessarily limited resource for formal approaches be used for most benefit?

Here a method has been presented for verifying requirements expressed in temporal logic by using both a rigorous and a formal proof. The rigorous argument has been used to determine the appropriate formal proof to be discharged. The temporal logic used FLTL-2, although having a first-order language, has an intended interpretation over the actual operating system limits which are large but nevertheless finite. This makes FLTL-2 decidable even for the *actual* system limits and means that *in theory* designs could be

0-7695-0493-0/00 $10.00 (c) 2000 IEEE

proved correct without recourse to any rigorous argument. *In practice* even the most powerful temporal logic theorem provers have a far more modest capability and would be of little use in producing such a proof. The method here enables use to be made of such theorem provers to benefit the correctness of the design. Indeed, the machine proofs of the case study in this paper were carried out on a small in-house theorem prover. The approach appears both cost-effective and thorough. Tool support is envisaged for the production of the PTL formulae based on the FLTL-2 formula and the numbers of processes and resources derived in the rigorous argument. Similarly, tool support to translate the form of PTL used here to that used as input of a particular theorem prover is also envisaged.

The case study example, whist small enough to be described here, is complex enough to involve concurrent and different levels of users. Other case studies with this method are being considered, for example a weakened requirements **prop** has been suggested, and distributed store management is under consideration. Following these, it is hoped that heuristics for the type of rigorous argument required for concurrent system development may be derived.

Acknowledgements

Thanks to all ex-colleagues in the Flagship and EDS projects, and to the anonymous referees and the attendees of ICSE-16 who encouraged us to carry out verifications and thus motivated much of this work.

References

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

[2] E.M.Clarke, O.Grumberg and D.E.Long, Model Checking and Abstraction, *ACM TOPLAS* 16:5, 1994.

[3] G.D. Gough, Decision Procedures for Temporal Logic, MSc. Dissertation, Department of Computer Science, University of Manchester, UK, Technical Report Series UMCS-89-10-1, 1984.

[4] W. Hussak, Temporal Analysis of a Microkernel, *Software Engineering Journal* 10:1, 1995.

[5] W. Hussak and J.A. Keane, Cost-effective Specification and verification of an Operating Systems case study, Technical Report, department of Computation, UMIST, 1998.

[6] J.A. Keane, An Overview of the Flagship System, *Journal of Functional Programming* 4:1, 1994.

[7] J.A. Keane and W. Hussak, A Formal Approach to Determining Parallel Resource Bindings, In *Proc. of 16th International Conference on Software Engineering*, IEEE Press, 1994.

[8] F. Pong and M. Dubois, A New Approach for the Verification of Cache Coherence Protocols, *IEEE Transactions on Parallel and Distributed Systems* 6:8, 1995.

[9] C.J. Skelton, C. Hammer, M. Lopez *et al.*, EDS: A Parallel Computer System for Advanced Information Processing, In *PARLE'92*, D. Etiemble and J.-C. Syre (Eds.) LNCS-605, Springer-Verlag, 1992,

0-7695-0493-0/00 $10.00 (c) 2000 IEEE