The use of formal methods in parallel operating systems


Additional Information:

- This is a conference paper [© IEEE]. It is also available from: http://ieeexplore.ieee.org/xpl/RecentCon.jsp?punumber=438. Personal use of this material is permitted. However, permission to reprint / republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Metadata Record: https://dspace.lboro.ac.uk/2134/4152

Version: Published

Publisher: © IEEE

Please cite the published version.
This item was submitted to Loughborough’s Institutional Repository (https://dspace.lboro.ac.uk/) by the author and is made available under the following Creative Commons Licence conditions.

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/
The Use of Formal Methods in Parallel Operating Systems

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

Work on parallel operating systems has been in progress at the University of Manchester for the past 5 years. In this time the operating systems for two experimental parallel machines have been developed. A common specification approach has evolved as part of the development of these two very different systems.

In this paper a description of how both a technique for formally specifying sequential systems (VDM) and a technique for specifying concurrent systems (Temporal Logic) have been used compatibly. In both cases the issue of verification is addressed.

1 Introduction

This paper reports on the use of formal methods in the development of parallel operating systems for two experimental declarative systems over a 5 year period. A common specification approach has evolved as part of the development of these two very different systems: one being for a parallel graph reduction machine and written in a functional language enhanced with state-based objects, the other is a distributed UNIX lightweight process model and is written in C++. Given that these were parallel operating systems projects and collaborative with industry, the actual use of formal methods in an exhaustive way was not feasible. Consequently it has been necessary to apply formal methods in what has been considered the “most beneficial” way.

In the approach described here, operations that affect different parts of state have been developed top-down using VDM [6]. These are decomposed into finer-grained operations as part of the reification process. Ultimately, operations are a sequence of atomic low-level operations. It is at this stage that sequencing concerns appear. High-level concurrent requirements are expressed as required properties, in temporal logic, and the temporal description of the implementation is verified against this.

In section 2 the operating system for the Flagship parallel graph reduction system is discussed, and in section 3 the lightweight process model for the EDS parallel declarative system is considered. A brief overview of each system is given before concentrating on the use of formal methods.

2 The Flagship Operating System

The Flagship project aimed to build a complete computing system based on a functional programming style, using a packet-based fine-grained graph reduction model of computation. The machine is made up of a set of processor-memory pairs (PEs). The distributed store is globally addressable.

The kernel of the system software was decomposed into a structure of components designed to work concurrently with each other [8]. A process at the system software level corresponds to a functional program, more fine-grained execution is provided by the reduction of the graph of packets representing the program. The system software was implemented above the graph reduction computational model. As a result a functional language (with the possibilities of efficient parallel execution) was used for implementation. However, the necessity for some form of mutable state and non-determinism for the system software meant that the pure functional language Hope+ [10] was extended with state-based object-like structures, termed Flagship ADTs (FADTs).

FADTs were the unit of abstraction, design and decomposition for the system software. The subsystems of the kernel were composed of these FADTs. The most important characteristic of an FADT is encapsulated state. Each instance of an FADT can have its own state. The state is encapsulated in that access to it, from outside the FADT, can only occur via the interfaces provided by the FADT. State is introduced as part of the basic building block. Its encapsulation
is based on the principle of information hiding. In addition, a mechanism ensures the consistency of the state of an FADT, thus each operation that updates the state of an FADT is an atomic action. Each FADT with state resides on a particular PE.

Flagship VDM (FVDM) [2], a hybrid version of VDM, was developed to specify FADTs. FVDM includes properties of Hope+, such as polymorphic data types. A specification of an FADT in FVDM consists of type definitions together with functions and operations (which specify state changes) on those types.

An Example Kernel Subsystem

To indicate the use of FVDM parts of an example subsystem will be specified. It is not intended to describe the subsystem in detail.

The subsystem of interest managed the Static Copying Environment Table (SCET) [7]. The SCET mechanism enables locality of reference within functional programs to be exploited on Flagship. Briefly the purpose of this mechanism is to ensure that whenever a copy of a function definition or constant data is held in one of the PEs, it is always located in a well-known place. A SCET is conceptually a tuple (Master SCET, set (Local SCET)), where each master SCET and its associated local SCETs reside on separate PEs. Each Master and Local SCET consist of (SCET index, SCET entry) tuples, where the SCET entry is conceptually another tuple (global address, local address). This mechanism essentially provides a local copy of a "packet" at local address, if no local copy exists then one is made from the global address and stored at the local address (recall the store is globally addressable).

The subsystem is made up of three FADTs:
- Global SCET Manager: which acts as the interface to SCET Manager from other subsystems. This is partly specified below.
- Global-to-Local SCET Manager: which directs calls to the appropriate PE where the call is to be serviced. This is partly specified below.
- Hardware SCET Manager: the system software model regards Hardware SCET Manager as an FADT with state and operations. On the actual machine this state and operations existed as part of the basic execution mechanism.

GLOBAL SCET MANAGER FADT:

props STATE-BASED_FADT;
pubtype GLSCETM_MANAGER;
pubfun Init_GLSCETM;
pubop Execute_On_PE;
data HW_SECT_Stf_Map_Type;
state GLSCETM_MANAGER == REC ((
  HW_STF_MAP, !STF=state transition function!
  HW_SECT_STF_Map_Type));

dec Init_GLSCETM: GLSCETM_MANAGER;
 --- Init_GLSCETM <= mk_GLSCETM_MANAGER
          (map_abstract (}
{0, 1, 2, 3} ! pe set ! 
(lambda pe_id =>
HW_SCET_MNG_MANAGER (pe_id)
end)); 25.

OP Execute_On_PE: pe_id: PE-id
ext rd GLSCETM: GLSCETH-MANAGER; 27.
post let this_stf_set == map-lookup
   (HW_STF_MAP (GLSCETM), pe_id) in
   let func == f(this_stf_set) in
   let result == func (arg)
in (result, GLSCETH); 28.

This FADT receives requests to the SCET Manager
subsystem (via the Global SCET Manager) and directs
them on to the appropriate Hardware SCET Manager.
Only points new facilities of FVDM will be stressed.
* Line 16 indicates that this is a state-based FADT.
* Line 22-23 defines the state.
* Line 24-25 define the function Init-GLSCETM,
which creates an instance of this FADT (and its
associated state) on each PE (0, 1, 2, 3).
* Line 26-30 define the operation Execute-On-PE
which is higher-order in the sense that it takes a func-
tion name, f and its arguments and a designated PE,
and calls the instance of f associated with the FADT
that resides on the designated PE. Execute-On-PE is
specified in terms of its pre (line 28) and post (lines
29-30) conditions. Line 27 lists the state that it reads.

Verification of Individual Operations

All of the Flagship Kernel operations were specified
in FVDM. Explicit functions are expressed as normal
Hope* functions. Operations that manipulate state
were expressed in terms of pre and post conditions.

These FVDM specifications could then be trans-
formed into Hope* executable models with minimum
effort and the model could then be tested. For ex-
ample, the pre and post conditions for an operation f
generate truth-valued functions pre-f and post-f, which
are tested before and after the call to f. In a simi-
lar way data type invariants [6] could be tested. This
form of executable model, allied to testing and fur-
ther Fagan inspections, of the specifications, provided
a limited form of verification of the required sequential
behaviour of the individual Kernel operations.

Verification of Concurrent Behaviour

An informal definition of the Flagship concurrent
Kernel was given in terms of an implicit parallel com-
binator ||, as in [5], in an understood language for the
FADT operations (Kernel operations). The FVDM
specifications 'work' in the sense of [5] with suitable
rely and guarantee conditions. The conditions are
assumed to be:
* I rely on no other operation updating the piece of
state that I am undating.
* I guarantee to only update the state indicated in my
post-condition.

Thus, for OP1 and OP2 in the FADT system,

\[ OP1 || OP2 \]

as is [5]. In [5] || is used when decomposing a higher
level operation OP into

\[ OP = OP1 || OP2 \]

Here, there is no higher level operation - OP1 || OP2
with the disjunction of their post-conditions constitute
OP. The only proof obligation to be discharged is that
of co-existence, in other words

\[ \text{rly}_{T_i}(\sigma, \sigma') \Rightarrow \text{guar}_{T_i}(\sigma, \sigma') \]

Clearly, this is the case as

\[ \text{guar}_{OP1}(\sigma, \sigma') = \sigma_1 = \sigma'_1 \Rightarrow \text{rly}_{OP2}(\sigma_2, \sigma'_2) = \sigma_2 = \sigma'_2 \]

and

\[ \text{guar}_{OP2}(\sigma, \sigma') = \sigma_1 = \sigma'_1 \Rightarrow \text{rly}_{OP1}(\sigma, \sigma') = \sigma_1 = \sigma'_1 \]

Therefore, the existing FADTs could be composed in
parallel as in [5], and this provides a semantics. All
the above is the rigorous argument contained in the
informal definition of the concurrent kernel implied in
[8]: given a Flagship concurrent FADT system, the
behaioir of each FADT is to a certain extent, equivalent
to the behaviour of the FADTs in isolation.

3 EDS: Lightweight Process Model

The European Declarative System (EDS) is inten-
tended to support various declarative languages. The
machine architecture is made up of a set of nodes, each
node contains 2 processors and a store unit. The operating system is intended to be UNIX-like with a 4-level process model. To experiment with this model a light-weight micro-kernel was layered on top of standard UNIX processes. This micro-kernel enabled light-weight micro-processes to be scheduled, thus providing fine-grained non-deterministic multi-programming [3]. Essentially, at an abstract level the system comprises processes being serviced by the processor and being switched on expiry of their allotted timeslice. This is caused by a timer signal. However, a problem arises:

"... when the timer causes a signal to occur whilst a process is in kernel mode. The switcher (scheduler) must not schedule another process since to do so might lead to the corruption of kernel data structures, but on the other hand to give the currently executing process another timeslice would be unfair to other processes which are ready to run. Indeed if this solution was adopted then a process could continue indefinitely by always being in kernel when the timer signal is received. A compromise solution is to allow a process to continue after its timeslice ends if it is in kernel mode when this happens, but to force a context switch when kernel mode is left ... " [3].

It is necessary to place all the aspects of the system in a formal setting, in order to translate this into a formal requirement. In common with many methods for specifying concurrency including [1], the first step is to identify the observable events.

**Observable Events**

The system comprises the components: processes, a processor, a scheduler and a timer signal.

Concurrent systems are to be specified as sets of sequences of observable events that are allowed to occur over the passage of time. The observable events are represented, formally, by propositions which are to be true at a given point in time if the event can be deemed to be occurring at that point in time.

- \( p(n): \) process \( n \) is active
- \( k(n): \) process \( n \) is active and in kernel mode
- \( \text{start}(n): \) process \( n \) performing start of kernel work
- \( \text{end}(n): \) process \( n \) on the point of completing kernel work
- \( t(n): \) process \( n \) is about to terminate
- \( \text{sg}: \) a timer signal is occurring
- \( \text{sw}: \) a new process will be scheduled at the next instant

For the above informal requirements to be presented formally a set of sequences of these events need to be specified. As is shown below, this is expressed very naturally in temporal logic. Precisely, the allowable set of sequences of events are given as the set of solutions to a temporal logic formula over \( \omega \)-sequences.

The formal semantics of the temporal logic used are given in [1], but the power of the intuitive appeal of the logic means that the specification can largely be understood with only a brief description of the operators used. The specification below make use of three temporal operators (\( \phi \) and \( \psi \) are temporal formulae):

1. \( \Diamond \phi \) ("always \( \phi \)) is true if the formula \( \phi \) is true now and in every future moment.
2. \( \Diamond \phi \) ("at the next moment \( \phi \)) is true now if the formula \( \phi \) is true at the next instant.
3. \( \phi \land \psi \) ("\( \phi \) until \( \psi \)) is true if \( \phi \) becomes true at some point in time and until that happens \( \phi \) will be true.

**Formal Requirements**

The following are the constraints on how the processes may be scheduled. From the informal requirements, a process is to continue after its timeslice ends if it is in kernel mode, but to force a context switch when kernel mode is left. From this it is clear that three situations are allowed to arise:

1. The process terminates before its timeslice expires.
2. The process is switched at the end of its timeslice as it is not in kernel mode.
3. The process is in kernel mode when its timeslice expires, so it continues in an active state until it has completed that block of kernel work and at the end of that a process switch occurs.

This behaviour is expressed formally by describing what may happen between consecutive process switches. In other words, if a switch occurs at some point in time, what can happen up to the next process switch? The appropriate condition for the switching constraints is thus of the form:

\[
\text{switching constr} \defeq \Box (\text{sw} \Rightarrow \phi_1 \lor \phi_2 \lor \phi_3),
\]

where \( \phi_1, \phi_2 \) and \( \phi_3 \) correspond to the three situations given above:

\[
\phi_1 \defeq \bigvee_n \Box ((\neg \text{sw} \land \neg \text{sg} \land p(n)) \cup (t(n) \land \text{sw}))
\]

states that after a process switch there is no process switch, no timer signal and the process remains active until it terminates at which point a process switch does occur;

\[
\phi_2 \defeq \bigvee_n \Box ((\neg \text{sw} \land \neg \text{sg} \land p(n)) \cup (\text{sg} \land \text{sw} \land p(n) \land (\neg \text{k}(n) \lor \text{end}(n))))
\]

is the normal situation where after a process switch there is no process switch, no timer signal and the process is active until a timer signal does occur at which point there is a process switch as the active
process was either not in kernel or at the point of exit from kernel mode:

\[ \phi_3 \triangleq \bigvee_n \left( \bigwedge \left( -sw \land -sg \land p(n) \right) \right) U \left( \bigwedge \left( sg \land -sw \land k(n) \land -end(n) \right) \right) \bigwedge \bigvee_n \left( \bigwedge \left( sw \land end \land k(n) \right) \right) \]

is when after a process switch there is a period of no switching and no timer signal, until a timer signal occurs when the active process is in kernel mode and so it continues, without being interrupted by either a timer signal or process switch and not starting new kernel work, up to the point at which it completes its portion of kernel work and finally a process switch occurs. Notice how closely the verbal description follows the temporal logic.

This forms the basis of the formal requirements formal_req given in full in [4].

**Actual Behaviour of the Implementation**

The required process scheduling is implemented by use of flags which are set at various times, and on the basis of which a process switch is invoked either by a scheduler or a process. Thus, the implementation level is more detailed and has observable events denoted by ps, ss, c, setc, psetc, sc, ssetc, psetxs, ssetxs.

There are two propositions ps and ss, which indicate whether a process switch is initiated by a process or by the scheduler as the result of a timer signal. The remaining propositions relate to flags that control the scheduling of processes. There are two flags - the critical and switch_on_exit flags (see [3]). The propositions c and sz respectively, indicate the truth value of these flags at any given moment in time. In order to change their value the flags need to be set. This calls for propositions setc and ssetc to indicate when a flag is being set. As with process switching, flags can be set either by a process or by the scheduler. Thus there are the propositions psetc, sc, psetxs and ssetxs.

The temporal behaviour of the processes in the implemented system is extracted from the code as shown below. One of nine statements of interest, with respect to process switching and flag setting, can be true for a process at any instant:

\[
\begin{align*}
\text{action}_1(n) & \triangleq p(n) \land \neg k(n) \land \neg \text{start}(n) \land \neg \text{end}(n) \\
\text{action}_2(n) & \triangleq p(n) \land k(n) \land \neg \text{start}(n) \land \neg \text{end}(n) \\
\text{action}_3(n) & \triangleq p(n) \land k(n) \land \text{start}(n) \land \neg \text{end}(n) \\
\text{action}_4(n) & \triangleq p(n) \land k(n) \land \text{start}(n) \land \text{end}(n) \\
\end{align*}
\]

These reflect whether a process is inside or outside a critical section (kernel), and, if inside, whether it is at the start, in the middle or at the end of it. Whether process switching is about to take place and the flag setting is given, in each case. Consider the structure of the code each process executes (taken from [3]):

```c
void P(n)
{...
7. /* Semaphore requested */
8. s->sem_value.sem_count -= 1;
9. _critical = FALSE;
10. ...
13. switch_on_exit = FALSE;
14. if (_save_regs(&pcurr->context)) _do_switch();
15. ...
}
```

Line 4 corresponds to action$_2$, line 5 corresponds to action$_3$, lines 6-9 correspond to action$_4$, line 10 corresponds to action$_5$, line 11 to action$_1$, line 12 to action$_6$, line 13 to action$_8$, and line 15 to action$_9$. The behaviour of all the processes together interleaves the actions of the individual processes, so that each process "performs" its own actions in the order that the displayed code dictates. The temporal logic formula in terms of \( \square \), \( \Box \) and \( U \) is rather messy. An abbreviated notation using an intuitive notation period and point is used in [4]. It is intended to indicate whether, conceptually, something occurs for a period or for the duration of one instantaneous point.
period (actions(n) v action1(n))
point (actions(n))
period (actions(n) v action2(n))
point (actions(n))
period (actions(n))
point (sz ∧ action1(n))
period (actions(n))
point (actions(n))
period (actions(n))
point (actions(n))
period (actions(n))
point (actions(n))
period (actions(n))
point (actions(n))
period (actions(n))
point (actions(n))
period (actions(n))

A (A, A, 0 ∧ action1(m)))

The large disjunction results from the testing of the switch_on_exit flag in line 11 of the code above and the execution of additional code if the value is true.

Other features of process behaviour are given in [4]; they are denoted by processes here. The overall temporal behaviour of the implementation is:

\[ \text{implementation} \triangleq \text{flag-conds} \land \text{interleaving} \land \text{processes}. \]

Proof Obligation

The proof obligation is simply:

\[ \text{implementation} \Rightarrow \text{formal req.} \]

In the original specification a hand proof was carried out. A complete specification is contained in [4]. The specification reduces to a specification in ordinary propositional temporal logic with the not unrealistic assumption that there is a (known) finite limit to the number of processes. This means that the proof of the temporal properties can be mechanised, indeed the area of mechanising proofs in propositional temporal logic is currently an active area of research. In particular, it demonstrates that the specification techniques of [1] can be used in the production of real systems.

4 Conclusions

Various approaches such as [9] have tried to reconcile relational methods of specification such as VDM and trace-based methods such as temporal logic. However, the problems are great and a theoretical solution is not feasible as an objective for a project producing a real system. The pragmatic solution that has been adopted on these projects, was to only introduce formality where informality was present, whether it be written in English or in the mind of the designer. This paper has described the systems that the projects have developed and illustrated the transition from informality to formality. The resulting formal techniques have demonstrated that the two distinct methods of specification can be used compatibly and effectively in the development of real systems.

Acknowledgements

The authors wish to thank all members of the Flagship and EDS groups at the University of Manchester and ICL, in particular Graham Boddy, Sean Holdsworth, Steve Leunig and Ian Watson.

The Flagship work was supported by a UK SERC grant GR/E 21070. The EDS work is supported by a European ESPRIT II grant EP2025.

References