Short Notes

SIMSTRICT: A Behavioural Simulator for use with the STRICT Hardware Description Language

SIMSTRICT is an advanced behavioural simulator for VLSI system design. It can perform verification between levels of the design hierarchy by dynamic behavioural comparison and can simulate incomplete designs. It is integrated with the hardware description language STRICT and autolayout tools to create a powerful computer aided design system, useful at the architectural level as well as gate levels.

Received June 1988, revised November 1990

1. Introduction

In general, simulators can be classified by the degree of abstraction from the physical devices that constitute a design. Circuit stimulators such as SPICE[7] model at a low level using analog primitives such as resistors, capacitors, voltage sources and transistors. Gate simulators like CLASSIC[9] model a network of logic gate primitives (e.g. AND, OR and NOT) using discrete signal values (e.g. 1, 0 and undefined).

Functional simulators such as ELLA[8] and HILO[10] extend this method to allow the designer to define functional primitives for nodes, in the structural network. A hardware description language (HDL) is required as a medium for the functional descriptions and thus these simulators are normally part of a design system which includes language and simulation components. Some functional simulators provide mixed mode simulation[6] so that functional and circuit simulation can be combined. Gate and functional simulators have been described as behavioural simulators[3] and Coelho defines behavioural simulation as 'a tool which supports discrete value simulation of a system composed of black box behaviours described by a HDL'.[5]

SIMSTRICT is a behavioural simulator which uses the hardware description language STRICT.[2] STRICT, like some other HDLs, is hierarchical; design components being composed of networks of sub-components. However STRICT requires that a behavioural description is provided for each level of the hierarchy. Thus SIMSTRICT can extend the behavioural concept and perform dynamic behavioural comparison to verify that the design of a component meets its behavioural specification. It does this as part of an integrated STRICT environment.[4]

SIMSTRICT is a homogeneous level simulator. That is the functional levels of the components being simulated are transparent to the simulation and are wholly dependent on the behavioural descriptions provided in the STRICT language.

The design of the language and the simulator are intimately connected. However, it should be noted that SIMSTRICT supports features not yet embodied in STRICT, such as an AFTER temporal constraint.

The simulation process, for the user, consists of decomposing a component into its constituent interconnected parts (i.e. going down a level in the hierarchy) and then driving it with test signals. The signals are applied interactively, streamed in from data files or generated by an embedded pseudo-component. Output signals are collected from user selected points in the design and displayed interactively or stored in data files for later review or processing. Signals in STRICT are not limited to boolean values, but may be any type available to the behavioural description.

2. The simulation model

SIMSTRICT simulation consists of modelling the design as a network of blocks. These blocks have behaviours that respond to the arrival of signals from the network. We define a behaviour as:

$$B = \{(E, A)\}+.$$

That is, behaviour is a set of at least one causal event E and an associated assignment set:

$$A = \{(C, \{S\}+, T)\}+.$$

Which must contain at least one element. Assignment set elements consist of a selection condition C which specifies the conditions under which the assignment can occur, an assignment function set $\{S\}+$ containing at least one element and a temporal constraint T. The temporal constraint is defined as:

$$T = (a, w, G),$$

where a and w are defined as delay times between which the assignments can occur, such that $1 \leqslant a \leqslant w$. G is a guard function defined as:

$$G = (f, U)$$

where f is a guard period during which the assigned values must not change and U is a causal event which can preempt the completion of the period. It is possible for f or U to be null. Thus a complete definition of a SIMSTRICT behaviour is:

$$B = \{(E, \{(C, \{S\}+, (a, w,)f, U)))\}+)\}+.$$

The mapping of this description onto the STRICT language is best illustrated by some examples. We assume a block with inputs a, b and c, and outputs d and e:

```
intended behaviour
  whenever change (a) OR change (b)   ↔ E
    (e = 1):                          ↔ C
      after 5                         ↔ a
      within 10                       ↔ w
        set d = a+b                   ↔ {S}
            c = a−b
```

Here, a change in the signals at 'a' or 'b' trigger the behaviour. If the condition '$e = 1$' is satisfied then new values are assigned to 'd' and 'c'. These assignments occur between 5 and 10 time units after the causal event.

```
intended behaviour
  whenever change (a)     ↔ E
    within 10             ↔ w
      set d = a mod 2     ↔ {S}
        for 6             ↔ f
        unless change (b) ↔ U
```

Here, when there is a change at 'a' a new value is assigned to 'd'. This occurs between 1 and 10 time units after the causal event mapped to E. It is held at that value for 6 time units or until the signal at 'b' changes. There are no conditions in this example so the assignment is performed whenever the causal event occurs. When **after** is not specified a default of 1 time unit is assumed.

The blocks in the simulation network are of three types:

(i) *Functional blocks*: These correspond to STRICT language BLOCKS and have their behaviours explicitly specified by the **intended behaviour** section of the BLOCK.

(ii) *Network blocks*: These are implied by the **use structure** section of a STRICT BLOCK. Their behaviours are a subset of the definition given above:

$$Bn = \{(E, \{S\}+)\}+.$$

As can be seen, they are composed of only causal events and sets of assignments functions. There are no conditions or temporal constraints. In addition the assignment functions are restricted to those needed to join, split and merge signals.

(iii) *Virtual blocks*: These are implied by special features of STRICT such as clocks. They have full behavioural descriptions like functional blocks.

The blocks are connected by ports. These can be input, output or input/output and are explicitly specified in STRICT. Virtual ports are implied by the use of state variables in STRICT. The ports are connected on a one-to-one basis (see fig. 1). Since all the blocks have similar behavioural formats and network blocks handle the fan-in and fan-out of signals the simulator algorithm can be comparatively simple.

The behaviour of a block is used at two levels. If the block is decomposed and *behavioural comparison* carried out, then the behaviour is used as an *intended behaviour* for the comparison. Decomposition results in a network of sub-blocks. This is simulated using *actual behaviours* for the sub-blocks to produce a *simulated behaviour*. It is the intended and simulated behaviours that are compared.

The SIMSTRICT behaviour as defined above is used as the intended behaviour for a functional block. In order to derive an actual behaviour we proceed as follows:

$$Bi = \{(E, \{(C, \{S\}+, T)\}+)\}+ \quad \text{as defined.}$$

Now E, C and $\{S\}$ can be used directly. However, T has to be transformed to give a time for the assignment of signal values. In general:

$$T = (a, w, G) \succ T' = (d, G') \quad \text{where} \quad d = F(a, w, p, \ldots).$$

F is called an *interpretation function*. This can be any analytic function of a, w, and p such that $a \leqslant d \leqslant w$, where $p, \ldots$ are user supplied parameters. In practice the following function is used:

$$d = a + (w - a)p/100 \quad \text{where} \quad 0 \leqslant p \leqslant 100.$$

The transformation $G \to G'$ is simply one of interpretation. $G = (f, U)$ specifies that a signal is *expected* to remain the same for a guard period. While $G' = (f, U)$ indicates that the signal *will* be present and unchanged during the period.

If we take the time of the causal events E

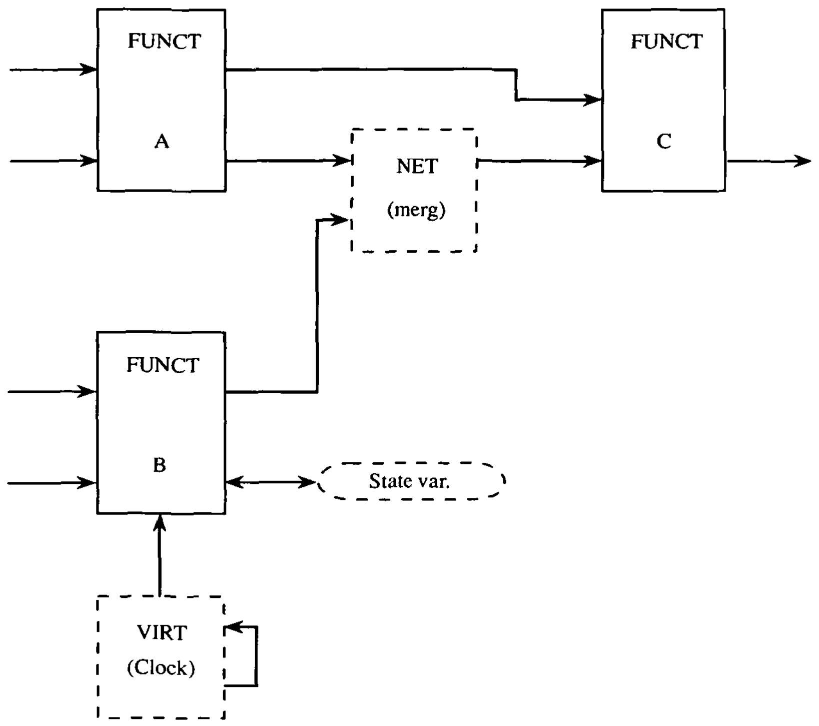

Figure 1. Blocks connected by ports.

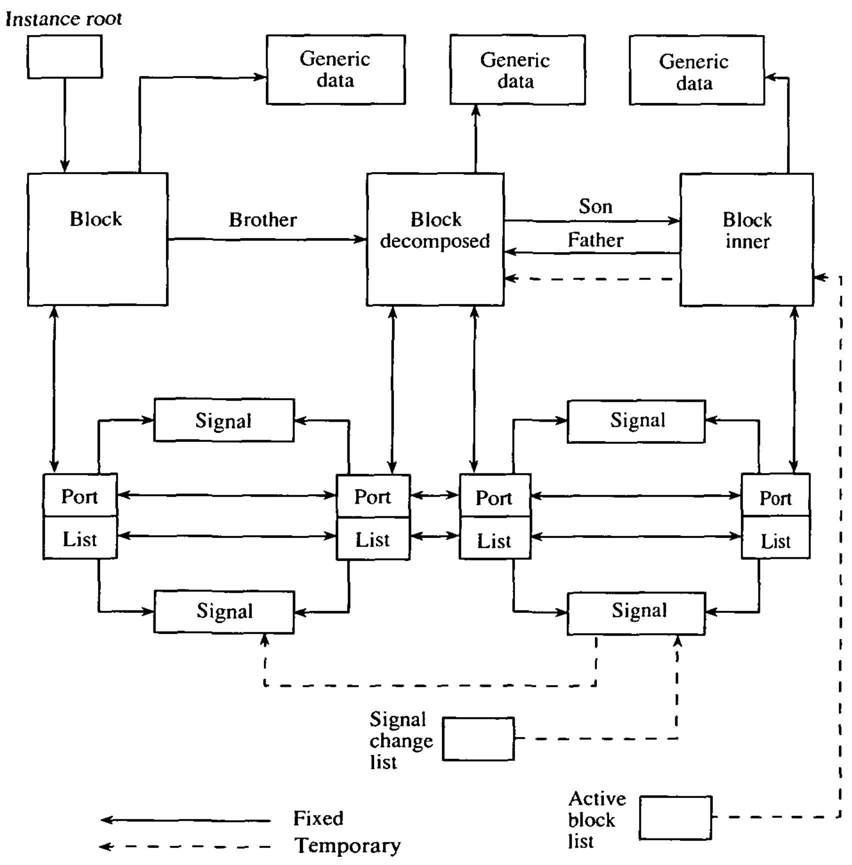

Figure 2. Schematic data structure.

and U to be te and tu respectively. Then an intended behaviour specifies that the $\{S\}$ assignments should occur at some time t such that:

$$te+a \Leftarrow t \Leftarrow te+w.$$

The signal will be expected to remain the same for the period:

$$te+w \text{ to } te+w+f \text{ or, } te+w \text{ to } tu, \text{ where } te+w \Leftarrow tu \Leftarrow te+w+f.$$

The associated actual behaviour is that the $\{S\}$ assignments are maintained for the period:

$$te+d \text{ to } te+w+f \text{ or, } te+d \text{ to } tu, \text{ where } te+d \leqslant tu \leqslant te+w+f.$$

The behavioural models above are used within the context of a simulation cycle which briefly consists of:

(i) Passing signals between blocks. Which causes.

(ii) Evaluation of actual behaviours. Which cause.

(iii) Scheduling of future actual and intended signal changes. When these eventually occur behavioural comparison can be performed. The actual signals cause (i) above.

3. Method

The design of SIMSTRICT can be divided into three broad areas: generic, schematic and events. These in turn can be described as data structures and related algorithms.

The event data structure and algorithm are the event queue and the event manager respectively. The event queue is used by the simulation management algorithm for the storage and subsequent retrieval of signal and control information 'in the future', and to step the simulation through time. A number of possible methods can be used, such as linked lists or rings. A simple linked queue method was selected for ease of implementation on the understanding that other methods could be substituted as required.

3.1. *The Data Structures*

The schematic data structure contains components (i.e. records) for each instantiation of a functional, network or virtual block; for each port on every block; and for each signal passing between a pair of ports. A linked structure (i.e. pointers) is used to configure these components. This results in a threaded structure with alternative paths to allow efficient algorithms for both enquiry and simulation to be used. Links are also used to connect each schematic block to its associated generic data structure (see below). When a block is decomposed the data structure is extended but the data components already associated with the block being decomposed are not removed. These remain integrated to support behavioural comparison. Thus the schematic data structure is dynamic. Its size and structure is controlled by the degree of decomposition requested by the user, while its contents change as the simulation proceeds. (See figure 2).

The generic data structure contains behavioural descriptions for the block instances in the schematic data structure. The generic data structure is composed of a number of separate behavioural descriptions called generic blocks. Each of these generic blocks is referenced (by a pointer) from the blocks in the schematic structure. A generic block can be used by more than one schematic block.

The generic data structure is designed to support all the features of STRICT's intended behaviour. A generic block is a linked structure containing a number of expression lists relating to the various clauses in STRICT's intended behaviour section (see figure 3). The expression lists are composed of stack language statements in a coded form. A stack language evaluator is used to process them

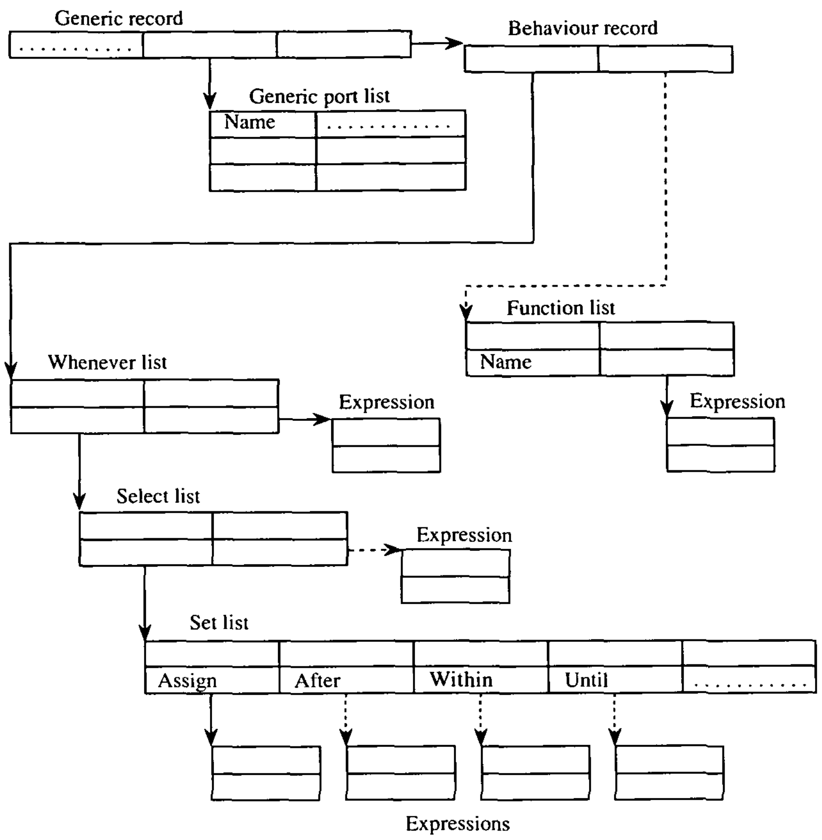

Figure 3. Genetic data structure.

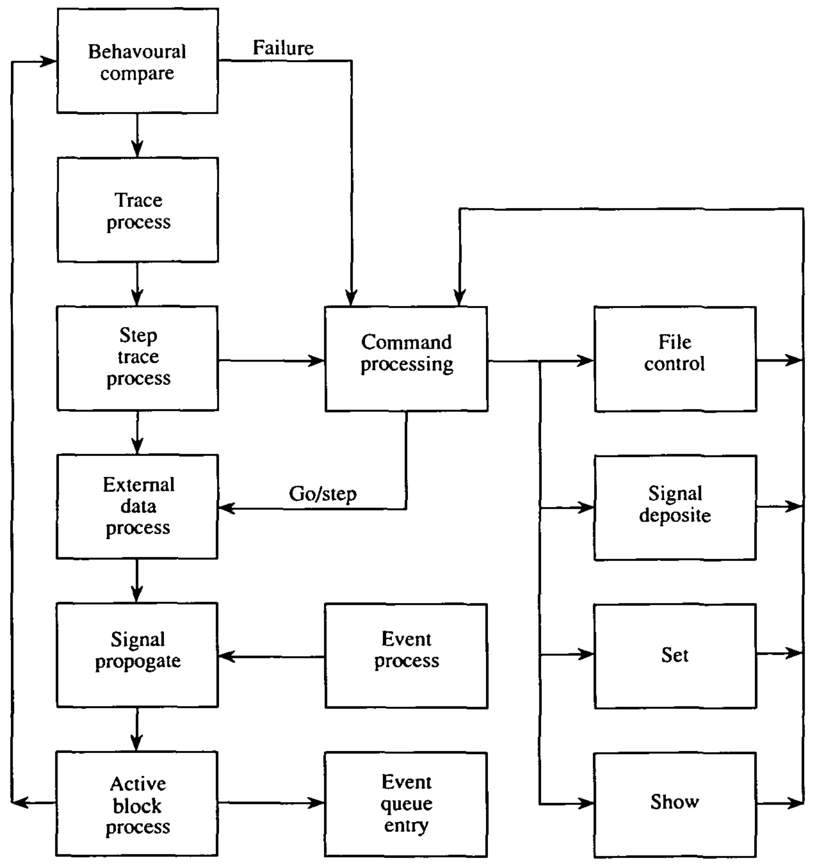

Figure 4. Simulator control flow.

and to support function calls. The generic structures does not change during the simulation, unlike the schematic data.

3.2. *The Algorithm*

Having considered the data structures and some of the minor algorithms used in the simulator, we can now discuss the simulation management algorithm in detail. The primary algorithm is event driven and is illustrated in figure 4. This broadly consists of:

(i) *Signal propagation* through the network blocks of the schematic structure, which results in the activation of the target blocks. This is assumed, at the moment, to be instantaneous.

(ii) *Active block processing* which accesses the generic data structure to evaluate behaviours and which may thus enter new signal values into the event queue.

(iii) *Event processing* which steps the simulation forward in time to the next block of events and applies them to the output ports of the blocks in the schematic structure.

(iv) Application of external data to ports in the schematic data structure if required.

This asynchronous algorithm steps over periods when there is no activity by obtaining the current time from the event queue. The propagation of signals through the schematic data structure is such that only those function blocks which might be affected are processed.

Behavioural comparison, when required, is performed as follows: When signals arrive at a decomposed block intended behaviour events are scheduled and behaviour *demons* are attached to the block. The signals are then passed through into the block for the simulation of its structure. The demons monitor causal completion of guarded periods (i.e. *U* in the model defined above) and can be triggered by the subsequent arrival of another signal. The output ports of a decomposed block have states belonging to a finite set. The action taken on the arrival of events, such as signal assignments and intended behaviour events, or the triggering of demons depends on the current state of the target allocated port. The states indicate what future events are expected if the intended behaviour is to be met.

3.3. *Implementation*

SIMSTRICT has been implemented in PASCAL under VMS on a VAX/750. A modular structure has been used to facilitate modifications (e.g. replacing the event queue manager).

The simulation manager has been described above and is supported by a number of interfaces:

A *language interface* gives information about the behaviour and structure of the design being simulated. This information is derived from the STRICT language definition of the design.

A *command interface* provides a route for the user to drive the simulator. A comprehensive range of commands are provided which allow the user to:

- Set up the simulation environment by decomposing selected blocks and choosing which signals to monitor.
- Deposit and examine signals.
- Display the schematic and generic data structures.
- Control the operation of the simulator

A *file interface* provides facilities for communicating with text terminals and supports files for:
- Logging the simulation run.
- Collecting output data for post processing.
- Supplying test data streams.
- Supplying command streams.

There is an optional format for input and output files that is compatible with that used by hardware test equipment. Simulation results can thus be directly confirmed on the fabricated design. Test data, as already stated, can be generated by pseudo components embedded in the design. This data can be collected in test data file format so that the STRICT language can be used as an independent test pattern generator for subsequent SIMSTRICT runs.

SIMSTRICT has been implemented to support processing in integrated interactive, standalone interactive and standalone batch modes. When integrated it is executed from the SAGA language editor[1] which forms the core of the STRICT design system. In standalone modes SIMSTRICT uses design files generated by a previous run of the language editor. It is used interactively for initial exploration of the design. However, large amounts of test data may be needed to fully verify a design, so a standalone batch model of operation is also provided. Command files can be used to execute predefined command sequences in all modes. The command sequences used during all or part of an interactive session can be stored and used to reproduce an identical run on, for example, a modified design.

4. Conclusions

SIMSTRICT is a powerful simulation tool for VLSI circuit design. It is integrated with the STRICT language and a fully automated layout tool to provide a comprehensive design environment not available elsewhere.

Many behavioural simulators are currently available but SIMSTRICT offers features which, for the first time, allow interactive verification between levels of the design hierarchy as part of a top-down philosophy. Dynamic behavioural modelling, involving demons attached to blocks and a finite state algorithm, is used to provide behavioural comparison. An interpretation function allows the simulation model to be interactively modified so that the boundaries of a behavioural specification failure can be explored.

Selective decomposition allows detailed investigation of the design without the time penalty of full decomposition. Incomplete designs can be simulated so that SIMSTRICT can cooperate in the stepwise refinement of a design. Comprehensive command and file interfaces support SIMSTRICT and give great control over its operation and increase its effectiveness.

Thus SIMSTRICT is able to act as a check on the design process at an earlier stage than other simulators.

4.1. *Future Work*

The development of SIMSTRICT will have to keep pace with and play an active role in the evolution of the STRICT language.

A prototype graphical display post processor has already been written but fully interactive graphics can be provided by the development and integration of a graphics interface.

The use of the simulator as an interactive language debugger, similar to those used for software programming languages, will enhance the system.

An intriguing use of SIMSTRICT is the simulation of high level systems, such as local area networks. The concept of interpretation functions can be extended to include probability distributions and thus allow a form of Monte Carlo simulation to be carried out.

5. Acknowledgements

The authors extend their thanks to their colleges in the Newcastle VLSI design group for their contributions to this work.

The work was carried out under UK Science and Engineering Research Council funding. It was started while A. P. Robson was a member of the Department of Electrical and Electronic Engineering, University of Newcastle upon Tyne, and was continued collaboratively after his move to the Department of Computing, Newcastle upon Tyne Polytechnic.

A. P. ROBSON*‡ AND D. J. KINNIMENT†

* Department of Computing, Newcastle upon Tyne Polytechnic, Ellison Building, Newcastle upon Tyne, NE1 8ST.

† Department of Electronic and Electrical Engineering, The University of Newcastle upon Tyne.

‡ To whom correspondence should be addressed.

References

1. R. H. Cambell and P. G. Richards, SAGA: a system to automate the management of software production, *AFIPS50*, 231–234 (1981).
2. R. H. Cambell, A. M. Koelmans and M. R. McLauchlan, STRICT A Design Language for Strongly Typed Recursive Integrated Circuits, *IEE Proc.* **132**, (E), (I), no. 2 March/April 108–115 (1985).
3. S. E. Evanczuk, Getting Out of the Gate: High Level Modelling in Circuit Design, *VLSI Design*, Jan, 60–66 (1985).
4. A. M. Koelmans, M. R. McLauchlan and A. P. Robson, The STRICT language and Design Methodology, *Proc. Electronics Design Automation Conf.*, 79–86 (1987).
5. D. R. Coelho, Behavioural Simulation a Survey, Automated Design and Engineering for Electronics – East, Proc. of tech. sessions BOSTON, MA, USA, Oct, 88–94 (1985).
6. D. Munns, A Hierarchical Mixed Mode Simulator, *Proc. Electronic Design Automation Conf.*, 573–578 (1987).
7. G. Barros, A Circuit Simulation Tutorial VLSI Design, 110–120 (1985).
8. Praxis Systems plc, The ELLA System Overview (1985).
9. Plessey Semiconductors Ltd. Classic Reference Manual (1986).
10. GenRad Inc. HILO-3 Users Manual (2522–0100) (1985).

Occupancy Models for the Estimation of Block Accesses

Estimating the number of block accesses required to retrieve a set of records is an important problem in the design and implementation of data and knowledge systems. The models underlying some of the proposed estimation formulas are discussed as variations of occupancy problems, which have been extensively discussed in the literature in probability and statistics. Some additional properties of the relevant distributions are presented. In addition, an approach based on a sequential occupancy problem is discussed. This approach, which involves the estimation of the number of records which will require the retrieval of a target number of blocks, can be used to avoid the estimation of block accesses in some situations.

Received November 1990, accepted November 1991

1. The block access estimation problem

The block access estimation problem involves the estimation of the number of blocks that must be accessed to retrieve a desired set of records. This problem arises in the context of query optimisation and physical database design in information systems. More formally, a set of k records is to be retrieved from a file of n records that has been divided into m blocks. The blocking factor p is equal to n/m; p is usually assumed to be integral. The number of blocks, $b(m,p,k)$, that must be accessed to retrieve the records is to be estimated.

This problem has been addressed by a number of authors, including Cardenas,[4] Chan and Niamar,[5] Cheung,[6] Christodoulakis,[7-9] Ijbema and Blanken,[12] Luk,[18] Maio, Scalas and Tiberio,[19] Palvia and March,[25] Pezarro,[27] Waters,[37-39] Whang, Wiederhold and Sagalowicz,[40] VanderZanden, Taylor and Bitton,[35,36] Yao,[41,42] Yue and Wong[43] and Zahorjan, Bell and Sevcik.[44] Two primary formulas for estimating the number of block accesses have been proposed. In addition, a number of variations and approximations have been developed.

The first estimation formula for the expected number of block accesses was presented independently by Cardenas[4] and Waters.[38] It is

$$b(m,p,k) = m\left[1-\left(1-\frac{1}{m}\right)^{k}\right].$$

The second estimation formula is based upon a different model of the block access problem. This formula was presented independently by Pezarro,[27] Waters[39] and Yao.[41,42] It is

$$b(m,p,k) = m\left[1-\frac{\binom{n-p}{k}}{\binom{n}{k}}\right]$$

$$= m\left[1-\prod_{i=1}^{k}\frac{nd-i+1}{n-i+1}\right],$$

where $$d = 1-\frac{1}{m}.$$

In both cases the problem is modelled as one of selecting k records from m blocks. The record distributions are assumed to be independent, and the blocks are assumed to be