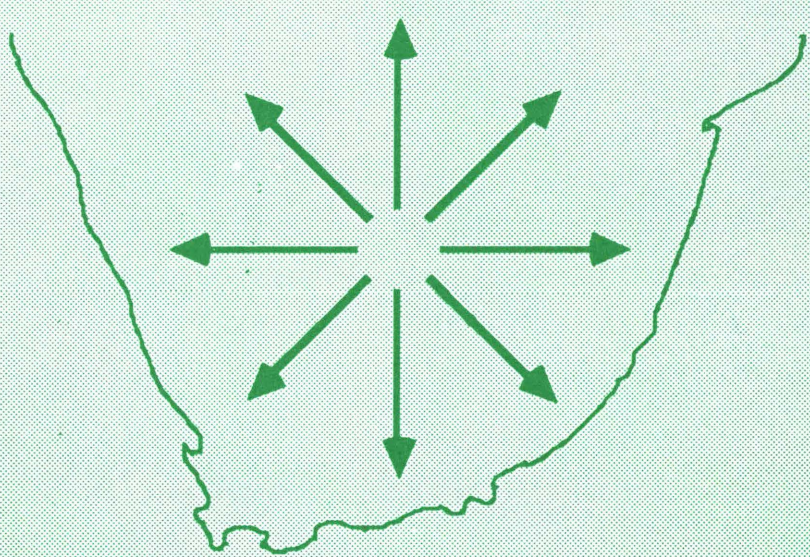
South African Computer Journal
Number 7
July 1992

Suid-Afrikaanse Rekenaar tydskrif
Nommer 7
Julie 1992

SPECIAL ISSUE



7th SA COMPUTER RESEARCH SYMPOSIUM

The South African Computer Journal

An official publication of the South African Computer Society and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif

'n Amptelike publikasie van die Suid-Afrikaanse Rekenaarvereniging en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
Email: dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof John Schochot
University of the Witwatersrand
Private Bag 3
WITS 2050
Email: 035ebrs@witsvma.wits.ac.za

Productio.. Editor
Prof G de V Smit
Department of Computer Science
University of Cape Town
Rondebosch 7700
Email: gds@cs.uct.ac.za

Subscriptions

	Annual	Single copy
Southern Africa:	R45,00	R15,00
Elsewhere	$45,00	$15,00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

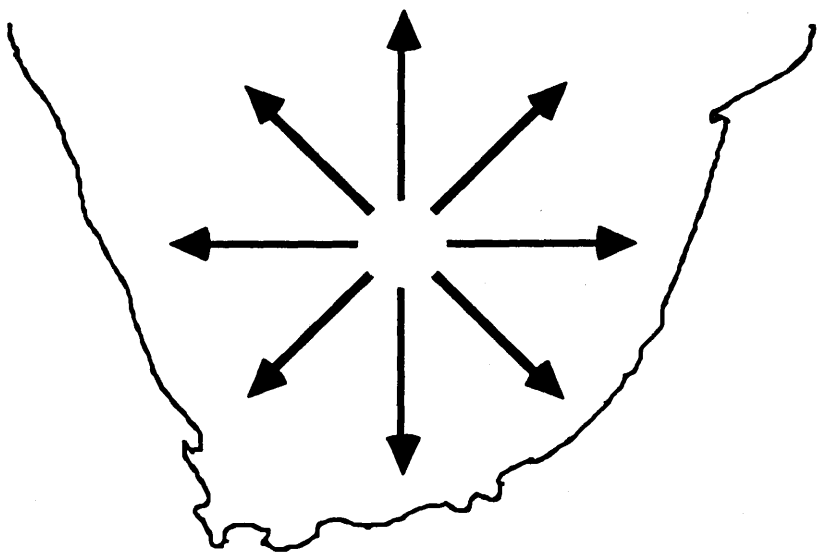
7th Southern African Computer Research Symposium

Karos Indaba Hotel, Johannesburg

1 July 1992

PROCEEDINGS

Guest Editor: Judy M Bishop



PERSETEL

Organised by the SA Institute of Computer Scientists
in association with the Computer Society of SA

Sponsored by Persetel and the FRD

SPECIAL ISSUE - 7th SA COMPUTER SYMPOSIUM

PREFACE

When the first SA Computer Symposium was held at the CSIR in the early eighties, it was unique. There was no other forum at the time for the presentation of research in computer science. In the intervening decade, conferences, symposia and workshops have sprung up in response to demand, and now there are several successful ventures, some into their third or fourth iteration. Each of these addresses a specific topic - for example, hypermedia, expert systems, parallel processing or formal aspects of computing - and attracts a specialised audience, well versed in the subject and eager to learn more. For the main part, the proceedings are informal, and certainly not archival.

SACRS, though, is still unique, in that it deliberately covers a broad spectrum of research in computing, and in addition, seeks to provide a lasting record of the proceedings. To achieve the second aim, we negotiated with the SA Institute of Computer Scientists for the proceedings to form a special issue of the SA Computer Journal, and the copy you have in front of you is the result. The collaboration between the symposium committee and the journal's editorial board placed high standards on the refereeing and final presentation of the papers, to the symposium's benefit, while we were still able to maintain a fresh, audience-oriented approach to the selection of papers.

This is SACJ's first such special issue, and the largest issue (at 145 pages) to date. We hope that it is only the beginning of future such collaborations.

In all 29 papers were received, all were refereed twice, and 19 were chosen for presentation by the programme committee. All the papers were thoroughly revised by the authors on the basis of the referee's comments, and the committee's suggestions aimed at making the material more accessible to a broadly-based audience. Papers had to be new, and not to have been presented elsewhere, a requirement that is still unusual within the SA conference round.

A third goal of SACRS has been to invite keynote speakers, usually from overseas. This year, we are fortunate to present Dr Vinton Cerf, the father of the Internet and a world-renown expert on computer networks. Although his paper is not available for this special issue, it will appear later in SACJ. Through the good offices of Professor Chris Brink of UCT, we also have three other speakers from Germany, Canada and the US adding interest to the event, and two of their papers appear in this issue.

The programme committee originally devised a theme for the symposium - "Computing in the New South Africa". We received several queries as to the meaning of this theme, but unfortunately few papers that addressed it directly. One prospective author went as far as to enquire whether computer research would survive in the new South Africa. Another felt that his work was definitely not in the theme, as it was genuine, old world, basic, theoretical science! Neverthless, there are two papers that consider one of South Africa's key issues, that of language. Others look at the success we have achieved in applying technology to mining, and the future of low-cost operating systems. In all, the mix of papers represents a balance between the theoretical and the practical, the past and the future, all firmly based in the computing of the present.

Organising the symposium has involved the hard work of several people, and I would like to thank in particular

- Derrick Kourie, my co-organiser, and the editor of SACJ for his invaluable advice and hard work throughout the planning and implementation stages;
- Riël Smit, the production editor, for attaining such a high standard in such a short time for so many papers;
- Gerrit Prinsloo and the staff at the CSSA for their efficeint and quite delightfully unfussy organisation;
- Persetel for their very generous sponsorship of R25000, and Tim Schumann for taking a genuine interest in our events;
- the Foundation for Research Development for sponsoring Vint Cerf's visit;
- and finally the Department of Computer Science of the University of Pretoria for providing the ideal working conditions for undertaking ventures of this kind, and especially Roelf van den Heever for his unfailing encouragement and support.

Judy M Bishop
Organising Chairman, SACRS 1992
Guest Editor, SACJ Special Issue

Referees

The journal draws on a wide range of referees. The following were involved in the refereeing of the papers selected for this special issue. Their role in certifying the papers and their contribution to enhancing the quality of papers is sincerely appreciated.

John Barrow	UNISA
Ronnie Becker	University of Cape Town
Danie Behr	University of Pretoria
Sonia Berman	University of Cape Town
Liesbeth Botha	University of Pretoria
Theo Bothma	University of Pretoria
Chris Brink	University of Cape Town
Peter Clayton	Rhodes University
Ian Cloete	Stellenbosch University
Antony Cooper	CSIR
Elise Ehlers	Rand Afrikaanse University
Quintin Gee	DALSIG
Andy Gravell	University of Southampton
Wendy Hall	University of Southampton
David Harel	The Weizmann Institute of Science
Scott Hazelhurst	University of the Witwatersrand
Derrick Kourie	University of Pretoria
Willem Labuschagne	UNISA
Doug Laing	ISM
Dave Levy	University of Natal
Graham McLeod	University of Cape Town
Hans Messerschmidt	University of the Orange Free State
Deon Oosthuizen	University of Pretoria
Ben Oosthuysen	University of Pretoria
Neil Pendock	University of the Witwatersrand
D Petkov	University of Natal
Martin Rennhackkamp	Stellenbosch University
Cees Roon	University of Pretoria
Jan Roos	University of Pretoria
John Shochot	University of the Witwatersrand
Morris Sloman	Imperial College, London
Riel Smit	University of Cape Town
Pat Terry	Rhodes University
Walter van den Heever	University of Pretoria
Lynn van der Vegte	University of Pretoria
Herman Venter	University of Fort Hare
Herna Viktor	Stellenbosch University

ESML—A Validation Language for Concurrent Systems

P J A de Villiers and W C Visser
Institute for Applied Computer Science, University of Stellenbosch, Stellenbosch 7600

Abstract

Designing a concurrent reactive system which can be proven correct is a challenging task. A promising technique involves building a validation model which can be shown to have important correctness properties. This paper describes a language to specify such models. A model consists of one or more interconnected state machines. The global state of a model is the combined state of all state machines. Channels which enable state machines to communicate form part of the global state. Temporal logic is used to specify correctness requirements of a model and a validation system (based on model checking) can be used to check these requirements. Design errors such as deadlock can thus be detected.
Keywords: *Verification, Specification, Temporal logic, Concurrency*
Computing Review Categories: *D2.1, D2.4, F3.1*

Received March 1992, Accepted April 1992

1 Introduction

Formal methods supported by efficient software tools are now available to simplify the design and validation of complex protocols [11, 12]. Although these tools may not be useful to many people—protocols are standardised nowadays and few people design their own protocols—we are discovering that similar techniques can be applied to develop other reactive systems for which no "standard" designs exist. The correctness of even small reactive systems can be important when used in life-critical control applications. Examples are: process control systems, flexible manufacturing systems and systems used in aviation. It is therefore a challenge to develop effective tools for the design and validation of these systems.

Most software systems are still designed and specified by using only diagrams and descriptions in natural language. To make this process less error prone we need

- formal specification languages to describe the *behaviour* of a system and state *requirements* to be met by a design.
- effective *validation tools* which can be used to check that a design meets its requirements.

There are two approaches to specification: *single* language and *dual* language [17]. In the single language approach the specification of the *behaviour* of a system is assumed to capture the real intentions of the designer. Most specifications are complex, however, and may contain errors. It is so that the very formality of the notation forces one to think hard about the problem which helps to eliminate errors. But effective tools are not yet available to *prove* that such designs are "correct". The reason may be that currently available specification languages such as VDM [14] or Z [19] were not designed with mechanical verification in mind.

Manna and Pnueli introduced the dual language approach to specification based on *fair transition systems* and *temporal logic* [18, 15, 16]. A set of *requirements* are specified in addition to specifying the behaviour of a system. Temporal logic is used to specify typical requirements for reactive systems such as *responsiveness* and *deadlock freedom*. The advantage of the dual language approach is that a validation tool can be used to check the behavioural specification against a set of requirements. Such a tool can be based on model checking [3, 4, 2]—an efficient technique to search for error conditions. A system designer is thus provided with a secure method to check a specification for correctness.

We are developing a validation system for reactive systems which is based on the dual language approach. To specify requirements we use the temporal logic CTL as first defined in [3]. We have developed a model checker to verify a model against its requirements specification [5]. Unfortunately existing specification languages include unbounded constructs which make model checking impossible. We therefore decided to design an experimental validation language which is tailored to the efficient model checking of reactive systems. CSP [9] was accepted as a framework for the design. The language is described in this paper and is called ESML (Extended State Machine Language).

2 Model Checking System

Extended state machines (ESMs) were chosen to model reactive systems. The state machine formalism is one of the oldest and most widely accepted models of computation and provides a natural medium to describe the dynamic behaviour of complex reactive systems. An ESM is a state machine enhanced with data variables, guarded actions, concurrency and communication. CSP was adopted as a basis for concurrency and communication: unbuffered, blocking communication primitives are used to transmit messages over channels (which are accessed through ports). A location variable is associated with each ESM to in-

dicate which guarded action it should execute next. A brief overview of the most important features of our model checker is presented here to motivate the design of ESML. For details the reader is referred to [5].

The contents of all data variables, location variables and ports are collectively called the global system state. It is encoded as compactly as possible in a data structure called the *state vector*. Although early model checkers generated a graph of the entire reachable state space to be analysed during subsequent verification phases, we use an "on-the-fly" technique which is more efficient [12]: instead of generating the entire state space (which may be impractically large) only the subspace necessary to verify each requirement is generated.

Model checking is considered to be one of the most promising validation techniques available and is far more than a mere "brute force" state exploration technique. Researchers are currently launching an attack on the so-called state explosion problem by attempting to find methods to *prevent unnecessary states from being generated at all* [13, 20, 8]. Holzmann developed an efficient technique to determine whether each newly generated state has been visited before [10]. This allows a substantial amount of unnecessary work to be avoided. In a nutshell, the state vector is used as an index into a bit vector. Thus each possible global state corresponds to a unique bit in the bit vector. All bits are set to zero initially. For each newly generated state its corresponding bit in the bit vector is tested. If the bit is zero, it is set to one to indicate that the state has been visited; otherwise the state can often be ignored. Our model checker incorporates this technique and also uses a special memory allocation technique to save space by exploiting the sparseness of the bit vector.

To produce an executable validator the ESML model is translated into functionally equivalent code that is linked to the model checker. The code corresponding to the ESML model can be seen as a state generator which is (indirectly) controlled by the model checker. The model checker automatically determines which states are to be evaluated in order to verify a given requirement specification. When designing ESML we set ourselves the following goals:

- The language should support modular design and complex data structures.
- The state vector should be kept as small as possible to keep the corresponding bit vector down to a manageable size.

3 ESML Validation Language Overview

A reactive system consists of a *controller* (typically a hierarchy of cooperating components) reacting to signals from some external environment. A validation model must specify the behaviour of a system (the specification proper) and corresponding correctness criteria (the requirement specification).

We decided to use a small kernel [7] which supports processes and message passing as a representative example of the kind of reactive systems to be validated. CSP [9] was selected as a design framework for our validation language because it provided an effective guideline during the design of the kernel and seems to be generally applicable. The design of ESML was inspired by two existing languages: the systems programming language Joyce [1] and the validation language PROMELA [11]. Joyce is a strongly typed programming language based on CSP. It provided a design framework for ESML. PROMELA was designed to model and validate protocols. It shows which constructs can be implemented efficiently.

After studying PROMELA and also gaining some experience by modelling several fragments of the kernel we made the following observations:

- Dijkstra guarded commands [6] as used in PROMELA provide a powerful and convenient control structure.
- PROMELA supports rather sophisticated interprocess communication: synchronous as well as asynchronous message passing which can be buffered or unbuffered. With CSP as a design framework, communication is simpler: only unbuffered synchronous message passing is needed.
- Process creation in Joyce is simpler than in PROMELA. Shared variables are forbidden in Joyce and communication provides the only method of exchanging information between processes. This simplified framework for concurrency makes it possible to identify independent concurrent transitions. Only one of many possible interleavings of events need be analysed when concurrent transitions are independent [8, 13].
- To model the kind of reactive systems we have in mind, two constructs are needed for data structuring: *records*—to group related data together—and *sequences*. At the level we wish to specify systems, the actual implementation of sequences is not important. We thus decided to avoid indexed arrays as used in PROMELA. It is sometimes necessary to combine these structures to form more complex structures. For example, a typical data structure is a queue of process records associated with the process scheduler in the kernel we wish to model. Each process record groups together related data about a specific process (process number, process state) while the queue is a sequence of such process records. Model checking requires the state vector to be of finite length. Since all data structures form part of the state vector, sequences must be restricted to be finite (we call them *lists*). Records are finite by definition.

A model of a reactive system consists of a hierarchy of ESMs (similar to Joyce **agents**). An ESM can contain typed variables as well as other ESMs. Standard operators are provided to manipulate booleans and integers. ESMs execute concurrently once created and may communicate over channels. ESMs take the place of modules in a design.

Extended BNF notation is used to define the ESML grammar: $[\alpha]$ denotes the sentence α or the empty sentence and $\{\alpha\}$ denotes a finite sequence of sentences α or the empty sentence.

Type and Variable Definitions

Variables form part of the state vector and therefore compact representation is important. Variable ranges are fixed by type definitions. Type BOOLEAN is predefined and is represented by a single bit. Instead of predefining a number of standard types (say, 16 bits for an integer), subranges are declared in order to allocate just the right number of bits. For example, if numbers can range from 0 to 10 in a specific application, a type *Number* could be defined as a subrange 0..10 to fit into 4 bits. This idea would probably be awkward in a programming language. For example, it raises several questions about type equivalence. We settle these issues in a simple way by using *name equivalence* for type checking. Two different subranges of the integers are thus not of the same type. Although it is possible, the intention is not to define several different subranges of the integers in the same validation model. A user should define a single type (a subrange of integers) which is suitable for all numbers in a validation model. Although somewhat restrictive, we feel that this scheme can be tolerated here because only a limited number of variables can be afforded in any validation model. The keywords "TYPE" and "VAR" indicate type and variable definitions respectively.

```
TypeDef =
  TypeName "=" NewType ";".
NewType =
  NewListType | NewRecordType |
  NewPortType | NewSubrange.
NewSubrange =
  Numeral ".." Numeral.
```

Examples:

```
TYPE Number = 0..10;
VAR counter1, counter2: Number;
```

Port Types

Brinch Hansen found that the most common errors in Joyce programs were type errors in communication commands. He concluded that "any CSP language must include message declarations which permit complete type checking during compilation". For this we found it convenient to adopt the idea of a *port type* as defined in Joyce. A port type T defines an *alphabet* which is a set $\{s_0(T_0), s_1(T_1), ..., s_n(T_n)\}$ of symbol classes. The values in each symbol class $s_i(T_i)$ are formed by prefixing each value of type T_i with the name s_i. Each value T_i represents a message. Symbol classes make it possible to group related messages together conveniently. A special symbol class can be defined with no associated messages. This is called a signal. "EndStream" in the example below is an example of a signal.

```
NewPortType = "{" Alphabet "}".
Alphabet =
  SymbolClass { "," SymbolClass }.
SymbolClass =
  SymbolName [ "(" MessageType ")" ].
MessageType = TypeName.
```

Examples:

```
TYPE Number = 0..10;
TYPE Stream =
  {value(Number), EndStream};
```

An ESM sends a message to be received by another ESM by using two simple communication commands. A message m which belongs to symbol class S of alphabet T is sent via a port p by the command `p!S(m)`. Similarly a message m which belongs to symbol class S is received via a port p and assigned to a variable x of type T by writing `p?S(x)`. When a message is sent, the sending ESM waits until a matching receive command is executed by another ESM. Similarly an ESM wanting to receive a message waits until a message of the specified type is sent.

Structured Data

Lists (sequences of finite length) and records are used to model commonly used data structures. The maximum length of a list is specified by a constant between square brackets as shown in the example below. Combinations of these structures such as a list of records, a record of lists or a list of lists are also allowed.

```
NewListType =
  "LIST" "[" Constant "]" "OF" TypeName.
NewRecordType = "(" FieldList ")".
FieldList =
  RecordSection { ";" RecordSection }.
RecordSection =
  FieldName RecordTail.
RecordTail =
  "," RecordSection | ":" TypeName.
```

Examples:

```
TYPE Sequence = LIST [5] OF Number;
TYPE ProcessRecord =
  (ProcessNumber, Priority: Number);
```

The standard operations "::", "HEAD", "TAIL", "FIRST", "LAST" and "LENGTH" are available to manipulate lists. The operation "::" is used to catenate a single new value to either end of a list. "HEAD" returns the first element of a list. "TAIL" returns the list without its first element. "FIRST" returns the list without its last element. "LAST" returns the last element in a list and "LENGTH" returns the number of elements in a list.

Fundamental Instructions

Arithmetic expressions based on the four basic operators as well as logical expressions with *and* ("/\"), *or* ("\/"), *not* ("~") and the usual relational operators are supported. After evaluating an expression the result can be assigned to a type compatible variable. Range checks are always done before assignments. A control structure is provided by Dijkstra guarded commands:

```
IfStatement =
  "IF" GuardedCommandList "FI".
DoStatement =
  "DO" GuardedCommandList "OD".
GuardedCommandList =
  GuardedCommand { "[]" GuardedCommand }.
GuardedCommand =
  Expression "->" StatementSeq.
```

```
StatementSeq =
  Statement { ";" Statement }.
Statement = AssignmentStatement |
  IfStatement | DoStatement |
  ESMStatement | InputOutputStatement.
```

Examples:

```
IF x >= 0 -> z:= x
[] x <= 0 -> z:= -x
FI;
DO c > 0 -> c:= c - 1 OD;
```

Concurrency and Communication

A new ESM is activated (created and started) by executing an ESM statement. Every ESM may activate additional ESMs. An ESM cannot terminate until all ESMs created by it have terminated.

```
ESMStatement =
  ESMName [ "(" ActualParamList ")" ].
```

During the activation of an ESM new copies of all its variables are created by allocating space in the state vector. Recursive creation of ESMs (like in Joyce) is not supported because the small number of ESMs which can be created (due to the limited size of the state vector) renders such a feature hardly usable. ESMs are therefore similar, but not identical to Joyce agents. Two kinds of parameters are supported: value parameters and ports. Ports are marked by one of the keywords "IN" or "OUT" to indicate the direction of transfer. An ESM statement must provide an actual parameter (of matching type) for every formal parameter defined by the ESM definition. Space is allocated in the state vector for every formal parameter. In the case of value parameters the value of the corresponding actual parameter is assigned to each formal parameter. When an ESM is activated its ports are mapped onto the corresponding ports of the creator to form *channels*. An ESM definition consists of an ESMname, formal parameters and body.

```
ESM =
  "ESM" ESMName Block ESMName
Block =
  [ "(" FormalParamList ")" ] ";" ESMBody.
FormalParamList =
  ParamDef { ";" ParamDef }.
ParamDef =
  "IN" VariableGroup  |
  "OUT" VariableGroup |
  VariableGroup.
ESMBody =
  [ ConstantDefPart ]  [ TypeDefPart ]
  [ VariableDefPart ]  { ESM ";" }
  "BEGIN" StatementSeq "END".
```

Scope of Named Entities

ESMs can be nested and may declare variables, types and constants as named entities. The scope of a variable x declared in an ESM A extends from directly after its declaration to the end of A. However, x is unknown in any ESMs contained in A. There are no global variables. The scope rules for constants and types are slightly different. Constants and types are known from directly after their declaration to the end of the ESM containing the declaration unless they are redefined by some nested ESM.

Specifying Requirements

Temporal logic provides a powerful tool to specify correctness requirements of reactive systems. A classification of such requirements is given in [16]. We adopted the branching time temporal logic CTL as first defined in [3]. Most requirements are expressed in one of two general forms:

- For *all* execution sequences a property α holds *globally* (in every state). This is written as $AG(\alpha)$.
- For *all* execution sequences a state *can be found* where a property α holds. This is written as $AF(\alpha)$.

These basic forms can be used to specify meaningful requirements concisely. For example, let the fact that two processes are not both inside their critical regions C_1 and C_2 be expressed by $\alpha = \neg(C_1 \wedge C_2)$. Mutual exclusion requires that property α must hold for all states along all execution sequences. This is specified by $AG(\neg(C_1 \wedge C_2))$. For a more detailed discussion of the specification of requirements the reader is referred to [5].

A validation model consists of an ESM (which may contain other ESMs) followed by a requirement specification (a CTL formula).

```
ValidationModel =
  "MODEL" ESM ";" Requirement "END".
Requirement = "ASSERT" CTLFormula.
```

The validation system will determine whether the requirement following the keyword "ASSERT" is satisfied by the given ESM. If the requirement specification is violated, the validation system will produce a trace which enables a user to locate the error.

4 Examples

Objects

Figure 1 shows a queue which stores integer values in the range 0..5. The operations *put*, *get*, *isEmpty* and *isFull* can be performed on the queue. It is implemented by defining an ESM for the queue object, a receive (*IN*) channel for the operations and a send (*OUT*) channel for the results to be returned. The ESM *IntQueue* has two channels: *input* for operations and *output* for results. The example shows how the queue object is defined by means of an ESM. Requests are sent via the *intQ* channel and the results are received on the *Result* channel. For example the instruction sequence *intQ!get; Result?item(x)* would return the first item from the queue and assign it to x.

Resource Allocation

Two ESMs A and B share a resource. A and B use the *Req* channel to request the resource and receive it via the *Res* channel. The formula `AG((Req = Get) => AF(Res = OK))` following the keyword ASSERT states the requirement that "when a *Get* signal is put on the *Req* channel an *OK* signal must eventually appear on the *Res* channel".

```
ESM QueueSystem;
  TYPE
    INTEGER = 0..5;
    QueueInp =
      {put(INTEGER),
      get,
      isEmpty,
      isFull};
    QueueOut =
      {item(INTEGER),
      empty(BOOLEAN),
      full(BOOLEAN)};
  VAR
    intQ: QueueInp;
    result: QueueOut;

  ESM IntQueue(
    IN in: QueueInp;
    OUT out: QueueOut);
    CONST Qmax = 3;
    TYPE
      QueueType =
        LIST[Qmax] OF INTEGER;
    VAR
      queue: QueueType;
      x: INTEGER;
  BEGIN
    DO TRUE ->
      IF in?put(x) ->
        queue:= queue::x
      [] in?get ->
        x:= HEAD(queue);
        out!item(x)
      [] in?isEmpty ->
        out!empty(LENGTH(queue) = 0)
      [] in?isFull ->
        out!full(LENGTH(queue) = Qmax)
      FI
    OD
  END IntQueue;

BEGIN
  IntQueue(intQ,result)
END QueueSystem
```

Figure 1. A queue of integers

```
MODEL
  ESM ResourceAlloc;
    TYPE
      Request = {Get,Release};
      Result = {OK};
    VAR
      Req: Request;
      Res: Result;

    ESM A(
      OUT Req: Request;
      IN Res: Result);
    BEGIN
      DO TRUE ->
        Req!Get;
        Res?OK;
        Req!Release
      OD
    END A;

    ESM B(
      OUT Req: Request;
      IN Res: Result);
    BEGIN
      DO TRUE ->
        Req!Get;
        Res?OK;
        Req!Release
      OD
    END B;

    ESM Resource(
      OUT Res: Result;
      IN Req: Request);
      VAR InUse: BOOLEAN;
    BEGIN
      InUse:= FALSE;
      DO TRUE ->
        DO ~InUse /\ Req?Get ->
          InUse:= TRUE; Res!OK
        [] InUse /\ Req?Release ->
          InUse:= FALSE
        OD
      OD
    END Resource;

  BEGIN
    A(Req,Res);
    B(Req,Res);
    Resource(Res,Req)
  END ResourceAlloc;

  ASSERT
    AG((Req = Get) => AF(Res = OK))
END
```

Figure 2. Resource allocation

This deceivingly simple model contains an error. Assume that the resource has been allocated to *B* (an *OK* signal is present on channel *Res*). *A* now requests the resource (*Get* signal on channel *Req*). *B tries* to release the resource (*Release* signal on channel *Req*). However, *A* is already waiting (blocked) on the *Req* channel with a *Get* signal. Thus *B* is also blocked on the *Req* channel; *Resource* also cannot proceed (*InUse* is TRUE and no *Release* signal is available on channel *Req*). The requirement is thus violated.

The problem is that the *Resource* server as shown in Figure 2 ignores a get request until the resource is not in use. One way to eliminate this problem would be to change ESM *Resource* to return a signal *NotOK* to any *Get* request while the resource is in use.

5 Final Remarks

An efficient model checker capable of analysing models of non-trivial size has been developed. Models acceptable to the model checker are state transition systems. To make the system easy to use we designed a validation language, called ESML, to model reactive systems. Validation models written in ESML can be translated to equivalent state transition systems which can be accepted by the model checker. ESML is based on extended state machines which incorporate complex data structures and communication over channels.

We are currently developing a translator for ESML. The validation system is written in Modula-2. The system has been designed to be portable—it was developed on a personal computer running MSDOS and moved to a Unix workstation simply by recompiling it. A personal computer is adequate for small applications but the model checker needs a more powerful machine (more memory) to handle models of realistic size.

As a representative example of the kind of system we wish to validate we chose a small kernel which supports a variable number of processes and interprocess communication. The design of ESML has been influenced by modelling various fragments of this kernel. The size of the state vector places a limit on the size of problem which can be handled on currently available hardware and more work remains to be done to evaluate the suitability of ESML as a validation language for reactive systems.

References

1. P. Brinch Hansen, 'Joyce—A Programming Language for Distributed Systems', *Software—Practice and Experience*, vol. 17, no. 1, pp. 29–50, January 1987.
2. E. M. Clarke and O. Grumberg, 'Research on Automatic Verification of Finite-State Concurrent Systems', Tech. Rep. CMU-CS-87-105, Carnegie-Mellon University, January 1987.
3. E. Clarke and E. Emerson, 'Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic', in *Proceedings of IBM Workshop on Logic of Programs*, (D. Kozen, ed.), pp. 52–71, Lecture Notes in Computer Science, 131, 1981.
4. E. Clarke, E. Emerson, and A. Sistla, 'Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach', *Proceedings 10th ACM Symposium on Principles of Programming Languages*, pp. 117–126, 1983.
5. P. de Villiers, 'A Model Checker for Transition Systems', in *6-th Southern African Computer Symposium*, (M. Linck, ed.), pp. 262–275, July 1991.
6. E. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, New Jersey: Prentice-Hall, 1976.
7. W. Fouché and P. de Villiers, 'A Reusable Kernel for the Development of Control Software', in *6-th Southern African Computer Symposium*, (M. Linck, ed.), pp. 83–94, July 1991.
8. P. Godefroid and P. Wolper, 'A Partial Approach to Model Checking', in *6-th IEEE Symposium on Logic in Computer Science*, (Amsterdam), pp. 406–414, 15-18 July 1991.
9. C. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, New Jersey: Prentice-Hall, 1985.
10. G. Holzmann, 'An Improved Reachability Analysis Technique', *Software Practice and Experience*, vol. 18, no. 2, pp. 137–161, February 1988.
11. G. Holzmann, *Design and Validation of Computer Protocols*. Englewood Cliffs, New Jersey: Prentice-Hall, 1991.
12. G. Holzmann, 'Protocol Design: Redefining the State of the Art', *IEEE Software*, vol. 9, no. 1, pp. 17–22, January 1992.
13. G. Holzmann, P. Godefroid, and D. Pirottin, 'Coverage Preserving Reduction Strategies for Reachability Analysis', Unpublished Draft.
14. C. B. Jones, *Systematic Software Development Using VDM*. Prentice-Hall, 2 ed., 1990.
15. Z. Manna and A. Pnueli, 'Verification of concurrent programs, part I: the temporal framework.', Tech. Rep. STAN-CS-81-836, Dept of Computer Science, Stanford University, July 1981.
16. Z. Manna and A. Pnueli, 'Completing the Temporal Picture', Research Report STAN-CS-89-1296, Department of Computer Science, Stanford, California 94305, December 1989.
17. J. Ostroff, *Temporal Logic for Real-Time Systems*. New York: John Wiley and Sons Inc., 1989.
18. A. Pnueli, 'The temporal semantics of concurrent programs.', *Theoretical Computer Science*, vol. 13, pp. 45–60, 1981.
19. J. M. Spivey, *The Z Notation: A Reference Manual. International Series in Computer Science*, Prentice-Hall, 1989.
20. A. Valmari and M. Tienari, 'An Improved Failures Equivalence for Finite-State Systems with a Reduction Algorithm', in *11-th International Symposium on Protocol Specification, Testing, and Verification*, (Stockholm), pp. 1–16, IFIP WG6.1, June 17-20 1991.

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for *review* should be prepared according to the following guidelines.

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets [1, 2, 3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) LaTeX file(s), either on a diskette, or via e-mail/ftp – a LaTeX style file is available from the production editor;
2. In camera-ready format – a detailed page specification is available from the production editor;
3. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

   Further instructions on how to reduce page charges can be obtained from the production editor.
4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for LaTeX or camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 3 and 4 above will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

1. E Ashcroft and Z Manna. 'The translation of 'goto' programs to 'while' programs'. In *Proceedings of IFIP Congress 71*, pp. 250–255, Amsterdam, (1972). North-Holland.
2. C Bohm and G Jacopini. 'Flow diagrams, turing machines and languages with only two formation rules'. *Communications of the ACM*, 9:366–371, (1966).
3. S Ginsburg. *Mathematical theory of context free languages*. McGraw Hill, New York, 1966.

South African Computer Journal

Number 7, July 1992
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 7, Julie 1992
ISSN 1015-7999

Contents