A Comparison of Three Model Checkers Applied to a Distributed Database Problem

Andrew J Currie

Department of Electronics and Computer Science
University of Southampton, UK

1 Abstract

Increasingly, model checking is being applied to more abstract problem domains than the traditional protocol analysis. The extent to which such an approach is able to provide useful insight into the problem domain depends to a large degree on the nature of the tool used. This paper reports the results of a study using three different model checkers, which differ widely in their specification language, internal implementation, and facilities for specifying correctness properties.

An abstract model of an industrial distributed database application has been studied using the three tools. A detailed model of the application using each of the tools is presented, and the extent to which each tool allows us to investigate interesting properties of the problem domain is compared. Some conclusions are drawn regarding the usefulness of model checking at an abstract level and the importance of selecting a tool appropriate to the nature of the problem.

2 Introduction

Model checking is now widely accepted as a valuable tool for the analysis of many problems which can be abstracted by a finite state model. Most papers have either reported on the application of a specific model checker to a specific problem area, or have focussed on techniques for model checking ever larger problems. In contrast, in this paper we look at the application of three popular and widely used model checkers to the same abstract problem, and focus on their expressive power and usability rather than on problem size.

The problem considered is based on a real industrial application supplied by one of our industrial partners [4]. A distributed data base system is used to serve customers such that each service (a payment) can only be provided to one particular customer, and only once. The system has the following components:

- A single *Centre*, where most of the data is held;
- A number of *Offices*, where relevant parts of the data are held;
- Many *Customers*. Each customer has a home office which will normally hold the data pertaining to that customer. A foreign office should be able to provide a customer with the same services as the home office, after consultation with the Centre.

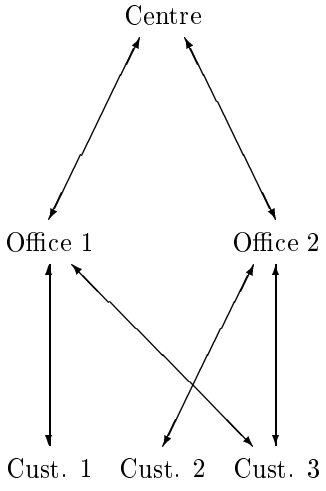

Figure 1: Graphical representation of the model, showing the centre, two offices and three customers.

The basic model of this system was kept as simple as possible. It is shown in graphical form

in Figure 1. This identifies the centre, two offices and three customers. Data for Customers 1 and 2 is held at their respective home offices 1 and 2. Customer 3 does not have a home office (in this model); their data is held by the centre.

All three customers will attempt to obtain payment from one or other of the offices. Customer 1 will be successful when contacting office 1, but unsuccessful when contacting office 2; similarly for customer 2. An office which does not hold the appropriate data passes the request on to the centre. Customer 3 should be successful in obtaining payment regardless of which office they visit. Once a customer has been served, successfully or not, they are satisfied and do not request further service.

Formal models of this system were built using three different formalisms, as detailed below. In each case, various properties both of the basic model and a number of variants were investigated. The properties considered were as follows:

1. The correctness of the basic model:
   (a) All customers should eventually be served.
   (b) Customer 3 will always be served successfully.
   (c) No customer can visit two offices at the same time.
2. The result of introducing an error corresponding to missing functionality—the model of the centre was modified so that it failed to reply to a query if it had no corresponding customer record in its database. This leads to deadlock, and property 1(a) above no longer holds.
3. A variant of the basic model was produced by changing the behavior of the customers so that they would repeatedly try to obtain service until they were successful. This can lead to livelock/starvation, and properties 1(a) and 1(b) no longer hold.
4. A refinement of the model in 3 above can be constructed by replacing the synchronous interactions between customer and office and between office and centre with queues. Although not free of livelock (a customer can repeatedly go to the wrong office indefinitely), provided the queues are long enough it is free of starvation, and properties 1(a) and 1(b) hold.

(Note: In the Murϕ model, as in this discussion, the customers are numbered from 1 to 3; however in the other two models they are numbered from 0 to 2).

3 The Tools

The three model checkers used were Spin [7, 12], Murϕ [2, 8] and FDR2 [9, 3]. All three were used on a Pentium processor running Linux and X-Windows.

3.1 Spin

Spin is a model checker originally designed for validating communications protocols [6]. Its modelling language, Promela, supports dynamic creation of concurrent processes and both synchronous and asynchronous message passing, as well as shared variables. Facilities for data abstraction are very limited, only bits, bytes, ints and arrays being provided. The syntax of Promela is loosely based on that of C, with nondeterministic choice and iteration constructs similar to those in Dijkstra's guarded command language [1].

Spin allows specification of correctness properties in a number of different forms. Assertions may be embedded in the model, and properties may be specified by means of a linear-time temporal logic (LTL) formula. In addition Spin will check for invalid end states and acceptance or non-progress cycles, with or without a weak fairness assumption.

Spin can perform random or interactive simulations, and can perform an exhaustive state space verification of both safety and liveness properties. Xspin is a user-friendly graphical front-end to Spin, which greatly enhances its usability. This provides a window-based, menu-driven interface to Spin and allows animation of simulations and error traces in a variety of different ways.

3.2 Murϕ

Murϕ is another model checker, also originally designed for communications protocols. A Murϕ description consists of a set of transition rules comprising a condition and an action; the execution model involves repeatedly making a nondeterministic selection of a rule whose guard is enabled and executing the corresponding action. All interactions take place by means of shared

variables. Its syntax is based on that of Pascal, and it shares Pascal's fairly good (if rather low-level) data structuring facilities.

Murϕ can only check safety properties, as there is no way of specifying liveness properties. Assertions may be associated with specific rules or defined as invariants. Invalid end states are also detected.

Murϕ can perform simulation or exhaustive verification, although only deadlock and assertion violations are detected. Error reporting is by means of a textual trace of rules fired which must be interpreted by hand.

3.3 FDR2

FDR2 is a model checker for CSP [5]. Its modelling language is a machine-readable form of CSP [11] augmented by a rich set of data abstraction facilities. Sets, sequences, tuples and compound types are all provided, together with a powerful pattern-matching facility.

FDR2 is unusual in that its method of establishing whether a property holds of a given system is based on the notion of refinement: a specification which captures the property of interest is written, in the same notation as is used to define the system, and the tool checks that the system is a refinement (in the CSP sense [10]) of the specification. It is also possible to check a system for determinism and for possible deadlock or livelock. No conventional simulation facilities are provided.

FDR2 is a window-based program which allows the user to select a process (or pair of processes in the case of the refinement check) and a check to be performed. If the check fails, there is a sophisticated debugger which allows the system structure to be navigated in a hierarchical manner and the behavior which led to the error to be observed.

4 The Spin Model

Below is the basic model written in Promela, the modelling language used by Spin. Firstly the database is modelled as a pair of arrays indexed by customer number - a boolean to record whether or not that customer's information is present, and an integer specifying the amount due. The variables `pmc`, `pm` and `ac` represent the centre's database, the offices' databases and the customers' personal accounts respectively. All are initialized to zero; the interesting values are set up by the `init` process (below).

```
#define NoOFCUSTOMERS 3
#define NoOFOFFICES   2

typedef Database {
      bool present[NoOFCUSTOMERS];
      byte amount[NoOFCUSTOMERS]
}

Database pmc = 0;
Database pm[NoOFOFFICES] = 0;
byte ac[NoOFCUSTOMERS] = 0;
```

4.1 The Centre

The centre is modelled as a server process `Centre`, which sits in an infinite loop waiting for a request on its input channel `s`. This loop is marked with the label `end` to indicate that it is a valid end state; i.e. that it is not an error for the process to remain blocked at this point indefinitely.

The request (from one of the offices) consists of a customer number `b` and a reply channel `r`. If information about customer `b` is present in the centre's database, the amount due to that customer is sent on the reply channel and the customer is removed from the database. If the customer is not known to the centre the value zero is returned.

```
proctype Centre (chan s)
{     byte b;  chan r;
 end: do
      ::  s?b,r;
          if
          ::  pmc.present[b]  ->
                    r!pmc.amount[b];
                    pmc.present[b] = false;
          ::  !pmc.present[b]  ->
                    r!0;
          fi;
      od;
}
```

4.2 The Offices

Each office is modelled by a server process (`Office` below) in the same way as the centre. An individual office is distinguished by its identifier `n`, and uses the channels `r` and `s` to receive requests from customers and to send queries to the centre, respectively. The request consists of a customer number `b` and a reply channel `c`. If the

office knows about customer b, it sends the specified amount on the reply channel and removes the customer from its database; otherwise it queries the centre, and passes on the reply from the centre to the customer, using an unbuffered channel t to communicate with the centre.

```
proctype Office (byte n; chan s, r)
{       byte b, x;  chan c;
        chan t = [0] of {byte};
 end:   do
        ::  r?b,c;
            if
            ::  pm[n].present[b]  ->
                    c!pm[n].amount[b];
                    pm[n].present[b] = false;
            ::  !pm[n].present[b]  ->
                    s!b,t;  t?x;  c!x;
            fi;
        od;
}
```

4.3 The Customers

Each customer is modelled by a process (`Customer` below) whose parameters are an identification number `n` and two channels to allow it to communicate with either of the two offices. The customer makes a nondeterministic choice between the offices, engages in a transaction with the chosen office by sending its customer number `n` and the name of a reply channel `c` on the appropriate channel, and waits for a reply. The amount received is stored in the customer's account (simply to make it globally visible during simulation), and the customer process then terminates.

```
proctype Customer (byte n; chan t0, t1)
{       chan  c = [0] of {byte};
        byte  x = 0;
        if
        ::  true  ->  t0!n,c;  c?x;
        ::  true  ->  t1!n,c;  c?x;
        fi;
        ac[n] = x;
}
```

4.4 Initialization of the processes

The `init` process is the first one to be run, and sets up the model. First it initializes the databases so that office 0 knows about customer 0, office 1 knows about customer 1, and the centre knows about customer 2. It then starts six concurrent processes representing the centre, two offices and three customers in the configuration shown in Figure 1.

```
init
{       chan  s = [0] of {byte, chan};
        chan  t0 = [0] of {byte, chan};
        chan  t1 = [0] of {byte, chan};

        atomic {
            pmc.present[2]   = true;
            pmc.amount[2]    = 42;
            pm[0].present[0] = true;
            pm[0].amount[0]  = 42;
            pm[1].present[1] = true;
            pm[1].amount[1]  = 42;

            run Centre (s);
            run Office (0, s, t0);
            run Office (1, s, t1);
            run Customer (0, t0, t1);
            run Customer (1, t0, t1);
            run Customer (2, t0, t1);
        }
}
```

5 Analysis of the Spin model

Using Xspin makes Spin very easy to use. The Promela source code is displayed in a window, and can be edited, checked and debugged in a nicely integrated way. The various options and facilities are selected by means of menus and check boxes. The actual analysis is done by invoking Spin (a command-line program) as a background process, which analyzes the Promela description and generates a C program. This is then compiled and executed to perform the state-space search. If an error is found an error trace is produced, which can then be used to guide a simulation to reconstruct the execution sequence that led to the error. All this is handled by Xspin, and the user need not be aware of the detailed operation of the tool.

The model was investigated as described in Section 2 above; the results were as follows:

1. (a) All customers should eventually be served: this is readily established by performing checks for invalid end states (a safety property—absence of deadlock) and for acceptance cycles (a liveness property).

   (b) Customer 3 will always be served successfully: this can be shown by specifying a constraint by means of an LTL formula: $\Diamond p$, where $\Diamond$ is the operator *eventually* and p is defined as `(ac[2] == 42)`.

(c) No customer can visit two offices at the same time. This is rather trickier to specify, as the interactions between customers and offices are modelled by synchronous communications rather than by means of global variables. It can of course be shown to be valid by a static analysis of the Promela code; using Spin, the simplest way is to modify the code for the `Office` process to make globally visible the number of the customer currently being served, and to add in `Office` an assertion that the customer numbers for the two offices are different. This can then be established by running a check for assertion violations. Alternatively this could be specified as an LTL formula using the *always* operator.

2. Missing functionality in the Centre: this is readily detected by the invalid end states check. To determine the cause of the error is straightforward using the guided simulation facilities. The most useful facility in this context is the (graphical) message sequence chart, which shows the communications between processes. This abstracts away from the internal process details and makes it simple to pinpoint where the error occurred.

3. Customers repeatedly try to obtain service until they are successful. The check for acceptance cycles finds the possibility of livelock; customers 1 and 2 can repeatedly visit the wrong office, and this can result in customer 3 never being served. Checking the LTL formula $\Diamond p$ as above also fails, for the same reason. Again, the message sequence chart makes it easy to determine what is going wrong.

4. Introduction of queueing. As buffered communications are provided as primitive operations in Promela, adding queueing to the model is simple. This does not eliminate the possibility of livelock, so an acceptance cycle is still found, but $\Diamond p$ can now be shown to hold.

6 The Murϕ model

Below is the basic model in Murϕ. The description consists of a series of constant, type and variable declarations, followed by a set of transition rules which define the behavior of the system. Each rule is a guarded command, consisting of a condition (a Boolean expression on the global variables) and an action. The execution model is to repeatedly evaluate the conditions, choose (nondeterministically) one rule whose condition evaluates to true, and execute the corresponding action. The execution of the actions is atomic — all interleaving of executions is specified explicitly by breaking a process down into a set of rules.

The first section declares some constants, types and variables. The database is modelled as an array of records, where each record has two fields: a boolean to record whether or not that customer's information is present, and an integer specifying the amount due. The variables `pmc` and `pm` represent the centre's database and the offices' databases respectively. `CustStat`, `OffStat` and `CentreStat` are state variables for the customers, offices and centre respectively. The remaining variables are used to model the interactions between customers and offices, and between the offices and the centre.

```
const NoOfCustomers:  3;
      NoOfOffices:    2;

      NoOne:          0;

type  CustRng:      NoOne..NoOfCustomers;
      RealCustRng:  1..NoOfCustomers;
      OffRng:       1..NoOfOffices;

      CustType:   enum {Pennyless, Waiting,
                                    Done};
      CentreType: enum {Ready, Busy};
      OfficeType: enum {Free, Serving,
                        Querying, Awaiting};

      Data:       record
                     present:  boolean;
                     amount:   0 .. 255;
                  end;
      Database:   array [CustRng] of Data;

var   pmc:  Database;
      pm:   array [OffRng] of Database;

      CustStat:   array [RealCustRng] of
                                   CustType;
      OffStat:    array [OffRng] of
                                  OfficeType;
      CentreStat: CentreType;
      AtOffice:   array [OffRng] of CustRng;
      MoneyRecd:  Database;
      CentreReq:  CustRng;
```

```
    CentreResp: Data;

    CustAck:   array [CustRng] of boolean;
    CentreAck:  boolean;

    dummy:      boolean;
```

6.1 The Centre

The centre is modelled by a pair of rules and a two-valued state variable: `CentreStat` is either `Ready` (to accept a query from an office) or `Busy`. If it is `Ready` and there is a request pending, the rule `"Accept query"` is enabled; the corresponding action acknowledges the request, changes the status to `Busy` and responds appropriately. If the status is `Busy`, the centre must wait until the whole transaction has been acknowledged by the office which requested it; the centre then returns to `Ready` status.

```
rule "Accept query"
    CentreStat = Ready &
    CentreReq != NoOne
==>
    CentreAck  := true;
    CentreStat := Busy;
    if pmc[CentreReq].present then
        CentreResp := pmc[CentreReq];
        pmc[CentreReq].present := false;
    else
        CentreResp.amount  := 0;
        CentreResp.present := true;
    end;
end;

rule "Wait for Office Acks"
    CentreStat = Busy &
    !CentreResp.present &
    !CentreAck
==>
    CentreReq  := NoOne;
    CentreStat := Ready;
end;
```

6.2 The Offices

Each office is modelled by a set of four rules and a four-valued state variable: `status` is either `Free`, `Serving` (engaged in a transaction with a customer), `Querying` (the centre), or `Awaiting` (a reply from the centre). The `ruleset` construction replicates the set of rules for each possible value of the parameter `off` (in this case 1 or 2).

```
ruleset off: OffRng do
  alias c: AtOffice[off];
    status: OffStat[off] do

    rule "Accept Customer"
        status = Free &
        c != NoOne
    ==>
        if pm[off][c].present then
           status             := Serving;
           MoneyRecd[c]       := pm[off][c];
           pm[off][c].present := false;
        else
           status             := Querying;
        end;
    end;

    rule "Await customer ack"
        status = Serving &
        CustAck[c]
    ==>
        CustAck[c]    := false;
        AtOffice[off] := NoOne;
        status        := Free;
    end;

    rule "Contact Centre"
        status = Querying &
        CentreReq = NoOne
    ==>
        CentreReq := c;
        status    := Awaiting;
    end;

    rule "Await centre's reply"
        status = Awaiting &
        CentreAck &
        CentreResp.present
    ==>
        CentreAck          := false;
        MoneyRecd[c]       := CentreResp;
        CentreResp.present := false;
        CentreResp.amount  := 0;
        status             := Serving;
    end;

  end;
end;
```

6.3 The Customers

Each customer is modelled by a pair of rules and a three-valued state variable: `CustStat[c]` is either `Pennyless` (the initial value), `Waiting` (engaged in a transaction with an office) or `Done` (having completed a transaction). Once a customer's status becomes `Done` it can no longer engage in any transactions (i.e. it effectively terminates).

```
ruleset c: RealCustRng do

  ruleset off: OffRng do
      rule "Go to an office"
          CustStat[c]   = Pennyless &
          AtOffice[off] = NoOne
      ==>
          AtOffice[off] := c;
          CustStat[c]   := Waiting;
      end;
  end;

  rule "Complete transaction"
      CustStat[c] = Waiting &
      MoneyRecd[c].present
  ==>
      CustAck[c]  := true;
      CustStat[c] := Done;
  end;

end;
```

6.4 Detection of valid end states and avoiding deadlocks

Murϕ assumes that system execution is infinite; it does not distinguish between valid and invalid end states. It therefore always reports an error in a client/server model where all the clients have terminated, and is unable to differentiate this case from a genuine deadlock. To circumvent this problem, we introduce an additional rule which detects the valid end state, and avoids deadlock by repeatedly changing the value of a dummy variable. (Murϕ's definition of deadlock is that the current state has no successor other than itself.) This allows genuine deadlocks to be detected and reported correctly. We also use this rule to introduce an assertion regarding a property of the system - customer 3, whose details are held at the centre, will successfully complete his transaction regardless of which office he goes to.

```
rule "Reached valid end state"
  forall c: RealCustRng do
      CustStat[c] = Done & !CustAck[c]
  end  &
  forall off: OffRng do
      OffStat[off] = Free &
      AtOffice[off] = NoOne
  end  &
  CentreStat = Ready & CentreReq = NoOne &
  !CentreAck & !CentreResp.present
==>
  assert
    MoneyRecd[3].amount = 42
    "Customer 3 completed transaction OK";
  dummy := !dummy;  --  Avoid deadlock
end;
```

6.5 Correctness properties

In addition to associating assertions with particular rules, it is possible to specify properties which are globally true, i.e. invariants. As an example, we include a specification that no customer can be at more than one office at the same time.

```
invariant
  "No customer at >1 office simultaneously"
    forall o1: OfficeRange do
        AtOffice[o1] = NoOne |
        forall o2: OfficeRange do
            o1 = o2 |
            AtOffice[o1] != AtOffice[o2]
        end
    end;
```

6.6 The initial state of the system

The startstate describes the initial state of the system. All the variables are initialized by the `clear` command to the lowest values of their type, and the databases are set up so that office 1 knows about customer 1, office 2 knows about customer 2, and the centre knows about customer 3.

```
startstate

  clear pmc;  clear pm;
  pmc[3].present   := true;
  pmc[3].amount    := 42;
  pm[1][1].present := true;
  pm[1][1].amount  := 42;
  pm[2][2].present := true;
  pm[2][2].amount  := 42;

  clear CustStat;
  clear OffStat;
  clear CentreStat;
  clear AtOffice;
  clear MoneyRecd;
  clear CentreReq;
  clear CentreResp;
  clear CustAck;
  clear CentreAck;
  clear dummy;
end
```

7 Analysis of the Murϕ model

Murϕ is a text-only, command-line program, but straightforward to use. Running Murϕ on the file containing the system description causes a C++ program to be generated, which then has to be compiled and run in order to do the analysis. The command line arguments given to this program determine whether a simulation or an exhaustive state-space search is performed, and control the amount of debugging information produced.

Running the Murϕ verifier on the above model allows us to establish the safety properties, as follows:

1. (a) All customers should eventually be served: this is established by the fact that the valid end state is always reached.

   (b) Customer 3 will always be served successfully: this is established by the assertion in the end state.

   (c) No customer can visit two offices at the same time: this is established by the invariant specified as part of the model.

2. Missing functionality in the Centre. If we introduce the missing centre functionality error, an error trace is produced, allowing the user to determine (albeit with some difficulty) where the error occurred. The trace consists of the sequence of rules that were fired to reach the state in which the error was detected. For each rule, the values of the state variables that changed as a consequence of executing that rule are printed, except for the initial and final states, for which the values of all the state variables are printed. By working backwards through the trace the sequence of events which led to this situation can be determined, and hence the nature of the error can be deduced. Although this method of debugging is perfectly feasible, it is difficult and time-consuming, especially in more complex cases.

As Murϕ can only determine safety properties and has no way of reasoning about liveness, nothing was learned from investigating the variants of the basic model described in Section 2.

8 The FDR2 model

Below is the basic model written in the machine-readable form of CSP used by FDR2. The database is modelled as a set of pairs mapping customers to amounts. The sets `pmc`, `pm0` and `pm1` represent the databases of the centre and the two offices respectively. Interaction between customers, offices and the centre is modelled by CSP compound events, declared in FDR2 as channels of a particular type. The auxiliary function `amount(r)` extracts the amount from a set containing a (customer, amount) pair using FDR2's pattern matching facility.

```
NoOfCustomers = 3
NoOfOffices   = 2
CUSTOMERS = {0 .. NoOfCustomers - 1}
OFFICES    = {0 .. NoOfOffices - 1}

Payment = 42
AMOUNT = {0, Payment}

channel centrein: OFFICES.CUSTOMERS
channel centreout: OFFICES.AMOUNT

channel tooffice: OFFICES.CUSTOMERS
channel getpayment: OFFICES.CUSTOMERS.AMOUNT

channel result: CUSTOMERS.AMOUNT

pmc = {(2, Payment)}
pm0 = {(0, Payment)}
pm1 = {(1, Payment)}

amount(r) = let {(o, a)} = r within a
```

8.1 The Centre

The centre is modelled as a process which waits for an event on the channel `centrein`, corresponding to an enquiry from office `off` concerning customer `cust`. It then extracts the record corresponding to `cust` from its database; if such a record is present it notifies the corresponding amount to the enquiring office and removes the corresponding record from its database; otherwise it sends zero.

```
CENTRE(d) =
  centrein?off?cust ->
    let
      rec = {(c, a) | (c, a) <- d, c == cust}
    within
      if empty(rec) then
        centreout.off!0 -> CENTRE(d)
```

```
    else
      centreout.off!amount(rec) ->
                        CENTRE(diff(d, rec))
```

8.2 The Offices

Each office waits for an event on the appropriate `tooffice` channel, corresponding to a request from a customer. If a record for this customer is present in its database, it sends the corresponding amount to the customer and removes the record from the database; otherwise it queries the centre, waits for a reply on the appropriate `centreout` channel, and passes the reply on to the customer.

```
OFFICE(d, n) =
  tooffice.n?cust ->
    let
      rec = {(c, a) | (c, a) <- d, c == cust}
    within
      if empty(rec) then
        centrein.n!cust -> centreout.n?p ->
          getpayment.n.cust!p -> OFFICE(d, n)
      else
          getpayment.n.cust!amount(rec) ->
                      OFFICE(diff(d, rec), n)
```

8.3 The Customers

Each customer makes a nondeterministic choice from the set `OFFICES`, requests payment from that office, waits for a reply, notifies the result on the `result` channel (to make it globally visible), and then terminates.

```
CUSTOMER(n) =
  |~| off: OFFICES @ (tooffice.off!n ->
                        getpayment.off.n?p ->
                          result.n!p -> SKIP)
```

8.4 The System and the Specification

The system model is a parallel composition of centre, offices and customers. Two `OFFICE`s in parallel interact with the `CENTRE` via the channels `centrein` and `centreout`; This subsystem in turn interacts with the `CUSTOMER` processes via `tooffice` and `getpayment`. The internal communications on `centrein`, `centreout`, `tooffice` and `getpayment` are hidden.

The two specifications `SPEC` and `SPEC2` encapsulate behaviors that we want to check that `SYSTEM` exhibits. The first specifies that each customer will eventually receive a response from the office, while the second adds the additional constraint that customer 2 will always be successful in obtaining payment.

```
SYSTEM =
  ((OFFICE(pm0, 0) ||| OFFICE(pm1,1))
       [|{|centrein, centreout|}|]
    CENTRE(pmc))
       [|{|tooffice, getpayment|}|]
    (||| c: CUSTOMERS @ CUSTOMER(c))
  \ {|centrein,centreout,tooffice,getpayment|}


SPEC = ||| c: CUSTOMERS @ result.c?p -> STOP

SPEC2 =
  (||| c: {0 .. 1} @ result.c?p -> STOP) |||
   result.2.Payment -> STOP
```

9 Analysis of the FDR2 model

Running FDR2 is fairly straightforward, although it takes longer to learn to use it effectively than for Xspin. Although the CSP source code is not displayed by FDR2, it is possible to invoke an editor from within FDR2 which can be used to make changes, and then to reload the changed file. As properties of the system are specified in FDR2 by writing a CSP specification that captures the property in question, this has to be done quite frequently. The debugging facilities in the event of an error being discovered are powerful but not as intuitive and easy to use as those of Xspin.

The model was investigated as described in Section 2 above; the results were as follows:

1. (a) All customers should eventually be served: this is readily established by showing that the system (SYSTEM) is a refinement of the specification SPEC, which encapsulates this property.

   (b) Customer 3 will always be served successfully: this is established by showing that SYSTEM is a refinement of SPEC2. (In fact it is equivalent to SPEC2, which can be established by showing that SPEC2 is also a refinement of SYSTEM).

   (c) No customer can visit two offices at the same time. Although this property

can easily be established by an analysis of the communication structure of SYSTEM, it is difficult to prove it using FDR2. It is necessary to add extra (globally visible) events to OFFICE to indicate the "arrival" of a customer, and then to write a specification which allows all possible legal combinations of arrival and result events. This is not too difficult for the basic model, but may need to be changed substantially whenever the model is changed.

2. Missing functionality in the Centre: this is readily detected as SYSTEM is no longer a refinement of SPEC. The debugging facility allows the process traces leading to the error to be investigated, and the cause of the error to be pinpointed.

3. Customers repeatedly try to obtain service until they are successful. Again, the refinement checks fail, this time owing to the presence of livelock (called divergence in CSP parlance), and the debugging facilities pinpoint exactly how and where this occurs.

4. Introduction of queueing. This involves adding explicit buffer processes to the model, which makes it substantially more complex. It was possible to some extent to investigate the properties of this extended model, but because livelock is still present, the properties which could be established are quite limited. According to the semantics of divergence in CSP, processes are considered identical after they have diverged, so it is often not possible to distinguish other process behaviors in the presence of divergence. In particular, it was not possible to show that property 1(b) holds of the extended model, not even by relaxing the specification (SPEC2) to allow the possibility of divergence.

10 Conclusions

Applying the three tools to the same, fairly simple, problem has produced some interesting results. The three models described in detail above are quite different from one another, and each required different skills and thought processes during development. This was even more pronounced when variants of the basic model were constructed. Similarly, effective use of the tools entailed a different approach and philosophy in each case.

Writing the descriptions in both CSP and Promela was natural and fairly straightforward, because of the nature of the distributed database/ client-server problem. In contrast the Murϕ description was much longer and more difficult to write, because the nature of the problem does not map well onto the modelling paradigm provided by Murϕ. The Murϕ model is far removed from the user's mental model of the problem; it is necessary to make explicit the points at which non-deterministic choice and interactions between processes occur, and to provide explicit handshaking for each synchronous interaction. As a result, every sort of structure in the mental model has to be unraveled in order to express it in Murϕ, and reported errors have first to be tracked down in terms of the model, then related back to the original problem. This is very much a characteristic of the particular application studied; it is likely that for problems which are naturally described by a set of rules, Murϕ would be natural and appropriate.

Comparing CSP and Promela, CSP is more elegant and has a much more rigorously defined semantics, and the data abstraction facilities provided by FDR2 are very much better than those of Promela. However Promela is richer and strictly more expressive (e.g. asynchronous communication is supported, and channels are first class objects in Promela but not in CSP). Furthermore, Promela's C-like syntax makes it more accessible to non-experts.

None of the three tools was particularly difficult to use. Both Spin and FDR2 benefit greatly from having a graphical front end, as the facilities they provide are complex. Murϕ is much simpler, so the textual command-line-based approach is quite adequate. The fact remains though that debugging using the Murϕ error trace is quite hard work. Xspin is probably the easiest to use, being both powerful and intuitive. FDR2 is initially rather harder to understand, but once the learning curve has been overcome it provides an equally powerful and productive debugging environment.

As previously discussed, Murϕ is significantly less powerful than the other two tools as it cannot deal with liveness properties. The contrast between the refinement approach to specifying system properties of FDR2 and the more traditional temporal logic approach of Spin is particularly interesting. It is possible using temporal

logic formulae to specify concisely many properties which cannot easily (or in some cases at all) be expressed as a refinement relationship. On the other hand, using refinement allows a number of properties to be captured and checked simultaneously; and more importantly, it allows the stepwise construction of more and more detailed models while ensuring that essential properties are preserved.

The overall conclusion is that any model checker can be of value in investigating an abstract model of a problem; but selecting one which is appropriate to the nature of the problem and the properties of interest can greatly increase its effectiveness.

References

[1] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[2] D. L. Dill, A. J. Dexler, A. J. Hu and C. H. Yan. Protocol verification as a hardware design aid. In *Computer Design: VLSI in Computers and Processors*, 522–525, IEEE Computer Society Press, 1992.

[3] FDR2: **http://www.formal.demon.co.uk/FDR2.html**

[4] P. H. Hartel, M. J. Butler, A. J. Currie, P. Henderson, M. A. Leuschel, A. P. Martin, U. Ultes-Nitsche, R. J. Walters. Questions and Answers about Ten Formal Methods. *Proc. 4th International ERCIM Workshop on Formal Methods for Industrial Critical Systems*, 179–203, Trento, Italy, July 1999.

[5] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[6] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[7] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[8] Murϕ: **http://sprout.stanford.edu/dill/murphi.html**

[9] A. W. Roscoe. Model-checking CSP. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Prentice-Hall, 1994.

[10] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[11] J. B. Scattergood. *Tools for CSP and Timed CSP.* D.Phil Thesis, Oxford University Computing Laboratory, September 1995.

[12] Spin: **http://netlib.bell-labs.com/netlib/spin/whatispin.html**