Verifying an infinite family of inductions simultaneously using data independence and FDR*

S. J. Creese and A. W. Roscoe

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, UK
{Sadie.Creese, Bill.Roscoe}@comlab.ox.ac.uk

March 7, 1999

Abstract

We present a technique for formally establishing results for scalable systems, such as distributed systems and communication protocol networks, where the results are independent of the system's parameters. Example parameters are network topology, size and buffer capacity. The technique combines the use of the process algebra CSP to model systems and their specifications, and the FDR tool to help reason about them. We give examples of the techniques implementation on a simple distributed system and a communications protocol involving the multiplexing of channels.

Keywords: process algebra, verification, communication protocols, induction, data-independence, model checking.

1 Introduction

An important current area of research in the verification of distributed systems is scalability: how do we handle the verification of realistic sized sys-

*The work reported in this paper was supported by DERA Malvern and the US Office of Naval Research

tems, and families of system under parameterisations that lead to unboundedly large state spaces? Two of the best known approaches to this problem are induction and data independence. This paper explores the interface between the two – specifically, by developing a proof method which combines them. What we actually show is how data independence can allow a finite automated proof which establishes an infinite number of inductions simultaneously.

The technique is implemented using the process algebra CSP[9, 14] to model systems, and the FDR model checker tool [7] to help reason about them, though we have no doubt that our ideas would work in other contexts as well. CSP models a system as a *process* which interacts with its environment by means of atomic *events*. A series of semantic models facilitate the capture of a wide range of behaviours including safety and liveness properties. The theory of refinement in CSP allows correctness conditions to be en-coded as refinement checks between processes. If A $\sqsubseteq$ B is true (where $\sqsubseteq$ represents refinement) then the behaviours of B are contained within the behaviours of A. In particular, refinement is transitive; if P $\sqsubseteq$ Q and Q $\sqsubseteq$ R this implies that P $\sqsubseteq$ R.

FDR takes a machine-readable dialect of CSP as its input syntax[1], and can be used to check refinements as well as determinism, deadlock freedom and livelock freedom of processes. FDR performs a check by invoking a normalisation procedure for the specification process, which represents the specification in a form which the implementation can be checked against using simple model-checking techniques.

This paper is organised as follows: we will briefly survey existing work on induction and the theory of data independence we will be using; we then present the method and give a detailed example of its implementation; we discuss a further example of a communications protocol for channel multiplexing; finally, we present our conclusions and discuss its general applicability.

2 Related Work

A number of authors have demonstrated that induction is a method which can be successfully used in the analysis of distributed systems. Kurshan and McMillan [11] and Wolper and Lovinfosse [18] published similar work using structural induction to reason about systems with unboundedly large numbers of identical components. Both methods require an invariant to be defined and rely on proof obligations which correspond to the base case

[1] Appendix A contains a summary of the CSP relevant to this paper.

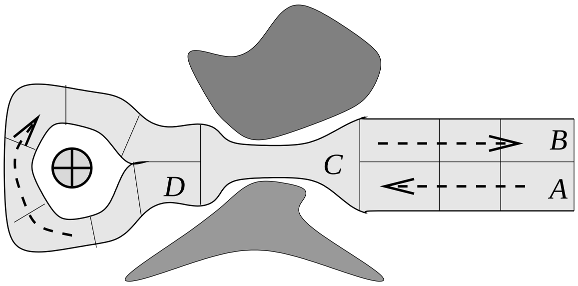

Figure 1: Road with a narrow section

and inductive step (using model checkers to discharge them). In [11] the method is demonstrated on an algorithm for maintaining the consistency of multiple copies of data object in a distributed system. In [18] two examples are given: a buffer composed of identical elementary buffers and a version of the distributed mutual exclusion problem (similar to that in [11]).

The following example is a good illustration of this basic induction method, in particular because it demonstrates the power of structural induction to cover a wide variety of network topologies, as alluded to in [3]. Figure 1 shows a road layout allowing visitors to drive around a monument[2]. All cars enter the road at A and pass around the loop until they exit at B. Two adjacent rocks mean that one stretch of road is too narrow for cars to pass each other – it is single-lane there and two-lane or one-way elsewhere. To prevent deadlock in the road system a constraint has to be applied to the network, namely that out-going cars (i.e. rightward in the picture) are given priority over in-coming ones, in the sense that an in-coming car is not allowed to enter the single-lane segment C until segment D is empty.

It is possible to model the actual road network seen in Figure 1 to show that the constraint described makes the network deadlock-free, and FDR will prove this in a few seconds. However, this only establishes this property for the particular network modelled, and in no way implies that any other network topology is also deadlock-free. This more general goal can be achieved using structural induction.

To perform a structural induction a range of structural rules are required which describe how the various components have to be connected. In this case these components are the CSP processes representing the lane segments of type A, B and CD (it is necessary to constrain the appearance of segments of type C to only occur with segment D attached, as described above). The rules describe how these lane segments should be connected, and also how two

[2]This example is based on an exercise in [14].

deadlock-free road networks can be connected to each other. For example: (i) *"We can have a single road segment of type A"* (this is the base case). (ii) *"The road network can have segments of type A connected to its* `in` *channel"* (an inductive step).

The rules are translated into refinement relations which are the proof obligations for the induction, they can be automatically discharged using FDR. The FDR checks performed are:

```
(i)      P [F= A
(ii)     P [F= P [in <-> out] A
(iii)    P [F= P [out <-> in] B
(iv)     P [F= P [in <-> leftout, out <-> leftin] CD
(v)      P [F= P [in <-> out] P,
```

where P is the property of deadlock freedom, and '`[F=`' refinement in the failures model of CSP. The piping operator `X[a<->b]Y` connects the channels `a` and `b` (belonging to `X` and `Y` respectively) together, and hides all communications over them.

Note that the processes `A` and `B` are equivalent, they are distinguished in this example for consistency with the Figure. Check `(i)` establishes the base case for the induction. Checks `(ii)` and `(iii)` establish that road segments of the type A and B can be tacked on to deadlock-free road networks in the manner described and the resulting road networks will maintain this property. Check `(iv)` establishes the same for segments of type `CD`, and `(v)` establishes that any two deadlock-free road networks (of the type under consideration) can be connected together (the input to one network is the output of the other), and the resulting road network will also be deadlock-free.

It is clear that by transitivity of refinement, and so by using normal inductive reasoning, that in combining `(i)` to `(v)` above, it is possible to claim that any road network made up of segments of type `A`, `B` and `CD`, according to the given rules, will be deadlock-free. More interesting case studies of this technique can be found in [4, 5, 15].

The buffer example in [18] is a remarkable one. Here the property proved by induction bears little resemblance to the final objective, but when combined with data independence properties of the implementation is enough to prove it (though the definition of buffer implied in that paper is not quite the CSP standard one of [9, 14]). We remark that this provides a completely different combination of data independence and induction to the one we will shortly describe. Other approaches for proving properties of systems of arbitrary size have included the use of temporal logic [1, 2, 8, 16].

Just because a property is true of all the systems constructed by a set of structural rules does not guarantee that it can be proved inductively. Ideas such as *strengthening the hypothesis* frequently help, but can require considerable ingenuity. The main limitation on the induction method, described above, is that it can only address systems in which the individual components are completely independent of the overall structure of the system: adding an extra process does not change the semantics of the components that were already there. For example, they cannot know the identities of the other components, for then it is not true to say that an $n+1$–process system can be constructed by adding one more to the n–process version. It is this limitation on induction that we address.

3 Data Independence

A system is said to be data independent of a data-type variable T when it makes sense for any non-empty substituted type and it satisfies structural rules, meaning that it handles the type in a relatively simple way. The process representing the system may not perform any operations on values of that type, it can only input them, store them, output them, and perform equality tests between them. Many communications protocols are data independent since they simply pass around data ensuring that only desirable recipients are able to extract it; their behaviour is entirely independent of the data itself.

Data independence ideas have been developed for a number of notations, having first been studied formally in [17], but we use here the theory developed for CSP by Lazic and Roscoe (see [12, 13] and Section 15.2 of [14]). They have developed theorems which state that for certain (data independent) systems it is possible to verify that the system possesses certain properties, for all instances of its independent types, by performing a specific finite number of checks with finite instances of the types. Data independence theorems frequently allow us to generate thresholds for given checks: a size of type T such that one or more checks of a property for this and perhaps smaller types will imply that the property holds for all T. Space unfortunately prevents us from quoting in detail the various theorems and definitions of data independence, for which we refer the reader to [12, 13, 14].

However, of particular relevance to the technique we present in this paper is the ability to consider data independent processes with the addition of constant symbols and (in general many-valued) predicates on variable types. The theorems used put some constraints on the use of these predicates. They must be uninterpreted, in that they should be treated as symbols and a verification should establish a property for all possible interpretations. The predicate must be a function into a fixed finite type, which in our case will be {*true*, *false*} (otherwise the problem would become intractably infinite). This will be important to us as it will allow statement of an inductive principle which to some extent overcomes the limitation discussed at the end of Section 2. Again we must refer the reader to [12, 13] for relevant theorems and definitions.

4 Induction Over Data Independent Types

Our technique is useful in the analysis of systems which are built as indexed parallel compositions over types which each individual process (elements of the type) treats data independently. We can't deal with these systems using traditional inductive techniques because the individual processes vary with the type (they know each others identities, for example, if their identities are the data independent type) and we can't deal with them using data independence alone because indexed parallel composition over data independent types is prohibited in that theory[3]. While our method readily extends to some more complex cases (one of which appears in Example 2 below), we will consider here the objective of proving that a specification `Spec`, data-independent in T, is refined by the parallel combination `|| x:T @ [A(x)] N(x)` in which the terms `A(x)` and `N(x)` are themselves data independent in T. Here `N(x)` is the component process, and `A(x)` is its alphabet. Our aim is to show that the parallel combination of `N(x)` processes, where `x` is an element of type T, refines `Spec` no matter how large T is. Further, we want to do this using only a finite amount of work.

We build a (usually sequential) process representing partially constructed versions of the system under consideration. The aim is that it should be data independent in T, should be refined by the partially constructed network it models, and when complete should refine `Spec`. As this process is often a representation of the state of a parallel system we term it `Superstate(S)`, where `S` is the (nonempty) subset of T that constitutes the partial network. The aims can thus be formally stated

$$\texttt{Superstate(S)} \sqsubseteq \texttt{(|| x:S @ [A(x)] N(x))} \tag{1}$$
$$\texttt{Spec} \sqsubseteq \texttt{Superstate(T)} \tag{2}$$

These two statements, if true, are of course sufficient to establish our overall aim. If an appropriate `Superstate` has been constructed, then (2) presents no new theoretical challenges as it already fits within the established theory

[3] This is largely because allowing such compositions would allow one to count the type T.

of data independence: we can reasonably expect it to be verifiable independently of T by finding the appropriate threshold and doing the necessary refinement checks.

The role of induction comes in establishing (1), which does not come within standard data independence theory as it contains the indexed parallel composition. What we do is to perform an induction on the size of `S` (from 1 up to $|T|$). The base case of the induction is to show that one node refines `Superstate`:

$$\texttt{Superstate(\{c\})} \ \sqsubseteq \ \texttt{N(c)} \tag{3}$$

where the constant symbol `c` represents an arbitrary member of T. The inductive step is to show that for any nonempty `S` and constant `c` outside `S`,

$$\texttt{Superstate(S')} \ \sqsubseteq \ \texttt{Superstate(S)[AA(S)||A(c)]N(c)} \tag{4}$$

where `S'` is the union of `S` and `{c}` and `AA(S)` is the alphabet of `Superstate(S)`, which is the union of `{A(x) | x <- S}` (the alphabets of each node).

If we can establish these two things the inductive proof of (1) follows easily: if it is true for `S` then

$$\begin{array}{lll} \texttt{Superstate(S')} & \sqsubseteq & \texttt{Superstate(S)[AA(S)||A(c)]N(c)} \\ & \sqsubseteq & \texttt{(|| x:S @ [A(x)] N(x))[AA(S)||A(c)]N(c)} \\ & \sqsubseteq & \texttt{(|| x:S' @ [A(x)] N(x))}. \end{array}$$

The second line is from the monotonicity of the parallel operator and induction, and the final one from the definition of indexed parallel.

We can thus conclude that the truth of (3) and (4) (for arbitrary T, `S` and `c`) imply that (1) holds for all nonempty S and finite T. Note that this conclusion actually requires not one induction, but rather demonstrates that an infinite number of different inductions (one for each T) all work.

Fortunately, both (3) and (4) are statements that can be proved using data independence extended by predicate symbols and constants, as we can naturally treat `S` (the subset of the network present in a partially-built network) as a predicate symbol (mapping a member of T to *true* if it is in the set). Therefore we can reasonably hope to justify these statements for all values of their parameters, and thus be able to conclude (1) from a finite collection of checks which can be verified automatically on FDR.

One way of looking at this technique is that it provides a way of extending the advantage of data independent proofs to systems which involve indexed parallel composition. However this can only be done in cases where an appropriate `Superstate` definition can be found, something which at best requires more ingenuity on the part of the user than other data independence results, and may well in some cases be impossible. (The greatest challenge in building Superstate appears to be keeping it sufficiently finite-state to enable both finite thresholds and automated checking.)

Example 1 Consider an example of a distributed system which combines leadership election and token passing. This example has much in common with one of those given in [3, 18], though the main point of the present one – the knowledge of all identities by each process – is not present in these papers. It is the leader's job to distribute a token in turn allowing some critical region to be entered into by the process having it. The leader can do critical sections (though this is completely optional in the model), and the leadership can only move when the leader has the token. Below is the CSP description of a process expressed as input syntax for FDR:

```
Leader(T,id) = leader?j:T -> (if j==id then Leader(T,id)
                                        else Empty(T,id,j))
             [] cr_start.id -> cr_end.id -> Leader(T,id)
             [] pass!id?j:others(id,T) -> pass.j.id -> Leader(T,id)

Empty(T,id,L') = pass.L'.id-> Full(T,id,L')
             [] leader?j' -> (if j'==id then Leader(T,id)
                                         else Empty(T,id,j') )

Full(T,id,L') = pass.id.L' -> Empty(T,id,L')
             [] cr_start.id -> cr_end.id -> Full(T,id,L')

others(i,T) = diff(T,{i})
```

diff(P,Q) is the difference of the two sets P and Q. The parameter T is the type of nodes in use; it is required since nodes need to know who is present in the system. If S is any set of these nodes, and L0 is a constant representing the initial leader, we can build the composition of all the nodes from S as:

```
LNet(T,S,L0) = || i:S @ [A(T,i)] (if i==L0 then Leader(T,i)
                                           else Empty(T,i,L0))
```

LNet(P,S,L0) is the parallel composition of processes in S of the appropriately initialised node processes, which we can name N(T,L0,i) for consistency with the notation earlier. Note that this model abstracts away from the leadership election process, which is delegated here to a perhaps unrealistic multi-way synchronisation.

We need, for our method, to create a process Superstate(S) that models the behaviour of the parallel composition of the nodes from a subset S of T. These will inevitably agree (thanks to the synchronisation on leader) on who the leader is. The main problem one faces in building the process is what to do when the leader is outside S, for then the partially composed network has many aberrant behaviours due to the fact that there is no coordinated leadership; since the leader is outside of the network it can misbehave. In particular, the leader may allow any number of nodes to get into critical sections at once. In order to overcome this the `Superstate` process incorporates this misbehaviour; essentially things stay fine as long as no imaginary leader outside `S` hands out too many tokens:

```
Superstate(T,S,L) =  leader?j:T -> Superstate(T,S,j)
            [] member(L,S) & cr_start.L -> cr_end.L ->
                                         Superstate(T,S,L)
            [] member(L,S) & pass!L?j:others(L,T) ->
                                         Superstate'(T,S,L,j)
            [] not member(L,S) & pass!L?j:others(L,S) ->
                                         Superstate'(T,S,L,j)

Superstate'(T,S,L,j) =  member(j,S) & cr_start.j ->
                                         Superstate''(T,S,L,j)
               [] pass.j.L -> Superstate(T,S,L)
               [] not member(L,S) &
                      (STOP |~| pass.L?k:S -> CHAOS(Events))

Superstate''(T,S,L,j) =  cr_end.j -> Superstate'(T,S,L,j)
               [] not member(L,S) &
                      (STOP |~| pass.L?k:S -> CHAOS(Events))
```

`CHAOS(Events)` is the process that can perform or refuse any possible event: it is used here when we don't care what the process does, namely after one of the misbehaviours mentioned above. This approach is similar to that used by Wolper and Lovinfosse in [18].

Because we have adopted the names for processes as used in the earlier formulation of the inductive claim, the base case and step case of the induction are almost literally (3) and (4). The result proved by the induction is (1) where, when $S = T$ the right-hand-side is the complete network `LNet(T,T,L0)`. As we will shortly see, by analysing the checks (3) and (4) we can establish finite collections of checks which prove them, and hence (1), for all T and S.

Though we have not had space to detail the results underlying these calculations, it is nevertheless interesting to derive thresholds and sufficient sets of checks to establish the base and inductive cases of our induction. For checks which either explicitly or (like ours) implicitly involve equality checks between members of the data independent type, the threshold when there are no predicate symbols, is the greatest number of potentially distinct values that might be present simultaneously during a symbolic comparison between the specification and implementation. When there is a single predicate symbol, it is the sum, over all values the predicate might take, of the largest number there might be in a comparison with that value. The formulae, such as the following, (quoted in Section 15.2 of [14]) provide upper bounds, often crude, for these quantities.

$$B = W^{Spec} + W^{Impl} + max(L^{Spec}, L^{Impl}_{?}, L^{Impl}_{\sqcap})$$

where W^{Spec} and W^{Impl} are the maximum number of values of type T that the Specification and Implementation, respectively, ever have to store for future use. $L^{Impl}_{?}$ and L^{Spec} are the largest number of values of type T that can be input in any *single* visible event of the Implementation and Specification, respectively. $L^{Impl}_{\sqcap}$ is the largest number of values of type T that can be nondeterministically chosen in any single nondeterministic choice made over sets involving T in the implementation. In the base case (3) this particular formula predicts a threshold of $2 + 2 + 1 = \mathbf{5}$, and in the step case (4) the same formula gives $2 + 2 + 2 + 1 = \mathbf{7}$. However, the presence of the predicate symbol increases the second threshold to 13, 6 in S and 7 outside S, (as by definition we know that the constant c does not satisfy the predicate). The predicate brings about the increase because the values stored in the program may or may not satisfy it, and the threshold has to be big enough to cater for this.

These naive calculations assume that all processes can have distinct values for the leader, whereas in fact they all agree. Let us concentrate on the more complex case: the threshold for (4). Consider W^{Spec} and W^{Impl}, the leaders `L` and are counted in both but must be identical, as must `L'`. Therefore, we can ignore two of them giving a new threshold of 9: 4 in S and 5 outside S. Also, the `j` in W^{Impl} is either identical to the `L'` or the `id` in `Empty` giving a new threshold of 7: 3 in S and 4 outside S. Finally, we know that `j` and `L` can not both be outside $S \cup \{c\}$ because if the leader and the token are actually both outside the $S \cup \{c\}$, then no-one in our system knows where the token is. Thus we can deduct another 1 from the number that can be outside S. Our final threshold is 6, 3 in S and 3 outside. A largest inductive step check we would need to do is:

```
Superstate({1..6},{1..4},1) [FD=
Superstate({1..6},{1..3},1)
[AA({1..6},{1..3})||A({1..6},4)] Empty({1..6},4,1)
```

Where '`[FD=`' means refinement in the *failures/divergences* model of CSP, this being the strongest notion of process refinement. Further checks (there are 24 including the above) need to be performed for the cases in which there are fewer objects either or both inside and outside S, and to deal with the

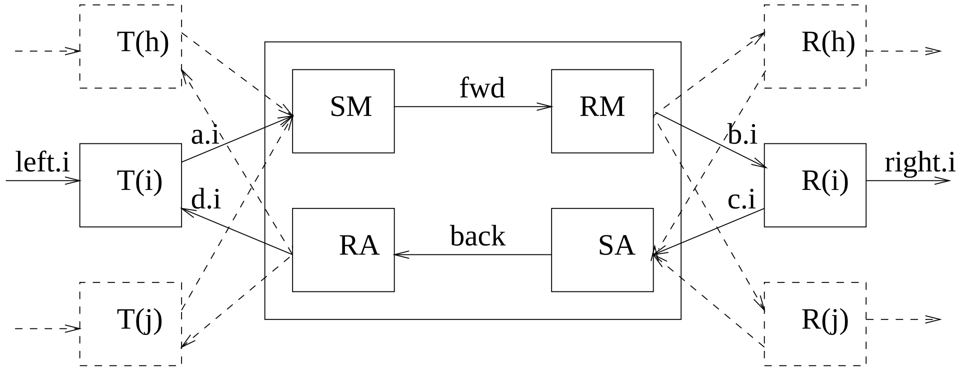

Figure 2: Multiplexing Channels

cases where `L0` is inside `S`, equal to `c`, or neither. These checks can, of course, be performed quickly and easily on FDR. We believe that the threshold and number of required checks could be reduced further by *ad hoc* arguments, but much the best prospect here is the ideas for symbolic execution discussed in the Conclusions below. Similar analysis can bring the threshold for the base case down to 3, so a largest check of the 5 required is

```
Superstate({1..3},{1},1) [FD=  Leader({1..3},1)
```

An interesting fact about this example – though by no means universally true – is that `Superstate(T,T,L0)=LNet(T,T,L0)`. It follows that *any* data independent specification we want to prove of `LNet(T,T,L0)` can be addressed through this particular `Superstate`. Examples that can be proved include deadlock freedom and at most only one critical section being active at any one time: each, provided it is suitably finite-state and data independent, should be provable in general as an instance of (2).

Example 2 Consider a simple communications protocol which enables the multiplexing of as many channels as we please over a single pair of links – one each way (`fwd` and `back`). This is done in our implementation by passing tagged data and acknowledgements over the links, and combining a protocol with buffering to stop any one channel impeding others. For simplicity we only consider the case where all the channels are in the same direction (from `left.i` to `right.i` in Figure 2).

Evidently, as we let the number N of channels grow, the number of states in the resulting CSP implementation soon becomes unmanageable for a model checker. However, the sort of data independent induction presented above can prove that our system is equivalent to N independent one-place buffer processes for any N. Thereby establishing the correctness of the protocol or any arbitrary number of multiplexed channels.

There are actually two types in the example that are susceptible to data independence arguments. The more obvious one is the type of data passed by the channels: the theorems of [13, 14] easily show that it is sufficient to set this type with size 2 to prove that the system behaves properly for any type. This argument is quite separate from the induction. The type over which we induct is that of `TAGS` (the channel names).

The system is constructed of the processes `T(i)` and `R(i)`, which are the transmitter and receiver processes respectively (one for each `i` in `TAGS`), connected to the fixed collection of processes `SM`, `RM`, `RA` and `SA`. The `T(i)` and `R(i)` communicate with the environment on the `left.i` and `right.i` channels. Below are the definitions of the processes:

```
SM = a ? i:TAGS ? x -> fwd ! i . x -> SM
RM = fwd ? t ? x -> b . t ! x -> RM
SA = c ? i:TAGS  -> back ! i -> SA
RA = back ? t ? x -> d . t ! x -> RA

T(i) = left.i ? x -> a.i ! x -> d.i -> T(i)
R(i) = b.i ? x -> right.i ! x -> c.i -> R(i)
```

The central core of the system is the following:

```
CORE = (SM [|{|fwd|}|] RM)\{|fwd|} |||
       (SA [|{|back|}|] RA)\{|back|}
```

This process (data independent in `TAGS`) has the same parallel structure however large `TAGS` is, but we get a situation similar to that in Example 1 when the transmitter/receiver pairs are added, as this requires parallel compositions indexed over `TAGS`. The result has the channels `a`, `b`, `c` and `d` hidden, leaving only `left` and `right` visible to the environment. Our objective is to prove that the complete system is the composition of a `COPY` for each tag, where `COPY` is the process which inputs data on its *left* channel, and outputs that data on its *right* channel (behaving as a one-place buffer). `COPY` is defined as:

```
COPY = left?x -> right!x -> COPY
```

In order to prove this we show that the abstraction to an arbitrary channel is correct by induction, and deduce the overall result from that. We shall concentrate on channel `C0`. The specification `SPEC(S)` of a partially constructed system, where `S` $\subseteq$ `TAGS` and `C0` $\in$ `S`, says that on `C0` the process must act like a one-place buffer except that it can refuse to input or output when there is outstanding data outside S on the a/b or c/d link. (Space constraints prevent us from giving the detailed code of this specification here.) Our claim is that a network constructed from CORE plus all the T(i) and R(i) from i in S, all internal communication hidden and all channels in S other than C0 lazily abstracted[4], refines SPEC(S) (the specification of the system of S channels multiplexed).

The base case of our proof is that the system with one transmitter/receiver pair (labelled C0) refines the corresponding specification:

```
BASE = (CORE [|{|a.C0,b.C0,c.C0,d.C0|}|] (T(C0)|||R(C0)))
                          \ {|a.C0,b.C0,c.C0,d.C0|}
SPEC({C0}) [F= BASE
```

The inductive step required is that if we add a further T(K)/R(K), hiding internal and abstracting external communications appropriately, we preserve SPEC:

```
SPEC(S') [F= (LAbs(SPEC(S) [| intevs |]
             (T(K)|||R(K)))({|left.K,right.K|})) \ intevs
intevs = {|a.K,b.K,c.K,d.K|}
LAbs(P)(X) = (P [|X|] CHAOS(X))\X
```

Here LAbs(P)(X) is the *lazy* abstraction (as discussed above) of events X from process P, and S' is the union of S and {K}.

As in Example 1, data independence theorems allow thresholds to be calculated which establish an upper limit on the size of TAGS which guarantees that the refinements will hold for any arbitrary non-empty size of TAGS (i.e. largest size of TAGS). For the base case the threshold is no more than **8**. For the inductive step the threshold is **7**. This is unlike the situation in Example 1 where the step threshold was significantly larger. This is primarily because only the base case contains the process CORE which stores 4 values. So by performing a fixed finite amount of refinement checks on FDR, for each possible configuration of each check with S smaller than or equal to the threshold, we have shown that for any size of TAGS our multiplexed system is equivalent to that many copies of COPY. Therefore, the integrity of each channel is not compromised by the addition of further channels over the same pair of links (fwd and back).

[4]To lazily abstract a channel means to hide the channel in such a way that we do not assume that hidden events must occur. It provides the best way of formulating what a process looks like to an observer who can only see a subset of its alphabet. See Chapter 12 of [14].

5 Conclusions

The technique described here provides an interesting combination of induction and data-independence which is capable of allowing model checkers to deal with a significant new class of problem. While the examples quoted here are fairly simple, we in fact developed the method as part of an analysis of a much more complex case study [6] – a version of the Time Triggered Group Membership Protocol(TTP) [10]. We successfully applied the method there after generalising the protocol to make it sufficiently data independent, and correcting a fault that our analysis revealed. Other examples we have considered include: extending the applicability of the structural induction argument in [5] from binary to arbitrary branching; and the proof of a routing protocol.

Future work will hopefully lead to an understanding and broadening of the classes of systems to which our method applies. We are also interested in the methodology required for formulating the `Superstate` induction hypothesis and whether this can be automated. Ongoing developments in the symbolic handling of data in refinement checks on FDR should be a great help in discharging the model checking obligations generated by our techniques. In effect this should completely automate the application of data independence. It would no longer be necessary to calculate thresholds (and the ad hoc arguments used to bring the threshold down), as all of the analysis would be done at "run-time" and automatically. Furthermore, there is every hope that the checks would complete much faster, as this method should reduce many equivalence-classes of essentially similar states down to a single one.

Acknowledgements

We would like thank Ranko Lazić for his work on data independence, and Gavin Lowe whose independent work on the integration of data independence and induction led us to look for the connections we eventually found.

References

[1] M.C. Browne, E.M. Clarke and O. Grumberg, *Reasoning about Networks with Many Identical Finite State Processes*, Information and Computation, 81(1), 13-31, April 1989.

[2] E.M. Clarke and O. Grumberg, *Avoiding The State Explosion Problem In Temporal Logic Model Checking Algorithms*, Proceedings of the 6th Annuak ACM Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987.

[3] E.M. Clarke and O. Grumberg and S. Jha, *Verifying Parameterized Networks using Abstraction and Regular Languages*, Proceedings of CONCUR'95, 395-407, LNCS 962, 1995.

[4] S.J. Creese, *An Inductive Technique for FDR*, Master's thesis, Oxford University, 1997.

[5] S.J. Creese and J. Reed, *Verifying End-to-End Protocols Using Induction with CSP/FDR*, Paper accepted for presentation at FMPPTA'99, Puerto Rico, April 1999.

[6] S.J. Creese and A.W.Roscoe, Oxford University Computing Laboratory Technical Report, PRG-TR-1-99.

[7] *Failures-Divergence Refinement: FDR2 User Manual*, Formal Systems (Europe) Ltd, 1992-7.

[8] Steven M. German and A. Prasad Sistla, *Reasoning about Systems with Many Processes*, Journal of ACM, 39, 675-735, July 1992.

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

[10] S. Katz, P. Lincoln and J. Rushby, *Low-overhead time-triggered group membership*, Proceedings of WDAG '97, LNCS 1320, 1997.

[11] R.P. Kurshan and K. McMillan, *A Structural Induction Theorem for Processes*, Proceedings of 8th Symposium on Principles of Distributed Computing, Edmonton, 1989.

[12] R. Lazić and A.W. Roscoe, *Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays*, Proceedings of INFINITY'98, Aalborg, Denmark, July 1998, extended version as Oxford University Computing Laboratory TR-2-98.

[13] R.S. Lazić, *A semantic study of data-independence with applications to the mechanical verification of concurrent systems*, Oxford University D.Phil thesis, 1999.

[14] A.W. Roscoe, *The theory and practice of concurrency*, Prentice Hall, 1998.

[15] J.N. Reed, D. M. Jackson, B. Deianov and G. M. Reed, *Automated Formal Analysis of Networks: FDR Models of Arbitrary Topologies and Flow-Control Mechanisms*, Proceedings of ETAPS'98, Lisbon, Portugal, to appear in IEEE Transactions on Software Engineering, March 1998.

[16] Moshe Y. Vardi and Pierre Wolper, *Reasoning about Infinite Computations*, Information and Computation, 115, 1-37, 1994.

[17] P. Wolper, *Expressing interesting properties of programs in propositional temporal logic*, 184-193, Proceedings of the 13th ACM POPL, 1986.

[18] P. Wolper and V. Lovinfosse, *Verifying Properties of Large Sets of Processes with Network Invariants (Extended Abstract)*, Proceedings of the International Workshop on Automatic Verification Methods for Finite State Machines, LNCS 407, 1989.

Appendix A. The CSP Language

The CSP processes that we use are constructed from the following:

- `STOP` is the simplest CSP process; it never engages in any action, nor terminates. It is equivalent to deadlock.
- `a -> P` is the most basic program constructor. It waits to perform the event `a` and then behaves as process `P`. The same notation is used for outputs (`c!v -> P`) and inputs (`c?x -> P(x)`) of values along named channels.
- `P |~| Q` represents internal choice. It behaves as `P` or `Q` *nondeterministically.*
- `P [] Q` represents external choice. It will offer the initial actions of both `P` and `Q` to its environment at first; its subsequent behaviour is like `P` if the initial action chosen was possible only for `P`, and like `Q` if the action selected `Q`. If both `P` and `Q` have common initial actions, its subsequent behaviour is nondeterministic (like `|~|`). `STOP [] P` behaves as `P`.
- `P [| a |] Q` represents parallel composition. `P` and `Q` evolve concurrently, except that events in `a` occur only when `P` and `Q` agree to perform (i.e. *synchronise* on) them.
- `P ||| Q` represents interleaved parallel composition. `P` and `Q` evolve separately, and do not synchronise on any events. Equivalent to `P [| {| |} |] Q`
- `P [ a || b ] Q` represents alphabetised parallel. `P` and `Q` have to agree on events in the intersection of their alphabets.
- `|| x:a @ [A(x)] P(x)` represents replicated alphabetised parallel. This constructs the parallel composition of `P(x)` processes, one for each `x` in `a`, over their respective alphabets.
- `P \ A` is the CSP hiding operator. This process behaves as `P` except that events in set `A` are hidden from the environment and are solely determined by `P`; the environment can neither observe nor influence them.
- `P [[ a <- b ]]` represents the process `P` with `a` renamed to `b`.
- `P [ a <-> b ] Q` is the linked parallel operator. `P` and `Q` synchronise on channels `a` and `b`, which have been renamed to the same and hidden. There are also straightforward generalisations of the choice operators over non-empty sets, written `|~| x:X @ P(x)` and `[] x:X @ P(x)`.