TTP: A case study in combining induction and data independence

S.J. Creese and A.W. Roscoe*
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, UK

Abstract

The TTP, or *Time Triggered Protocol* is designed as an efficient mechanism for diagnosing faulty members of a network of peer processes. It exists in a number of versions and can be considered at a number of levels of abstraction. We take as our starting point the version presented in [10]. We first examine briefly how it can be modelled in CSP and particular instances analysed on FDR. This gives some useful insights into the model checking process and in particular the role of bisimulation as a compression mechanism, and also uncovers a flaw in the protocol as presented in the original version of [10]. We generalise the protocol in several ways to make it data independent in the type of nodes, and show how this can lead to an inductive proof that is checkable on FDR.

1 Introduction

The TTP, or *Time Triggered Protocol* is a low-cost mechanism that allows classes of communication faults in nodes to be diagnosed in networks where processes (i) can perform and listen to broadcasts and (ii) where there is a notion of time which allows each to determine which node's turn to broadcast it is. The version we are considering is that of [10], which ignores questions such as how the clocks are kept sufficiently synchronous. Later versions of the protocol are being developed for use in the automotive industry.

The scenario of the protocol is that there are N nodes, all initially working normally. It is assumed that faults take one of two forms:

- A node which attempts to broadcast fails, and *none* of the other processes hear. The guilty node is not immediately aware of the failure, however, and continues as through its broadcast has succeeded. This is termed a *send fault.*

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

- A node may fail to hear a broadcast, thereby committing a *receive fault*. It cannot immediately tell when this occurs, since there is no way locally to distinguish this situation from a broadcast fault in the scheduled sender process.

The objective is to allow correctly-working processes to identify faulty ones as quickly as possible, and to have faulty ones recognise their own failings and shut down. Provided that assumptions are made about the rate at which new faults occur, and that at least two correct nodes always remain, the protocol seeks to achieve these objectives via one-bit broadcasts by each of the nodes in turn, cyclically. The details of the protocol are concerned with what each node does (a) when it is its own turn to broadcast and (b) when it either hears or does not hear a broadcast from another node. We will discuss the details of the protocol later.

As a protocol used to support safety-critical applications, it is especially important that the specification of what the TTP does be clear, including the assumptions which limit its applicability, and that we should get a high degree of confidence that it meets this specification. We have chosen to describe the protocol, the assumptions, and the specification in the CSP notation [8, 16]. This notation provides a wide range of constructs for describing systems that interact by communication, and has frequently been used for analysing the consequences of potential faults. It is rich enough to express things like fault rate assumptions and specifications both clearly and concisely.

The TTP protocol is defined to work for any number of nodes greater than or equal to 2, and we would like to be able to prove that it meets its specification however many there are. This creates a problem for model checking programs such as FDR [15, 6] or Murϕ [3], for while these are comparatively easy to use and well-suited to this type of system, they require a concrete system to work on. The best that can be achieved with a model checker, at least using it in the obvious way, is to show the extent to which the protocol meets its specification for as large a number of nodes as it can handle. In the present case this limit[1] is large enough to give the user a reasonable degree of confidence about what happens in general, but of course this is not a proof. The assertion that the TTP works for an arbitrary number of nodes is just one instance of the *Parameterised Verification Problem* or PVP [2, 20]: proving a result for a parameterised system independent of the value of one or more of its parameters.

There are a number of possible ways in which we could try to provide a proof. The first is "pencil and paper", namely a human being doing the proof and relying on other humans checking it (see [7] for a similar example). Unfortunately this is, in the case of complex distributed algorithms, both tedious and fallible. A second possibility is to aid the the human prover with a theorem proving tool such as PVS or Isabelle: depending on the complexity of the

[1] We will be more precise about what can be achieved later.

problem under consideration these might either require minimal or substantial input and guidance from the user to complete a proof. An examination of PVS applied to the TTP protocol can be found in [17]. The third approach, and the one we shall be examining in this paper, is the structured use of model checking to prove general results via techniques like induction (on the number of nodes) and data independence.

The concept of induction is well known to all mathematicians and theoretical computer scientists: one formulates an assertion that is claimed to hold over some well-founded partial order X (i.e., one with no infinite decreasing sequences) and proves that it holds for any $y \in X$ on the assumption that it does so for all $z < y$. This allows us to conclude that the assertion holds of all members of X. A number of authors [19, 2] have shown that it is applicable to distributed systems: the most usual approach has been to show that a system constructed out of N identical nodes (possibly build around some fixed "base case") satisfies some property S by showing that the base case satisfies S and that adding one node onto a system satisfying S creates a system that still does. As a trivial example one can show that versions of the dining philosophers either deadlock or not, independent of N, by a few assertions that are trivial to verify on model checkers. This approach tends to deal with systems where there is only one possible system of a given size N; it is also possible to use methods akin to *structural induction* to show that some class of networks built according to a fixed set of rules all have some property [1].

It is an obvious idea to try to use induction to prove that the TTP works. Unfortunately there are a number of reasons why this is non-trivial, the principle of which are (a) the calculations nodes perform to decide whose turn it is to broadcast: these are highly dependent on how many nodes are present; (b) the fact that each node knows the identity of all others, so that adding an extra node apparently means we have to adjust the ones that are already there; and (c) the fact that individual node processes' state spaces grow (for various reasons) with the number of nodes.

We will deal with these problems by two methods: firstly generalising the protocol in such a way that it does not rely on the round-robin allocation of broadcasting turns of the original – we will simply assume that there is some unspecified mechanism by which the nodes can agree whose go it is. This initial transformation allows the second method: *data independence* (first studied in [18]) in the type of nodes. A process P can be said to be data independent in a given type T if P makes sense whatever nonempty set is substituted for T, which can be thought of as an unconstrained parameter in P. In some circumstances, which we will discuss briefly later, one can prove properties of the data independent program $P(T)$ by establishing it for a finite collection (often with just one member) of finite types.

The rest of this paper is structured as follows: the next section discusses the modelling and specification of the protocol of [10] in CSP and our experience in running these checks on FDR. We then discuss the amendments required to remove the ring structure – the main considerations being how to adjust the assumptions that are required for the system to operate correctly. This leads to a version which is data independent in the type of nodes except for one important consideration, namely that it involves parallel composition indexed over the same type. The final sections of the paper are devoted to showing how inductive proof can overcome this difficulty and establish the correctness of the generalised protocol independent of size using via a finite number of finite checks on FDR.

We have included only a minimum of actual CSP code in this paper: giving even a reasonably complete presentation within a reasonable space would have been impossible given that we deal with multiple versions of a non-trivial program.

2 The TTP in CSP and FDR

2.1 The model of the TTP

The CSP model of the TTP is constructed of a process representing a node, and the processes representing the assumptions under which the protocol is designed to operate. It is possible to build networks of `size` protocol nodes by putting `size` node processes in parallel with the protocol assumptions. By constructing CSP processes representing the various specifications that the protocol is designed to meet we were able to use the FDR tool to perform refinement checks that established whether or not these specifications held.

At this level of abstraction the notion of time is important because the protocol guarantees aspects of nodes behaviours with respect to discrete points in time. It is important that all processors agree on whose turn it is to communicate: in our CSP model we abstracted this agreement into the processes communicating a special event *tock* at regular intervals. This event might correspond so a real communication such as a broadcast clock signal, or might just be a convenient way of implementing the *assumption* that nodes agree on time. (Note that at lower levels of abstraction than ours, the TTP includes mechanisms for clock synchronisation.)

There are two claims made of the TTP in [10]:

(i) That all non-faulty nodes agree on who is faulty at each point in time.

(ii) That faulty processors will be promptly diagnosed as faulty by non-faulty processors, and that the faulty processor will also diagnose itself as faulty.

In the TTP each node has a *local membership set* [*lms*] and an *ack* bit (of type *Bool*). The *lms* is the set of all nodes that the node thinks are non-faulty, note, this may or may not contain itself. The *ack* bit indicates whether the node believes that the previous slot was faulty or not; it is set to true when faulty. When it is a node's turn to broadcast it will only do so if it believes itself to be non-faulty, i.e. contained in its *lms*; it appends to its broadcast its *ack* bit, which is then used by the other nodes to determine whether or not a node has become faulty. After each broadcast nodes make calculations as to who, if anybody, they think has become faulty, based on the *ack* bits they have observed. If nodes believe it to be the broadcast slot of a faulty node they ignore any *broadcast* events that occur and do not change any of their local parameters. The *lms* of each node is updated accordingly. In our analysis of the protocol we modelled the broadcast communications as containing solely the *ack* bit, since we are only concerned with the *ack* bit and not the content of the message. A detailed specification of the protocol as described in [10] can be found in Appendix A.

Whilst it is the case that TTP protocol nodes only actually enter into broadcast communications, our CSP model of a protocol node takes a different form. The CSP node is able to perform a much wider variety of communications during every time step for two reasons: because we use synchronisation on channels with other processes to uphold protocol assumptions, and so that we can check the state of nodes at certain points in time allowing us to determine whether the protocol is meeting its specifications. The real node should be seen as sitting inside a CSP shell which provides us with channels on which we can observe its behaviour so allowing us to perform our analysis.

As has already been described, nodes can suffer one of two types of fault: a send fault or a receive fault. Faults are intermittent, a faulty node may behave correctly at one point in time and incorrectly at another. Once a node has become faulty it remains faulty. Once a node has self-diagnosed itself as faulty it will remove itself from its *lms* and no longer be prepared to broadcast in its broadcast slot. The protocol is designed to meet the specification given in the form of the two properties above under two assumptions:

- **Minimum non-faulty nodes**: there are always at least two non-faulty nodes.
- **Fault arrival rate**: only one non-faulty node becomes faulty in any $n+1$ consecutive time slots.

There are two other more subtle assumptions about fault occurrence that are discussed to some extent in [10], which we make explicit here:

- Even though faults are intermittent, a node always exhibits a fault immediately after going faulty: i.e., a node only 'becomes faulty' when it actually commits an error.
- Nodes only become faulty (for the first time) on the slots of previously non-faulty processes.

It is up to the user to decide whether these restrictions/assumptions are tolerable, but it is only fair to point out that failure to observe them can lead to disastrous behaviour such as all correct nodes wrongly diagnosing themselves as faulty (which occurs if, for example, two consecutive nodes commit broadcast faults). There is some discussion in [10] about the possibility of toughening the protocol up to deal with bursts of faults.

These assumptions were implemented in our model by manipulating the way in which faults occur (the first fault in any process is triggered by an appropriate `faulty.i` event and subsequent ones can occur nondeterministically) and via synchronisation with processes which regulate the global occurrence of these `faulty.i` events relative to other `faulty.j` and `tock`. In the following extract from the CSP code[2] we see the network composed of all the nodes and `FaultMan`, which implements the first, second and fourth of the above assumptions.

```
SYSTEM = ( [| {| agreedfault,broadcast,tock|} |] i:slot @ NODE(i))
         [| {| tock,faulty |} |] FaultMan
```

It is not necessary to distinguish the two different types of fault via different events, since that is apparent from the broadcast slot that the fault occurs in, and the protocol does not rely on (say) a receive-faulty processor only subsequently committing receive faults.

2.2 Checking the TTP: modelling the specifications

To check that the specification property (ii) is met we formulated two processes, *RightLMS* which specifies that the non-faulty nodes diagnose as faulty those nodes that are actually faulty, though perhaps with some delay in diagnosing newly-faulty ones, and *SelfDiag* which specifies that after *size* slots (steps in time) a faulty processor *must* have self-diagnosed itself. Both of these specifications need to be met in order for the protocol to satisfy property (ii).

The first specification is a trace specification, the code is given below. `assert` tells FDR to display the check following it, for ease of use. Note that events which the specification is not concerned with (and so don't affect the outcome of the check) are hidden, the *selfdiag* event is explained below.

```
RightLMS(badslots) =
     let goodslots = diff(slot,badslots)
     within
     agreedfault!{} -> RightLMS(badslots)
     [] faulty?i:badslots -> RightLMS(badslots)
     [] faulty?i:goodslots -> RightLMS'(union(badslots,{i}),i,0)
     [] tock -> RightLMS(badslots)

RightLMS'(badslots,i,count) =
     let goodslots = diff(slot,badslots)
     within
```

[2] The `agreedfault` event on which all of the node processes are synchronising will be explained below.

```
     agreedfault!{} -> RightLMS'(badslots,i,count)
     [] agreedfault!{i} -> RightLMS(badslots)
     [] faulty?j:badslots -> RightLMS'(badslots,i,count)
     [] count<size & tock -> RightLMS'(badslots,i,count+1)

assert RightLMS({}) [T= SYSTEM\{|broadcast,selfdiag|}
```

In order to check for self-diagnosis a *selfdiag* event is used. Nodes are willing to perform a *selfdiag* event when they have diagnosed themselves as faulty; they output their slot number on the event: the specification takes the form of asserting that sufficiently long after a given fault has occurred, these *must* be offered. Below is the code for this failures specification, note that `size` slots after a *faulty* event the process insists on offering the `selfdiag` event of the newly faulty node.

```
SelfDiag(badslots) =
     let goodslots = diff(slot,badslots)
     within
     selfdiag?x:badslots -> SelfDiag(badslots)
     []
     (STOP |~|
     (faulty?i:badslots -> SelfDiag(badslots)
     [] faulty?i:goodslots -> SelfDiag'(union(badslots,{i}),0,i)
     [] tock -> SelfDiag(badslots)))

SelfDiag'(badslots,count,dying) =
     let goodslots = diff(slot,badslots)
     within
     if count == size then SelfDiag(badslots)
     else
     (selfdiag?x:diff(badslots,{dying}) -> SelfDiag'(badslots,count,dying)
     (STOP |~|
     (selfdiag!dying -> SelfDiag'(badslots,count,dying)
     [] faulty?i:badslots -> SelfDiag'(badslots,count,dying)
     [] tock -> SelfDiag'(badslots,count+1,dying)))

assert SelfDiag({}) [F= SYSTEM\{|broadcast,agreedfault|}
```

Verifying that the TTP meets property (i) is achieved by simply checking that the model is deadlock-free. This is because at the beginning of every slot every non-faulty node has to perform a `agreedfault` event, on which it outputs all nodes that have been removed on this turn from the *lms*.[3] The model of the TTP, or the `SYSTEM`, is constructed by synchronising (putting in parallel over) all of the nodes on the `agreedfault` events in addition to the *broadcast* and *tock* events. If at any point in time non-faulty nodes disagreed for the first

[3] This is, of course, not a real communication of the protocol, rather another device we can use to monitor behaviour. A slightly simpler approach is to replace this communication with a synchronisation on the whole value of *lms*. In other words, we might make them agree on the set of correct nodes rather than the set of new diagnoses. The approach with the smaller sets has efficiency and other advantages.

time on who they thought was faulty, the values they output on `agreedfault` would differ and this would cause the `SYSTEM` to deadlock. Faulty nodes still communicate the `agreedfault` event, but are willing to communicate anything on it, so any incorrect beliefs that faulty nodes are ignored.

2.3 The protocol error and a fix

Our analysis of the protocol shows that it does not meet its requirements. *Rapid Self-Diagnosis* says that a newly faulty node will remove itself from its local membership set (and thereby diagnose itself) when the slots of at most two non-faulty nodes have been passed. It is this claim, and proof, that is incorrect, since it is possible for a faulty node to not self-diagnose. This can occur with a receive fault that happens when there are three non-faulty nodes remaining (however many were there at the start). It turns out that instead of realising that *it* is faulty, the node with the fault may decide that the other two are faulty and it is not. The details of why this happens can be found in Appendix B.

There is a very simple fix which can be applied to the protocol: namely that if a processor should believe that less than two processors are not-faulty (i.e. that only itself is non-faulty), it should self-diagnose. This is easily implemented by a processor checking the cardinality of its *lms*, and when this is smaller than 2, self-diagnosing. This bug was independently discovered by N. Shanker in January 1998 which lead to an alternative solution which is presented in [11].

2.4 Bisimulation effects and faster checking

We performed our analysis of this protocol using the FDR refinement checker. It was immediately noticeable that we were finding many fewer states than reported for Murϕ checks of presumably very similar systems in [10]. It is, for example, reported that a six-node system in which three faults are allowed has $\tilde{2}$M states on the Murϕ coding, and represented the limit of what they could check whereas the same system for us had only 7,512. It is our belief that this difference is explained by the factoring by strong bisimulation equivalence that FDR automatically performs on the sequential components of any system it explores before combining them together. The TTP protocol provides a quite unusually good example for this function because of the following observations:

- The correct nodes in the system always agree on the values of the *lms* and *ack* parameters, meaning that they do not multiply up to create any larger a state-space of their combination than any one process.

- On the other hand, once a node has gone wrong it no longer agrees, and so potentially we get many variants of each state differing only in the values of these parameters as held by the faulty processes.

- At any one time there is at most one faulty process that has not self diagnosed.
- The behaviour of a node that *has* self diagnosed is entirely independent of the values of *ack* and *lms* (except, of course, that it depends on its own name not being in *lms*). Thus the strong bisimulation factoring identifies most of the states of the component processes which are the worst culprits in creating states of the complete system.

FDR also provides other compression functions that are applied at the user's discretion (see [16], for example). These have been helpful in some of our codings of the TTP, but this example stands out as benefiting more than any other we are aware of from the automatic bisimulation.

The state spaces of the individual component processes as well as some of the assumption processes and specifications grow like some polynomial in `size` times $2^{\texttt{size}}$, and with significant numbers of nodes the compilation of these component state machines can become a significant part of the effort of the overall FDR run. Almost all of this work can be avoided by factoring these components into parallel sub-processes, for example the *lms* parameter of a node is devolved to a separate slave process, which can if we wish itself be factored into `size` two-state processes. The code of both the optimised and unoptimised versions can be found at the web site discussed earlier.

The system (allowed to run up to $\texttt{size} - 2$ faults) has the following numbers of states and running times on a 266MHz Pentium II laptop:

Number of Nodes	Number of States	Time in seconds	Number of Nodes	Number of States	Time in seconds
6	11622	6	9	252684	239
7	34223	22	10	654070	881
8	94864	77			

3 Towards induction

There are many similarities between how the protocol discussed in the last section behaves with $N+1$ nodes to how it behaves with N nodes. Unfortunately there are a number of reasons why these do not lead directly to an inductive proof on a system like FDR. These largely derive from the fact that the nodes of the larger system are different in significant ways from those of the smaller one, meaning that there is no trivial way of creating the larger one from the smaller one by just adding an extra node. The differences are:

- Nodes calculate whose slot it is by seeing how much time has passed since the protocol started up: they use arithmetic *modulo* `size`.
- Each node knows the identity of all the others, since they all keep a record of which others are good and bad.

So adding an extra node will require the behaviour of the ones already there to be changed in ways that increase their individual state spaces.

As we will see in the next section, there are ways of solving the second of these problems via data independence, but that only makes the first one more troublesome. For one of the primary requirements on a data independent type is that there be no operations on it which, like arithmetic ones, constrain how it can be interpreted. Our first objective, therefore, was to transform our protocol model so that this dependency on arithmetic was removed. This was straightforward once we realised that the protocol does not depend crucially for its correctness on the cyclic distribution of slots, rather on the following:

- All nodes agreeing, on each round, whose turn it is to broadcast.
- A few simple consequences of the cyclic order, such as no node getting to broadcast twice in a row.

We therefore modified our model so that the mechanism for deciding whose turn it is became a globally-synchronised communication `whose?whosego` rather than arithmetic. There is no suggestion that this synchronisation should exist in any implementation; it is simply an abstraction of *any* mechanism by which the nodes agree on the value of `whosego` each turn, just as the `tock` action is an abstraction of how nodes agree on the passage of time. In practice the cyclic order has advantages because of its simplicity and the fact that it minimises the maximum wait any node has between turns, but our generalisation means that we are simultaneously proving correctness of this and any other choice – it might, for example, be possible to implement a mechanism by which failed nodes get removed from the schedule in an agreed way.

Making this change only requires a trivial adjustment to the individual node processes and makes them data independent. Somewhat more complex were the consequences of this transformation on the fault rate assumptions and the specification of how rapidly faults are diagnosed, as both depend implicitly on the cyclic order. What they say, informally, is that each new fault is diagnosed promptly provided that no second new fault is introduced during this process. This notion of "promptly" comes down basically to one round of turns, plus one further turn, in the cyclic case, though careful examination shows that in many cases the diagnosis happens rather quicker; it just happens the above guarantees appropriate sequences of turns will have occurred no matter how the fault appeared.

When we move to the generalised model, we have to do two things:

(a) Identify any sequences of turns, impossible in the original version, that would create pathological behaviour. We can then assume such patterns do not occur either by putting a regulator process in parallel with the network that prevents them, or introducing "don't care" clauses into our specifications when they do.

(b) Identify patterns of turns, guaranteed in the original system by a full cycle plus one, that should guarantee that a new fault has been diagnosed both by others and by itself (has been *fully diagnosed*).

We can then seek to prove that *provided* any patterns of turn that (a) excludes do not occur, then whenever the conditions implied by (b) have occurred a new fault will have been fully diagnosed and the *ack* bit of all remaining correct processors will be *true* (the 'normal' value). Furthermore the non-faulty nodes agree at all times over who they *think* is faulty. It will then follow that if faults are sufficiently infrequent that one does not occur until the previous one has been fully diagnosed, then our system behaves properly as long as it stays within the bounds required. We should re-iterate that this only proves the original, cyclic, protocol works under its own assumptions and specifications if we can show that those of the new version are in the appropriate relationships with them. We will call a state in which (i) all correct nodes have diagnosed all incorrect ones and no others, (ii) all incorrect ones have self-diagnosed, and (iii) the *ack* bit of correct nodes is *true*, a *regular* state.

The new *assumptions* we have identified as necessary are these:

(i) As before, faults cannot manifest themselves for the first time in an already faulty slot.

(ii) New faults do not occur until the network has reached a a regular state.

(iii) No faulty process ever gets a *second* broadcast turn after going faulty before a regular state is reached.

Under these assumptions, a fault which occurs in a regular state, we claim, is always fully diagnosed and the state regular under the following conditions:

(a) A send fault (i.e., a process going faulty in its own slot) is followed by two distinct non-faulty processes (perhaps interleaved with faulty ones – it cannot have another turn itself by assumption (iii) above).

(b) Forgetting the turns of nodes all agree are faulty, when a node goes receive faulty *either* the next slot is one of the faulty node in which case this is followed by two good slots other than the one it went faulty in *or* it is followed by a good slot other than the one it went faulty in, eventually a turn of its own and any good slot.

These conditions are therefore also our limits (thanks to assumption (ii) above) on how rapidly faults can occur.

In the cyclic case where at least three good nodes remain (i.e., avoiding the self-diagnosis error pointed out earlier) the fixed order implies that (a) or (b) occur (i.e., the network becomes regular) by the time $N + 1$ time units have passed and also before the faulty process has had two turns. Therefore the **fault rate assumption** guarantees assumption (ii) above and assumption (iii) is satisfied. Hence correctness of the generalised protocol in the sense implied by the above guarantees correctness of the cyclic one.

Our assertions about how the protocol behave still require proof. For a fixed `size` it is straightforward to turn our assumptions about turn allocation and fault occurrence into regulator processes which run in parallel with the network (initialised with all nodes working), rather as we did in the cyclic case. We can then check that it satisfies our various requirements.

It is interesting to notice an advantage we gain at this stage from our re-coding to eliminate the cyclic turn allocation. This is the fact that even when a few nodes have gone faulty, it is completely symmetric which node goes faulty next. This is not nearly so clear in the cyclic case, since you could imagine that it was significant what position in the cycle of the new fault was relative to the others. It is easy to argue that any misbehaviour of the protocol would, in the new version, also be possible with the faults occurring in the order `faulty.0`, `faulty.1`, `faulty.2`... This is an example of *symmetry elimination* in the style of [4, 5, 9] which has a tremendous effect on the state space of the resulting check: for example it brings a six-node system down from 710,104 to 18,767 states. (Note that this is still more states than the cyclic protocol; we believe that this is caused chiefly by the constraint proceses implementing the assumptions about fault and turn order.)

4 Data independent induction

Data independence is a highly complex and technical subject, and we do not have space for more than a brief summary here. An accessible introduction (as it applies to CSP) can be found in Section 15.2 of [16], but the main reference is Lazić's thesis [13] where the details are revealed. Data independence in type T is very similar to the idea of *polymorphism*: a program can carry out operations on members of T only when these operations do not look inside them. It can input and output members of T and hold them as parameters, tuple them and include them in lists, but cannot apply operations which constrain what T is such as addition, or which represent mappings between T and other unrelated types. A data independent program must make sense for any nonempty type T. We also need restrictions on the use of indexed operations over T – the only standard CSP operator that may be indexed over T is nondeterministic choice – for other ones provide subtle ways of counting T, something which has to be avoided if we are to get the benefits data independence can bring.

Finally, the ways in which the control flow of a program can be affected by members of T are also highly constrained. Life is easiest (and we get the tightest results) when they are simply passed around dumbly and do not affect control flow at all. At present the theory only allows two ways in which they can affect it:

- Equality tests between members of T (as opposed to order comparisons, which are not allowed).

- The application of mappings from T to some fixed finite type, often $\{true, false\}$ in which case they are predicates. These mappings are constants: they are not arrays which can be changed during the run of a program.

The latter of these was introduced, together with the use of constant symbols representing members of T, in [12].

The usual motivation for data independence analysis is proving that a result holds independent of what T is used. We have two main techniques for these proofs: analysing programs via a *symbolic operational semantics* in which values of type T are stored as symbols rather than as actual values, and the calculation of *thresholds*. A threshold is a size of T beyond which the result of a check (that a data independent implementation meets a data independent specification) is unvarying. Thus if a check has a finite threshold, it is potentially verifiable for all T on a model checker. There are an increasing number of results, many of which can be found in the works cited above, that allow thresholds to be calculated for checks of CSP proceses. The existence and sizes of these thresholds depend on a number of factors such as:

- Whether and how T affects control flow (when it does not, the threshold is usually one or two).
- The extent to which the specification constrains what values of T the implementation communicates.
- The subtlety of nondeterminism in the specification.

The generalised version of the protocol created in the previous section is a data independent program, even including the processes introduced to implement the rules and assumptions regarding turn allocation and fault occurrence. Except for one thing: it uses parallel composition indexed over the supposed data independent type. Therefore the theory of data independence does not give us any threshold for the checks of the protocol as developed so far.

So suppose we have a system of the form `|| x:T @ [A(x)] P(x)` that is data independent except for the indexing of the outer parallel composition over `T` (i.e., the `P(x)` and `A(x)` are both created data-independently in `T`, though they can use the indexing value `x` as a constant). It makes no sense, either in general or in our example, to try to characterise the behaviour of this system for `T` of size $N+1$ by adding an extra node onto the complete system built for `T'` of size N. For the `P(x)` use the complete type in various ways, such as stored in state identifiers. On the other hand, if we fix `T`, we can do induction on the size of a partially constructed system

```
SubSys(S,T) = || x:S @ [A(x)] P(x)
```

where `S` is a subset of the type `T` (`T` still being the type used in defining `A(x)` and `P(x)`).

The induction will take the form of showing that, for all nonempty subsets `S` of `T`,

$$\texttt{Hypoth(S,T)} \sqsubseteq \texttt{SubSys(S,T)} \qquad (*)$$

where `Hypoth(S,T)` is a data independent program in which `S` is treated as a predicate symbol (the predicate mapping a member `x` of `T` to true if it is in the set). What `Hypoth(S,T)` should do is capture some property of how the subsystem behaves in such a way that `Hypoth(T,T)` captures something we want to prove about the complete system. The base case of the induction is to show that

$$\texttt{Hypoth(\{c\},T)} \sqsubseteq \texttt{P(c)} \qquad (\dagger)$$

for a constant symbol `c` representing an arbitrary member of `T`.

For the step case we show that for any nonempty `S` and constant `c` outside `S`,

$$\texttt{Hypoth(S',T)} \sqsubseteq \texttt{Hypoth(S,T)[AA(S)||A(c)]P(c)} \qquad (\ddagger)$$

where `S'` is the union of `S` and `{c}` and `AA(S)` is the union of `{A(x) | x <- S}`.

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

$$\begin{array}{lcl}
\texttt{Hypoth(S',T)} & \sqsubseteq & \texttt{Hypoth(S,T)[AA(S)||A(c)]P(c)} \\
 & \sqsubseteq & \texttt{(|| x:S @ [A(x)] P(x))[AA(S)||A(c)]P(c)} \\
 & = & \texttt{(|| x:S' @ [A(x)] P(x))}
\end{array}$$

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

The crucial thing about the lemma $(\ddagger)$ is that it contains no indexed parallel operators: both sides are data independent under our assumptions about the various sub-terms, and contain a predicate symbol `S` and constant symbol `c` in addition to any others there may be (`S'` is simply a formal combination of the two of these). We can therefore hope that it may be provable independently of what `T`, `S`, `c` and any other parameters may be. Since the base case $(\dagger)$ is also potentially provable by the same methods, we might be able to prove that $(*)$ hold *for all* `S` and `T` via a finite number of finite checks. This inductive structure, together with others ways of exploiting data independence to justify the use of model checking in inductions, will be explored further in [14].

This new method gives us hope, as unlike traditional data independence and inductive arguments it can be applied to the generalised TTP network. Along with this hope, however, quickly comes the realisation that the new method requires considerably more ingenuity to use than traditional data independence. We have to create the process `Hypoth(S,T)` which must both be strong enough, when `S` = `T`, to imply whatever we are trying to prove of our network, and must inductively capture this property for the cases where `S` is a proper subset – probably a good deal less intuitive than the "finished" case.

There is no need, in general, for the behaviour of `Hypoth(S,T)` to be very close to that of the partial system `|| x:S @ [A(x)] P(x)`: it may be a significant anti-refinement, designed to prove that the complete system refines some general form of specification that refers, for example, only to a small subset of the system's alphabet. This can, as is shown in [14], give us freedom to prove things about systems for which there is no data independent process that behaves like the complete system. However, in the case of the TTP, it *is* possible to create such a process, at least provided we build in the assumptions that we have introduced about turn allocation and fault occurrence. The reason for this is the great coherence of the protocol: correct nodes agree on the values of their state variables, and the state variables of faulty self-diagnosed ones are irrelevant (except for the fact they are self-diagnosed). Therefore we can capture the essential features of the behaviour of a subset `S` of the nodes if we know:

- Whose turn it is (all nodes agree on this).
- Which nodes the good nodes have diagnosed as faulty (`bads`).
- These same nodes `ack` bit.

and, if there is a node in `S` that is faulty but not completely diagnosed,

- Which node it is (`dying`).
- Who it believes to be faulty (`bads'`).
- Its ack bit (`ack'`).

There will never be two that are faulty and not completely diagnosed, because of our assumptions.

We therefore built a process, termed `Superstate` (as opposed to `Hypoth`) and with the parameters above, to reflect the behaviour of the subset of nodes. For example it was responsible for broadcasts when the current turn is a member of `S`, but listens to broadcasts when this is outside. This development essentially involved analysing what the subsystem would do in the various cases that could arise. It could not have been done in a reasonable amount of time without the use of FDR or some similar tool with excellent debugging feedback, since the only realistic way of finding and analysing these these cases is to test the emerging description against what it required of it. It would be fair to say that this was an iterative development until we had a definition that met the various requirements we had of it, namely (†) and (‡) with `P(c)` replaced with `Node(c)` from the protocol, and that it met the specifications we wanted to prove of the protocol.

The assumptions, and to some extent what to do about any improper behaviour of "nodes" outside `S`, are dealt with in `Superstate` by making it behave like `BOT`, a process refined by all others, if a violation occurs. This has the effect of saying we do not care what the system would do in these circumstances.

5 Finite thresholds

While the `Superstate` and `Node` processes are both data independent, data independence results do not give us finite thresholds for the checks (†) and (‡). The reason for this is the fact that the processes involved possess variables like `bads` and `bads'` which represent sets of nodes. For in cases where, as in this protocol, the processes being checked involve equality tests between members of data independent types, the thresholds given by Lazić's results contain a component which is the maximum number of values of the type that the specification and implementation hold in variables at one time. Clearly the existence of `bads` means that there is no such maximum that is independent of the type of nodes.

There is no easy way round this if we want to check in a single refinement relation the complete history of the protocol as it evolves from having all processes working to a state where many are broken. What we can do instead is to concentrate on proving two things: that a regular state will remain regular as long as no new faults occur, and that if a single fault is injected into a regular state then this fault will be completely diagnosed and the state again become regular within the bounds we have set. Clearly, since we are forbidding further new faults from occurring within these bounds, these two facts imply the correct behaviour of the protocol throughout its history.

This is done by examining a version of the system in which the unbounded set variables have been replaced by a combination of a predicate symbol, representing the *already diagnosed* nodes in some regular state, and a set variable representing newly diagnosed nodes. (Bear in mind that the great difference between a predicate symbol and a set variable is that the latter can vary while the former is a constant during a run. This helps to explain why we can get finite threshold for checks involving arbitrary predicate symbols, but not set variables.) Our aim will be to run checks in which the new set variables are bounded in size.

This replacement has to occur in both the `Node` and `Superstate` processes. This creates the following processes:

- `IndNode(i,bads, newbads, ack)`: node `i` which was initialised with the information that the set of nodes satisfying the predicate `bads` are faulty, and has additionally diagnosed the set `newbads`. The introduction of one fault should make this set no bigger than two. In any state where `i` is a member of either `bads` or `newbads` this reduces to the simpler form below.

- `IndSDNode(i)` a simplified representation of a node that has already self-diagnosed.

- `IndSuperstate(S,bads,newbads,ack)` in which there is no 'dying' node, and all the good members of `S` agree on the diagnosis of `newbads` (no bigger than one with one new fault) and the value of `ack`.

- `IndSuperstate'(S,bads,newbads,ack,dying,newbads',ack')` in which node `dying` has committed a fault, believes the nodes in `newbads'` are bad, and has ack bit `ack'`. We can limit `newbads` and `newbads'` to sizes one and two respectively.

It is now possible to prove, by an inductive structure effectively the same as that described earlier, that

```
       IndSuperstate(T,bads,{},true)[|{|faulty|}|]OneFault'
                                   ⊑
(|| i:T @ [A(i)] IndNode(i,bads,{},true))[|{|faulty|}|]OneFault
```

and do so in such a way that the set parameters are bounded, leading to finite thresholds, or alternately to checks which would run finitely on a symbolic model-checking program. The same is then true of checks of the assertion that `IndSuperstate(T,bads,{},true)` meets our requirements of agreement and fault diagnosis.[4]

Theorems which generate threshold values for checks with equality checks usually give extremely conservative values, because they have to deal with pathological cases, for example in which the specification and implementation are holding disjoint subsets of T in their variables. Certainly the thresholds they give for the present case are unworkably large, but special case analysis can bring down the values to reasonable levels: the biggest ones of the checks we need can be argued down to no more than 11, and can almost certainly be brought down further. (It should also be pointed out that thanks to the limitation to a single additional fault, the state spaces of such checks are very much smaller than the complete protocol of the same size.

This is one case where we believe that the symbolic execution of checks would have an enormous advantage over threshold calculations: it would not be necessary to do these special case arguments, since the benefits of them would drop out for free. Aside from this, the precision of the symbolic checks should cut out large classes of unnecessary and equivalent states that are generated by any threshold, however tight.

6 Conclusions

This case study has illustrated what can be achieved by the structured use of model checking and associated theories, and besides that provided some fascinating insights into the verification process in general.

The fact that we were forced to generalise the protocol in order to apply our inductive techniques both brought extra insights into the protocol – clearly a positive thing – but also required a fair amount of ingenuity. In order

[4] As the reader will find if he or she consults the citations on data independence, it is also necessary that in all cases the various processes involved in the check meet other technical conditions that we do not have space to detail here.

to make these techniques usable by a wider community it will be necessary either to systematise this process or to remove some of the restrictions that made it necessary. One ca, of course, make the same observations about the construction of the data independent inductions.

It can reasonably be argued that the stepwise proof we ended up with (that a single error is dealt with before the next arises) meant that what we really did was to use the model checker as a tool to verify lemmas that were needed in a larger proof. We are interested in how this process might occur automatically, with a theorem prover outputting lemmas for checking by the likes of FDR.

Acknowledgements

We are extremely grateful to the following people: Peter Ryan for suggesting the TTP as an example, Ranko Lazić for his work on data independence, Gavin Lowe who experimented before us with using data independence to support induction (though in different ways) and Joy Reed for showing us that induction is usable on practical protocol examples. John Rushby gave us useful insights into the work reported in [10] and Friedrich von Henke into the TTP.

Appendix A: details of the protocol

This gives those details of the protocol from [10] that were not set out earlier.

Each processor has a *local membership set* (*lms*), that initially contains all processors and an *ack* bit (of type bool). Processors will only attempt to broadcast if they are contained in their own *lms*. Processors that suffer send faults are unaware of the fact and assume that they have broadcast correctly. The broadcast of a processor contains its local *ack* bit.

Let the auxiliary predicate *arrived(b,p)* be defined as *true* in a step if and only if processor p receives a broadcast from b, and b is the expected broadcaster in that step. This predicate can be considered local to p because that processor can sense the non-receipt of a broadcast.

- For each processor p, if the current broadcaster b is not an element of it p's *lms*, none of the local variables are changed.
- If p is the broadcaster b and is in its own *lms*, it broadcasts $\mathbf{ack}(b)$ and then updates $\mathbf{ack}(b)$ to *true*.
- Otherwise, when p is not the broadcaster b, each field of p is updated as follows (notice that $\mathbf{ack}(p$ is a local variable of p, and that $\mathbf{ack}(b)$ is provided in the broadcast received from b).
  - Updated *lms*: same as previous *lms* except for the possible removal of p and/or b.

    p is excluded in two cases:

(a) (NOT *arrived(b,p)*) AND NOT **ack**(p), or
(b) *arrived(b,p)* AND **ack**(b) AND NOT **ack**(p).

b is excluded in the two cases:

(c) NOT *arrived(b,p)*, or
(d) **ack**(p) AND NOT **ack**(b).

– Updated **ack**: set to
arrived(b,p) AND (**ack**(b) OR NOT **ack**(p)).

Appendix B: self-diagnosis fault with three nodes left

Consider the case when there are three non-faulty nodes remaining in the cyclic TTP network. Suppose that in cyclic order these nodes are 1, 2 and 3. If one of these nodes were to suffer a receive fault then contrary to the claims made, it may never self-diagnose itself as faulty. This is because when the node, say 1, suffers a receive fault as 3 broadcasts it wrongly diagnoses the 3, as send faulty and removes it from its *lms* (and sets its *ack* bit to false). If 1 broadcasts next (amongst the nodes not already agreed faulty), it sends its its *ack* bit (false) (and sets its *ack* bit to true since nodes always assume that they have successfully broadcasted in their own slot). 2 and 3 will diagnose 1 as faulty since its broadcast disagrees with their own *ack* bit, and they will remove 1 from their *lms*'s. 2 will then broadcast its *ack* bit which will be false. This contradicts what 1 believes. 1 will not know whether it has suffered a fault or 2 has suffered a receive fault, and it removes 2 from its *lms*. This would be correct if 2 is faulty, and it would not matter if it were 1 that was faulty since faulty nodes can believe anything since they are ignored. 1 requires the next broadcast to be able to decide. But since there are only three nodes left 1 requires the next broadcast of 3 to self-diagnose. However, 1 will actually ignore the broadcast of 3 since it has removed 3 from its *lms*. The next broadcaster will be itself (1), since all nodes assume that their broadcasts are successful 1 will set it's *ack* bit to true on this turn and will not self-diagnose. At this point 1 actually believes that it is the only working node, and since it will therefore ignore the broadcasts of all other nodes it will never diagnose itself as faulty.

The 'proof' of *Theorem 3 (Rapid Self-Diagnosis)* presented in [10] is incorrect: it makes the simple but understandable error of forgetting that the faulty process may have misdiagnosed another correct process and thus ignore what it says.

References

[1] Sadie Creese, *An Inductive Technique for FDR*, Master's thesis, Oxford University, 1997.

[2] 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.

[3] D. Dill, *The Murϕ verification system*, Proceedings of CAV'96, 390-393, LNCS 1102, July/August 1996.

[4] E.A. Emerson and A.P. Sistla, *Utilizing symmetry when model checking under fairness assumptions: an automata-theoretic approach*, Proceedings of the 7^{th} CAV, 309-324, LNCS 939, 1995.

[5] E.A. Emerson and A.P. Sistla, *Symmetry and model checking*, Formal Methods in System Design, **9**, 105-131, 1996.

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

[7] Y. Gurevich and R. Mani, *Group Membership Protocol Specification and Verification*, Specification Methods for Programming Languages and Systems, Oxford University Press, 1994.

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

[9] C.N. Ip and D.L. Dill, *Better verification through symmetry*, Formal Methods in System Design, **9**, 41-75, 1996.

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

[11] Updated version of [10] to be found at www.csl.sri.com/wdag97.html.

[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, to appear in 1998.

[14] G. Lowe and A.W. Roscoe, *Dataindependent Induction*, to appear, 1998.

[15] A.W. Roscoe, *Model-Checking CSP*, in 'A classical mind, Essays in honour of C.A.R. Hoare', Prentice-Hall, 1994.

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

[17] J. Rushby, *Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms*, Expanded version of a paper from DCCA-6: the Sixth Working Conference on Dependable Computing for Critical Applications, 203-222, Garmisch-Partenkirchen, Germany, IEEE Computer Society Press, March 1997.

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

[19] 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.

[20] H.-C. Yen, S.-T. Jian and T.-P. Lao, *Deciding Bisimulation and Trace Equivalences for Systems With Many Identical Processes*, 445-464, Theoretical Computer Science 170 (1-2), 1996.