Data Independent Induction: CSP Model Checking of Arbitrary Sized Networks

Sadie Creese
Worcester College

Trinity 2001

Submitted in partial fulfilment of the requirements for the Doctor of Philosophy in Computation

Oxford University Computing Laboratory
Programming Research Group

Abstract

The challenge for researchers in the formal methods community is to produce formal techniques which industry can use, in terms of people being able to utilise the methods, and in terms of those methods delivering useful results for industrial systems. The formal verification of systems of realistic size and complexity in itself requires the help of computers. This type of tool support usually takes the form of either theorem proving or model checking. Model-checking is an important formal technique, since it is a form of tool support not only capable of positive verifications of system specifications, but also because where proofs fail it provides de-bugging information. This information makes model-checking applicable throughout the development life-cycle, since it can help remove errors early in system development. However, model-checkers can only reason about systems of a finite size, making it impossible to model check an network of arbitrary size directly. The Data Independent Induction technique presented in this thesis extends the classes of systems which may be reasoned about using a model checker. Its particular contribution is to allow us to prove things about arbitrary-sized networks where each node knows and uses the type of node identities.

The techniques presented in this thesis use the process algebra CSP and the model checker FDR. Chapter 1 is an introduction; it presents the motivation for the work, the CSP formalism, data-independence and induction. Chapter 2 is an example of a structural induction technique for CSP and FDR. In Chapter 3 the basic method of Data Independent Induction is presented. Included in this chapter are two small examples: a distributed token-passing system incorporating leadership election, and a communications protocol involving the multiplexing of channels. In Chapter 4 and Chapter 5 large case studies are considered. Chapter 4 demonstrates a proof of the Time Triggered Group Membership protocol for totally connected networks of arbitrary size. Chapter 5 presents a proof of a QoS resource reservation protocol for arbitrary broadcast trees. In Chapter 6 the method is extended to more structured network topologies, and is demonstrated for arbitrarily large rings. Chapter 7 discusses how to reason about toroid. Finally the conclusions and discussion on future research directions are discussed in Chapter 8.

Thanks

Thank you to my supervisor Bill Roscoe for giving me the opportunity to work with him, for his patience, for his intuition and for his friendship. Thank you Michael Goldsmith for being a constant source of insight into CSP, FDR, and the finer eating and drinking establishments of Oxford. Thanks to Ranko Lazić for his work on Data Independence. Thank you to the whole team at Formal Systems (Europe) Ltd, past and present, for providing a level of support second to none. Thank you to Colin O'Halloran, Zak, Nick Moffat and Peter Ryan for insight, informing discussion, case studies and many Thai lunches. Thanks to Joy Reed for inspiration and confidence, whose supervision during my MSc was the beginning. Thanks also to Carroll Morgan and Jeff Sanders for constructive criticisms.

During my time at Oxford I have made many friends, so, in no particular order, thank you Jim Davies, Jim Woodcock, Sam, Mark, Jude, Elliot, Alex, Sarah, Lavonne, Phil, Nick, Alvin, Andy, Chelsea, Dan Lunn, Silvija, Corrina, Karina, Ben, Dominic, Richard, Dan, Jon, Chris, Paul, Matt, Craig, Tom, Christie, Phillipa, Jo, Jonny and Rupert. Thanks to Worcester College for being such a perfect home, and to Tasha and Chris for feeding me in my hours of need!

I can't find the words to thank my mum, dad and grandparents enough for their support throughout my life as a student. Their love, respect and constant belief in me have been a pillar of strength. I couldn't have done any of it without them.

Thank you to the EPSRC and The Defense Evaluation and Research Agency (DERA) for financially supporting my studies.

Finally, thank you Stuart for your love, support, patience, and for putting up with me during this last year :-)

To Nanny

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

It is desirable that any computer system meets its specifications, and critical for some applications. Formal methods offer a route to developing high integrity computer systems through the rigour of mathematical proof, but it is still not standard industry practice to employ them. Such methods require a level of expertise often perceived as taking too much time and money to develop. This situation is exacerbated by the fact that, whilst these methods work well on small textbook examples, results cannot be guaranteed for systems of realistic size. But the complexity of computer systems is increasing, in functionality, size and topology, which in turn leads to a greater chance of error. Fierce competition results in time to market being drastically reduced, creating a tension between the needs for correctness and rapid delivery. In order to meet the demands of industry, techniques are required which bring results for large and complicated systems quickly. This inevitably requires as much automation as is possible, not only for speed, but also to limit the chance of human error[1] in the verification process.

Industry has traditionally employed simulation and testing in its verification of computer systems. Whilst these approaches are proven to identify errors/bugs, they are not exhaustive; unless you have simulated every situation, or tested every configu-

[1] In [70] John Rushby provides an excellent discussion on the need to automate and how it might best be achieved.

ration, you simply cannot be sure that your system will always behave correctly. In contrast, formal verification techniques can offer exhaustive analysis (equivalent to 100% coverage testing) which has not only led to the discovery of errors previously undetected, but also provides a guarantee of correctness.

The formal verification of systems of realistic size and complexity in itself requires the help of computers. This type of tool support usually takes the form of either theorem proving or model checking. A model checker can automatically check whether a given specification *Spec* is satisfied by a proposed implementation *Imp*, and if the answer is negative it can provide information on the behaviour which lead to the potential violation. Example model checkers are FDR[30, 74], Spin[38, 39, 40], Murϕ[26], SMV[56] and PRISM[62] (unique in that it can handle probabilities). No human interaction is required in the checking of the model; however, model checkers are based on exhaustive exploration of state spaces and as such require that *Spec* and *Imp* are limited to being only finite state. This means that it is impossible to model check an unboundedly large system directly[2].

In fact, for many complex systems the sizes of models which can be reasoned about with a model checker are relatively small. In systems constructed of the parallel composition of communicating processes the number of states of the system model may be exponential in the number of component processes. The number of states also frequently grows quickly with the sizes of data-types used in programs. This behaviour is referred to as the *state explosion problem*, and the challenge to verify such parameterised systems the *Parameterised Verification Problem* (PVP).

There are various techniques which are employed to alleviate the state explosion problem. These include abstraction techniques, partial order techniques, compression techniques, Binary Decision Diagrams (BDDs) and compositional reasoning:

Often a system is designed to operate over a large data set, but the behaviour of the system at any point in time is dependent only on the values of a small subset of this set. In such cases data abstractions can be created which are usually specified by mapping the actual data values to a small set of abstract data types. The size of the set of abstract data types depends on the maximal number of different values of the data type which are required to capture all possible values of the system at any one time. Theories of data independence implement a form of data abstraction.

[2] See [16] for a detailed discussion on the theory of model checking.

Other abstraction techniques focus on the system as a whole. They aim to find abstractions of the system which have a smaller number of states, but are strong enough to imply the desired property. However, finding such abstractions is not trivial and is, in itself, the focus of much research, see [14], [25] and [50].

Partial order techniques reduce the size of system to be traversed in a different way. If the same global state is reached regardless of the ordering of a set of events, then in certain cases it is only necessary to consider one ordering. In other words, provided that the ordering of intermediate states and any visible events that might occur are irrelevant to the specification property under consideration. By breaking the symmetry and fixing the order we discount other possible interleavings, thus significantly reducing the state space. See [16] for a thorough discussion on both abstraction and partial order methods.

Compression techniques work by finding smaller representations of a system without changing its meaning. So the denotational value of a process remains fixed while the operational or algebraic form would change. In [66] there is some discussion on compression techniques with respect to the process algebra CSP, and their implementation in the model checking tool FDR. Whilst this description is clearly focused on a particular formalism the ideas contained within it remain of a general nature.

BDDs are a form of symbolic representation for state transition graphs which work by replacing many states by a few which represent groups of related states. Again, the state space of the system to be reasoned about is reduced. In [48] Pnueli *et al* extend symbolic model checking to facilitate reasoning about systems with infinitely many states. However, they could only consider safety properties (not liveness properties), and systems with totally connected or tree topologies. Compositional reasoning verifies components of a system separately, but allows assertions for the whole system to be made ([5] and [49] are two approaches).

None of the techniques described above directly facilitate the model checking of an infinitely large system, but they do make a significant difference to the size of model which can be checked.

Whilst model checking meets our requirement for minimising human interaction, limits are put on the size and complexity of the system we can analyse. Establishing a system correct for a certain size/configuration does not necessarily mean it will be correct for any arbitrary configuration. In contrast, a theorem prover [31] can be used to prove that *Spec* is satisfied by *Imp* for all instantiations of its parameters (also known as *uniform verification*). Unlike model checking, the burden of proof is shared between the computer and the user. For complex (realistic) systems the user usually has to provide the difficult and inventive guidance for tactics, and the computer performs the simpler routine steps automatically. When a proof is not possible the tool will often not produce information relating to why. Whilst a theorem prover is capable of facilitating results for very large systems, it is costly in time and effort. Example Theorem Proving tools are HOL[37] and Coq[23]. See [8] and [31] for detailed discussion on automated deductive reasoning, and [10] and [72] for an overview of techniques designed to improve the proof process.

There is currently a large research effort being directed towards combining model checking and theorem proving. The aim is to decompose the verification in such a way that the benefits of both approaches are preserved, whilst compensating for the deficiencies of the other. In general this means utilising model checking wherever possible so maximising automation. A verification takes the form of a proof which is established using a theorem prover, using a model checker to discharge proof obligations appropriately. A general architecture for tools which use both model checking and theorem proving is given in [71]. Examples of this type of verification technique utilising the PVS tool[63] can be found in [68, 69] and [75, 76, 77]. Other tools include The Symbolic Analysis Laboratory(SAL)[78] and Symbolic Model Prover(SyMP)[79]. In [11] the theorem prover Coq and the model checker SPIN are used to verify a broadcasting protocol. A different but related approach is that of Chou and Peled described in [24]. Here they use the theorem prover HOL to verify a partial order method utilised in SPIN. The disadvantage of all of these combination methods is that they still rely heavily on labour intensive theorem proving.

The work presented in this thesis is motivated by the need to extend the classes of systems which may be reasoned about using a model checker, in particular, to include certain classes of systems whose parameterisations might lead to unboundedly large state spaces. Where this is possible we enable the benefits of using a theorem prover to be gained in a model checking environment. This is an important area of research for systems which are designed to be scalable, such as distributed systems and communication protocol networks. These systems are increasingly depended on in critical applications, are typically parameterised by size, topology and stored data types, and should behave correctly no matter what values these parameters take. In the techniques that follow the formalism used for describing systems specifications and implementations is the process algebra CSP[35, 66], and the model checker tool used to help reason about them is FDR[30, 74].

1.2 CSP and CSP$_M$

CSP is a process algebra which is useful for describing systems that interact by communication. A system is modelled as a *process* (itself possibly constructed from a collection of processes) which interacts with its environment by means of atomic *events*. Communication is synchronous; an event takes place precisely when both the process and the environment agree on its occurrence. The syntax of CSP provides a variety of operators for modelling processes, and the associated algebra provides rewrite laws. Primitive operators include process prefix, sequential composition, deterministic and non-deterministic choice, parallelism, hiding, recursion, deadlock and successful termination:

$$\begin{aligned} P \ ::= \ & a \rightarrow P \mid P \,; P \mid P \,\Box\, P \mid P \sqcap P \mid P \parallel P \mid P \,|||\, P \mid \\ & P \setminus b \mid \mu X \bullet F(X) \mid STOP \mid SKIP \end{aligned}$$

The collection of mathematical models and associated semantics that make up CSP facilitate the capture of a wide range of process behaviours. The traces model captures the observable traces of events which a CSP process might exhibit. The failures model captures not only the traces of a process, but also those events it can refuse to perform after a particular trace. The failures divergences model also captures information about events which a process may diverge on (perform infinitely often) from a particular state.

The theory of refinement in CSP allows correctness conditions to be encoded as refinement checks between processes. If process P refines process Q, then all of the possible behaviours of P must also be possible behaviours of Q (although Q may also possess many other behaviours). Therefore, P is a correct, and more deterministic, implementation of Q. This notion of refinement holds for all three of the semantic models, where the possible behaviours of processes are interpreted in terms of the semantic model under consideration.

Refinement is transitive:

$$R \sqsubseteq S \wedge S \sqsubseteq T \Rightarrow R \sqsubseteq T$$

If process R is refined by process S (written $R \sqsubseteq S$), and S is refined by T then R is refined by T. All CSP operators are monotonic with respect to refinement:

$$R \sqsubseteq T \Rightarrow C[R] \sqsubseteq C[T]$$

where $C[.]$ is a context built from CSP operators and constants. This facilitates compositional development of systems. Imagine we want to prove:

$$Spec \sqsubseteq C[System]$$

we can break this into two parts: Find a process which does refine the desired specification:

$$Spec \sqsubseteq C[P]$$

and prove that:

$$P \sqsubseteq System$$

By monotonicity:

$$C[P] \sqsubseteq C[System]$$

and by transitivity:

$$Spec \sqsubseteq C[P] \wedge C[P] \sqsubseteq C[System] \Rightarrow Spec \sqsubseteq C[System]$$

The FDR tool takes a machine-readable dialect of CSP$_M$ as its input syntax, and can be used to check refinements as well as determinism, deadlock freedom and livelock freedom of processes. CSP$_M$ is the combination of a rich data language, based on functional programming, and the CSP process algebra. The various CSP$_M$ operators which are used in this thesis are given in Appendix A. See [30, 66, 74] for detailed descriptions of CSP$_M$.

1.3 Data-independence

Two well known approaches to the verification of a system under parameterisations that lead to unboundedly large state spaces are induction and data-independence. The main topic of this thesis is combining them. Induction will be discussed in Section 1.4.

A system is said to be data-independent of a data-type when its behaviour does not depend on particular instantiations of the type (except that the type must be non-empty). This means that the system must treat the type in a relatively simple way: it may assign, input, nondeterministically choose, and output members of the type, and it may perform equality testing on any two values of the type. It must not be possible for the system to extract information from the data-type in other ways that could then be used to influence its control flow. Many real systems are data-independent: communications protocols are data-independent of the type of data they pass around; their behaviour is entirely independent of the data itself. Memory or database systems may be data-independent with respect to the type of data they store, and the type of addresses used to refer to areas of memory.

Data independence ideas have been developed for a number of notations (see [36, 43] for examples), having first been studied formally in [82]. In this thesis we use here the theory of data-independence developed for CSP by Lazič and Roscoe (see [51, 54] and Section 15.2 of [66]). 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 refinement checks with finite instances of the types. Lazi‘c and Roscoe have produced theorems which 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. See [66] Section 15.2 for an example: a proof of the correctness of a cache.

In addition to the restrictions placed on the treatment of values of the data-type, the data-independence theorems place an additional restriction on the specification process (property that the implementation is checked against). Namely, not only must it be data-independent in the type under consideration, but it also must satisfy the condition **Norm**: The condition **Norm** removes any ambiguity about what state the specification is in after any given trace. A specification process meets this condition if:

- its definition contains no hiding or renaming
- the only parallel operators it may contain are alphabetised parallel (where processes synchronise on all common events), and its replicated version;
- its definition contains no indexed nondeterministic choice whose indexing set depends on T, other than the nondeterministic offering of communications over a channel;
- ($\star$) all internal and external choice constructs have the initial events of each argument disjoint (i.e. once the event has occurred we must know which branch was chosen) and,
- the use of the process composition (;) and time-out (/\) operators in its definition do not break the previous condition ($\star$).

In most cases specifications will naturally meet **Norm**, and those that don't can easily be transformed into a process that does[3].

The techniques presented in this thesis require that data-independent processes also are able to distinguish particular members of the type, and partition the type into finitely many subsets. The theory of data-independence for CSP was extended to enable threshold calculations to be made on refinement relations between such processes. The extension of the theory of data-independence to enable threshold calculations for such systems is detailed in [53]. The theorems put some constraints on the use of predicates and constants, namely, that a predicate must be uninterpreted, in that it should be treated as a symbol, and a verification should establish a property for all possible interpretations. Also, the predicate must be a function into a fixed finite type. A constant is assumed to take one value of the type, and for that value to remain fixed throughout the lifetime of the process/programme.

The basic rules for constructing a process data-independent in the type T, which may contain predicates on T and constants identifying members of T, then become:

[3]This definition of **Norm** is taken almost directly from [66].

- The only operations which may involve values of type T are those provided by the language constructs: equality test, construction of tuples, selection from tuples and list formations. In addition, any operations which are built out of these are allowed. Essentially, operations are allowed if they do not interpret members of T in any way other than checking equality between two members.
- Concrete values from T may be present in the form of constants. There must only be a finite number of such constants.
- The process may not perform operations on T, or any set involving set T, that enable the size of T to be determined. This prohibits the use of operations such as the function `card(set)`, which determines the cardinality of the set. Any other operations on T, or any set involving set T, are permissible.
- No replicated constructs, except nondeterministic choice, indexed over the type T may appear in the process. For example, the use of replicated parallel composition indexed over T would provide a means for determining the size of T.
- A process is data-independent in the type T with a predicate symbol d if d is an uninterpreted function symbol from T into a type U which is fixed, nonempty and finite.

It is assumed in [53] that there is only one free type variable T. Processes containing more than one may be analysed one at a time by substituting fixed data types for all data-types other than T.

Two theorems are given in [53] which enable threshold calculations to be made for refinement relations involving processes containing predicates and constants (constructed as above). These theorems were obtained by extending the original threshold collection theorems 5.3.6 and 5.4.7, found in [66]. The first, **Theorem 6.1**, holds when there are no equality tests between members of the type, and the specification satisfies **Norm**. It states that threshold is the size of the range of the predicate; this holds for all three semantic models.

However, most examples we wish to consider (certainly all those examined in this thesis), *do* perform equality tests between values of T. In this case **Theorem 6.2** provides the threshold. This second theorem assumes that the specification process satisfies a weaker condition than **Norm** – but it is sufficient for our purposes that we take this to be **Norm**. It provides a means for calculating thresholds when T is at least 1 when working in the traces model, and 2 otherwise. The size of the threshold is the sum of three values: the largest number of distinct values of type T that a normalised version of the specification ever has to store for future use; the largest number of such values that the implementation process ever has to store for future use, and the largest number of such values that can ever be either input or nondeterministically chosen in a single event of the implementation.

Let B be the threshold, then **Theorem 6.2** states that B can be calculated as follow:

$$B = W^{Spec} + W^{Impl} + max(L_?^{Impl}, L_\sqcap^{Impl}), \tag{1.1}$$

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. $L_?^{Impl}$ is the largest number of values of type T that can be input in any *single* visible event of the Implementation. $L_\sqcap^{Impl}$ 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.

1.4 Induction

A large number of authors have demonstrated that induction is a method which can be successfully used in the analysis of distributed systems. In 1989 Kurshan and McMillan [47], and Wolper and Lovinfosse [83], published similar work using structural induction to reason about systems with unboundedly large numbers of identical components. In [47] and [83] examples are given where processes and specifications are modelled in CSP and CCS [57, 58] (a process algebra comparable to CSP). Both require an invariant to be defined which captures the desired behaviour of the system. Proof obligations are then constructed which correspond to the base case and inductive step of an induction. The proof obligations are then discharged using model checkers, and normal inductive reasoning combines the two steps to establish the invariant independently of the system's size.

Structural inductions are all variants on the following basic inductive technique: To show that property P (the network invariant) holds of some network no matter how large the network:

- show firstly that the network of size 1 satisfies property P. This is the base case.

  $network\ size = 1 \Rightarrow P$

- Secondly, show that if P is true of the network with size k, then P is true of the network with size $k+1$. This is the step case.

  $\forall k \bullet P[^{k}/network\ size] \Rightarrow P[^{k+1}/network\ size]$

In [47] the method is demonstrated on an algorithm for maintaining the consistency of multiple copies of a data object in a distributed system. In [83] two examples are given: a buffer composed of identical elementary buffers and a version of the distributed mutual exclusion problem. The second of these is similar to the example in [47], and to the structural induction technique described in Chapter 2 of this thesis.

However, the buffer example of [83] is different. The intention is to establish that any number of elementary (1-place) data-independent buffers can be piped into each other, and the resulting system will still behave like a buffer. Assuming there are no faults with our buffers this is clearly correct! If we were performing a structural induction on the number of elementary buffers which can be piped into each other and still satisfy the property, the invariant should essentially represent the behaviour of a buffer of arbitrary size. This is impossible to represent directly in the context of a finite-state system, a requirement if we were to use model-checkers to verify the proof obligations. So an alternative method is used in [83]:

A structural induction is performed, but not with the invariant being a buffer of unbounded size; instead it is a particular data independent process which, when used in the induction, implies the desired buffering property[4]. This is not proposed as a general technique for establishing a variety of properties of scalable systems, unlike those presented in this thesis, but rather as a method specific to the example

[4]The buffer dealt with in this example is not quite the CSP standard notion of a buffer. For

in question. Also, it should be noted that the use of data independence is entirely different to that found in this thesis.

In [61] Pyssysalo presents an induction theorem for ring protocols. The rings consist of an arbitrary number of nodes who know each others identities. Nodes can use identities as addresses in communications to other nodes. Structural induction, of the type described above, cannot be used since an addition of a node process would require a change in the structure of the nodes already present in the ring. The nodes in a ring of size n would require an extension to their communication events to allow them to communicate with the new member of the ring (node $n+1$). This is solved in [61] by using a Pr/T-net description of a node which abstracts away the need for knowledge of other nodes identities. The abstraction is done in such a way that identities may be used as addresses and passed around the ring, but that the semantics of the system is independent of the data type of node identities (a form of data abstraction). A structural induction is performed on the underlying Petri-net which establishes results for all sizes of ring.

It is this limitation on structural induction that the techniques in this thesis address: the inability to prove properties of scalable systems composed of components identical in every way, except that they have unique identities, and must possess knowledge of each others identities. However, the results obtained in [61] only hold for ring topologies. The techniques presented in this thesis are of a far more general nature, in that they can be applied to a greater variety of network topologies.

Other approaches for proving properties of systems of arbitrary size have included the use of temporal logic [13, 34, 81], the consideration of network [15] and graph grammars [64] and tableau methods for the mu-calculus [9]. In [13] Clarke and Grumberg attempt to address the problem in the context of model-checking; logics are devised and used to reduce systems using bisimulation, with the aim of avoiding the state explosion problem commonly associated with analysing these types of systems. The work of German and Sistla, presented in [34], is closer to that presented

example, by the reasoning in [83] a process like:

$$R = in?x \rightarrow in?y \rightarrow out!x \rightarrow out!y \rightarrow R$$

would be considered a buffer. However, by the standard CSP notion, this is not a buffer because it can refuse to perform an output when it is non-empty. Tom Newcomb (see [80]) has revised the invariant to prove the standard CSP buffer. Essentially you establish that not only are all inputs output, and in the correct order, but also that the buffer can always empty itself.

in this thesis; systems are modelled in CCS, specifications given in temporal logic, and methods described for automatically verifying properties of systems containing an arbitrary number of finite state processes. However, structural induction on the size of networks is not used. In contrast, the method presented by Clarke, Grumberg and Jha, in [15], does use structural induction. But it is the kind of structural induction of Wolper and Lovinfosse [83] which most resembles the work the author presented in [17], and which has been taken as a starting point for the research reported in this thesis. This is a form of structural induction for systems composed of many totally identical processes. Chapter 2 describes this starting point with an example of a structural induction for CSP.

1.5 Thesis outline

The techniques presented in this thesis are a natural progression on structural induction. The example which follows in Chapter 2 is a good illustration of this basic structural induction method for FDR, in particular because it demonstrates the power of structural induction to cover a wide variety of network topologies, as alluded to in [15]. In Chapter 3 the basic method of Data Independent Induction is presented. Included in this chapter are two small examples: a distributed token-passing system incorporating leadership election, and a communications protocol involving the multiplexing of channels. In Chapter 4 and Chapter 5 large case studies are considered. Chapter 4 demonstrates a proof of the Time Triggered Group Membership protocol for totally connected networks of arbitrary size. Chapter 5 presents a proof of a QoS resource reservation protocol for arbitrary broadcast trees. In Chapter 6 the method is extended to more structured network topologies, and is demonstrated for arbitrarily large rings. Chapter 7 discusses how to reason about toroid. Finally the conclusions and discussion on future research directions are discussed in Chapter 8. The appendixes contain a description of the CSP_M operators used throughout this thesis, code for all of the examples, and in Appendix I a comparison of statistics for various refinement checks performed on FDR.

Chapter 2

The Monument

This chapter presents a structural induction technique for CSP. The example used is based on an exercise given in Roscoe's *The Theory and Practice of Concurrency* [66]. Roscoe asks the reader to develop a way of preventing deadlock in a particular road network, shown in Figure 2.1 below. This is solved by putting a constraint on certain segments of the road network. Here we show how to construct an inductive proof which can extend the property of deadlock-freedom to all possible road networks constructed from segments of a particular type.

Preventing Deadlock

Figure 2.1 shows a road layout allowing visitors to drive around a monument. All cars enter the road at point A and pass around the loop until they exit at point B. Two adjacent rocks mean that one stretch of the road is too narrow for cars to pass each other – it is single-lane there and two-lanes elsewhere. Only one car may occupy a segment at a time, and cars may only leave the single lane section in the direction of B once they have passed around the monument. (This means that once you have entered the single-lane segment you must pass through to the other side; you may not turn back on yourself). If enough cars pass into the network that all of the road segments around the monument are full than it is possible for deadlock to occur: if a car passes into the single-lane segment C heading from A towards the monument then it will get stuck, since segment D will already be occupied.

To prevent deadlock in the road system a constraint has to be applied to the network. The constraint is that out-going cars (i.e. rightward in the picture) are given priority over the 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.

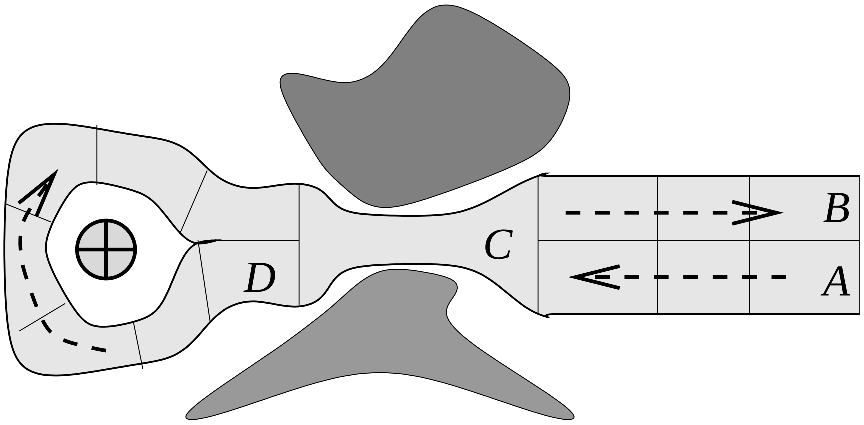


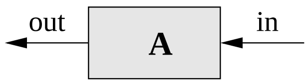
Figure 2.1: *The Monument Road Network*

The Induction

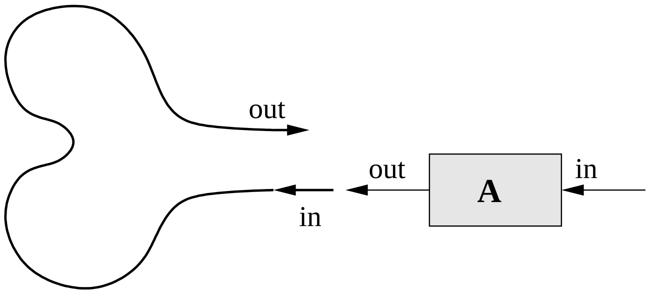
It is possible to model the actual road network seen in Figure 2.1 using CSP, and to use FDR to prove that the constraint described makes the network deadlock-free. This only establishes deadlock-freedom for the particular road network modelled, and in no way implies that any other network topology is also deadlock-free (even if it was only constructed from segments of the type in Figure 2.1). However, using a structural induction technique, such as that described in Section 1.2, it is possible to prove that any road network made up of certain road segments, according to a range of structural rules, is deadlock-free.

The proposition that is to be proven in this example is[1]: *All road networks constructed from road segments of type A, B and CD, according to rules (i) to (v) below, will be deadlock-free.*

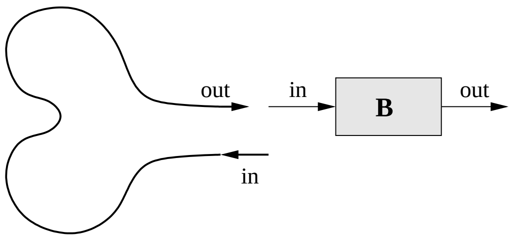
(i) We can have a single road segment of type A. This is the base case.



(ii) The road network can have segments of type A connected to its `in` *channel on A's* `out` *channel.*

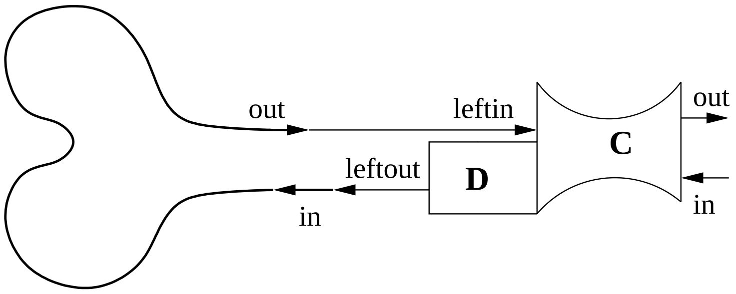


(iii) The road network can have segments of type B connected to its `out` *channel on B's* `in` *channel.*

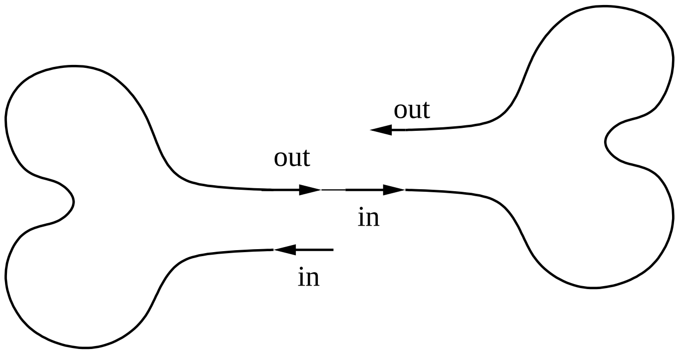


[1]Executable code for this example can be found in Appendix B.

(iv) The road network can have segments of type CD connected to its `in` *and* `out` *channels in the obvious way.*



(v) Two road networks can be joined to each other, where the `out` *channel of one is connected to the* `in` *channel of the other.*



The road segments of type A and B are equivalent; they are distinguished here because of the different roles they play. The corresponding CSP_M for these two processes is:

```
A = in->out->A
```

and

```
B = in->out->B
```

The process CD is the parallel combination of the processes describing segments C and D (over appropriate channels):

```
C = leftin->out->C
    []
    in->leftout->C'

C' = OK->C
     []
     leftin->C''

C'' = out->C'
      []
      OK->out->C

D = in->out->OK->D

CD = C [leftout <-> in, OK <-> OK] (D [[out <- leftout]])
```

This parallel construction of C and D prevents cars from entering the single-lane segment C whilst D is full. This is achieved in CSP_M using the `OK` event; if process `C` ever performs the trace '`in -> leftout`' (representing a vehicle moving into the loop), it may not do so again until it has communicated an `OK` event. Process `C` synchronises with process `D` on the event `OK`, and so `C` has to wait for `D` to perform an `out`. This means that segment C may never input a new vehicle into the loop whilst segment D contains a vehicle. Hence, whenever a car enters the single-lane from the direction of A, segment D is always empty and ready to receive it. This constraint prevents deadlock for the reasons described above.

The property of deadlock-freedom is simply the process which can choose to perform an `in` event or an `out` event non-deterministically :

```
DF = in->DF
     |~|
     out->DF
```

We can prove rules *(i)* to *(v)* by checking the following refinement relations using FDR:

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

It is clear, as described in Section 1.3, that by transitivity and monotonicity of refinement, and by using normal inductive reasoning, that we can combine *(i)* to *(v)* to construct an unbounded number of deadlock-free road networks. For example, a typical system one could create is:

```
((((A [in <-> out] A) [in <-> leftout, out <-> leftin] CD)
           [in <-> out] A) [out <-> in] B)
[in <-> out]
(A [in <-> leftout, out <-> leftin] CD)
```

and the induction has proved that the above system is deadlock-free:

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

Likewise the same claim can be made of Figure 2.2. By using this type of induction we can prove properties of systems which, if we tried to model them explicitly, would be intractable for a model checker. More interesting case studies of this technique can be found in [17, 18, 67].

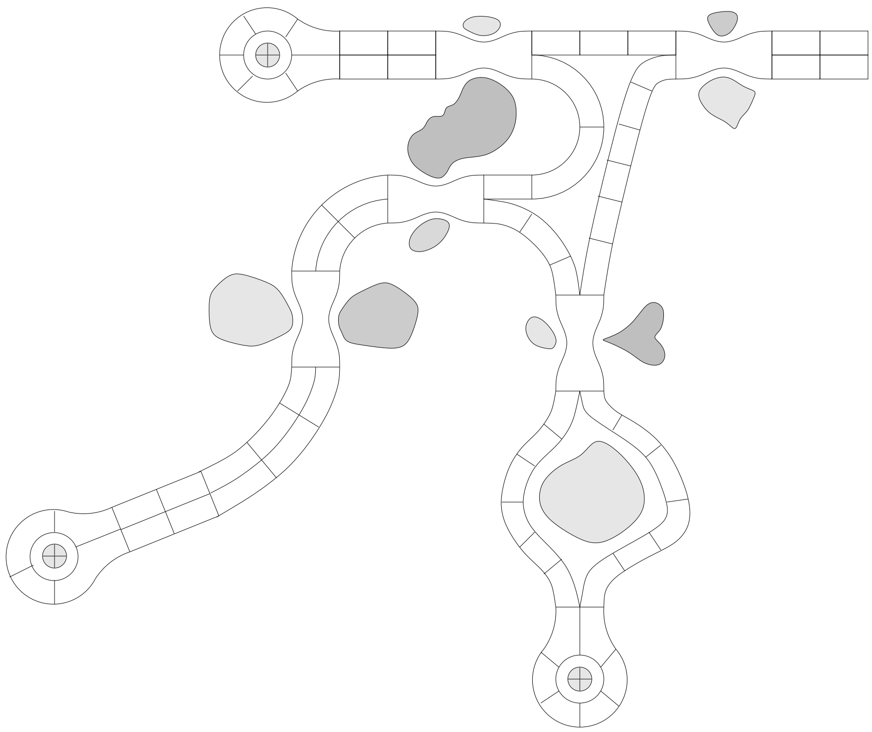
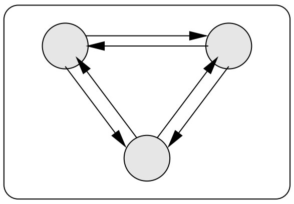


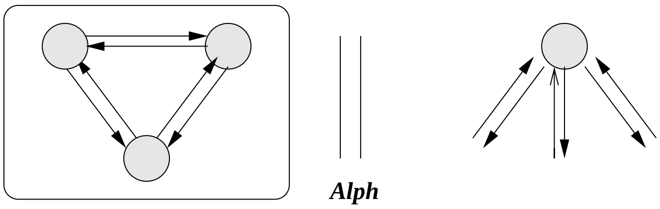
Figure 2.2: *More complicated deadlock-free road network*

Structural induction is a powerful method for proving properties of certain classes of systems, namely, those constructed from a set of fixed, finite types. However, as mentioned in Section 1.4, it is not a method which can be applied directly when components need to know of each others identity, whether for communication purposes or other behaviour decisions.

Imagine we have a network of nodes who communicate directly with each other:



It isn't possible to construct an inductive step which would allow the addition of a new node in such a way that the original network knew of its existence, and could communicate with it. Consider the diagram below:



Even if the new node is given knowledge of all nodes in the current network, and the alphabet *Alph* that it synchronises with that network over contains these events, the nodes in the original network still do not 'know' of its existence. The nodes in the current network will actually have to have their view of the world changed in order for the synchronisation to take place. Induction doesn't provide a means for changing the components, and so cannot provide a means for proving properties of such networks independently of size. This problem is the subject of the research reported in this thesis.

Chapter 3

Data Independent Induction

Our aim is to prove that a network constructed of some collection of replicated components meets some specification, and that it does so independently of the network's size. We want to solve an instance of the PVP where the network is designed to be scalable and the parameter in question represents the number of components present in the network. In particular, we are interested in networks formed of processes which are essentially identical in behaviour, but which possess unique identities and require knowledge of each others' unique identities. An example of such a network would be a distributed database formed from semantically equivalent nodes. The nodes implement a fault tolerance scheme requiring that they keep state reflecting who in the network they believe to keep reliable data. This state has an effect on the nodes' behaviour; in this instance it allows faulty data to be ignored. An intuitive CSP representation of such a network would be:

$$\|x : T \bullet [A(x)]N(x) \tag{3.1}$$

Here we have a collection of node ($N(x)$) processes in parallel ($\|$), synchronising on the common elements in their respective alphabets ($A(x)$). Each process in the network is identical apart from its unique identity x; T is the set of node names[1].

Imagine we want to prove a property *Specification*(T) true of such a network for any

[1] All CSP operators used, and their CSP_M equivalents, are described in Appendix A.

number of nodes. In CSP this would mean proving (3.2) for all sizes of (T):

$$Specification(T) \sqsubseteq \|x : T \bullet [A(x)]N(x) \tag{3.2}$$

A similar network constructed from two sorts of components ($A(x)$ and $B(y)$) could be:

$$(\|x : T \bullet [a(x)]A(x))_{alphA}\|_{alphB}(\|y : T \bullet [b(y)]B(y)), \tag{3.3}$$

where $alphA$ and $alphB$ are the alphabets of the group of $A(x)$ and $B(x)$ processes respectively. As discussed in Chapter 2 above, we can't deal with networks like this using traditional inductive techniques because the individual processes vary with the type T. If we wanted to add another node into the network then we would have to change the state of the nodes already present to enable them to interact with the new node.

We might consider applying data-independence results to such networks. In many cases the component processes in question are data independent of the type of node names. Processes only perform simple operations on the type, such as storing names and comparing one name to another. This is true of many network communication protocols. Let us assume that the nodes in (3.2) are data-independent in the type of T. The process $N(x)$ simply checks to see if messages are for itself, and records the set of nodes it is willing to believe. This is consistent with data-independence. We could then use (1.1), (the threshold calculation formula given by **Theorem 6.2** of [53]), to obtain a threshold on the size of T. If we could show (3.2) holds for every size of T smaller than or equal to the threshold value, this implies (3.2) is true for all possible sizes of T.

But we can't deal with such networks using data-independence alone because indexed parallel composition over data-independent types is prohibited in this theory. Data Independent Induction is a method designed to overcome this limitation and facilitate such proofs through a combination of induction and data-independent reasoning.

3.1 The Basic Method

We want to prove (3.2) for all sizes of T:

$$\mathit{Specification}(T) \sqsubseteq \|x : T \bullet [A(x)]N(x)$$

That is, a network which is built as the indexed parallel composition over a type T of node processes $N(x)$, where x is the identity of the particular node, satisfies a specification *Specification*(T). Both the specification and the nodes treat elements of the type T data independently. We cannot apply data independence to this network construction, but we might find a suitable abstraction for which (3.4) holds:

$$\mathit{Specification}(T) \sqsubseteq \mathit{Abstraction}(T,S) \sqsubseteq \|x : T \bullet [A(x)]N(x) \tag{3.4}$$

The process *Abstraction*(T, S) is data-independent in the type T, and it captures the behaviour of a partially complete network containing the subset S of nodes in T. When $S = T$ then the network is complete, and (3.4) should be true. In order to meet the requirements of data-independence S can be viewed as a predicate. The predicate maps members of the data-type to true if they are in the set S, and false otherwise. By transitivity of refinement (3.4) would imply (3.2). So we can decompose the proof of (3.2) into the verification of the two refinement relations:

$$\mathit{Specification}(T) \quad \sqsubseteq \quad \mathit{Abstraction}(T,S) \tag{3.5}$$

and

$$\mathit{Abstraction}(T,S) \quad \sqsubseteq \quad \|x : T \bullet [A(x)]N(x) \tag{3.6}$$

If an appropriate *Abstraction*(T, S) has been constructed, then (3.5) presents no new theoretical challenges as it already fits within the established theory 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 (3.6), which cannot be addressed within standard data-independence theory, for the same reason as (3.2): it contains a parallel composition, indexed over the data-independent type. We perform an induction on the size of S (from 1 up to $|T|$). As with any induction the main step is the construction of the Inductive Hypothesis ($\mathit{Abstraction}(T, S)$). In our case it is a (usually sequential) process representing partially constructed versions of the network under consideration. It should be data independent in T, should be refined by the partially constructed network it models and, when it represents a complete network, should refine the specification, $\mathit{Specification}(T)$.

The base case of the induction is to show that one node, $N(c)$, (c is an arbitrary member of T and is represented using a constant), refines the partial network containing only that node, $Abstraction(T, \{c\})$:

$$Abstraction(T, \{c\}) \quad \sqsubseteq \quad N(c) \tag{3.7}$$

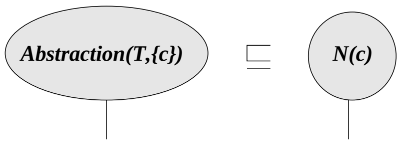


Figure 3.1: *Data independent induction: Base Case*

The inductive step is to show that for any nonempty S, (treated as a predicate), and constant c outside S:

$$Abstraction(T, S') \quad \sqsubseteq \quad Abstraction(T, S)[AA(S)||A(c)]N(c) \tag{3.8}$$

where S' is the union of S and $\{c\}$, $AA(S)$ is the alphabet of $\mathit{Abstraction}(T, S)$, which is the union of $\{A(x)|x \leftarrow S\}$ (the alphabets of each node), and $A(c)$ the alphabet of node $N(c)$. See Figure 3.2.

If we can establish these two things the inductive proof of (3.6) follows easily: if it

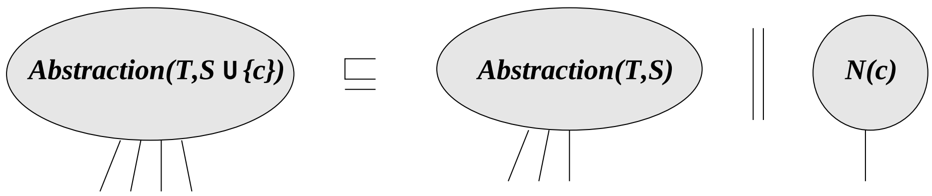


Figure 3.2: *Data independent induction: Step Case*

is true for a network S then

$$\begin{array}{rcll} Abstraction(T,S') & \sqsubseteq & Abstraction(T,S)[AA(S)||A(c)]N(c) & \\ & \sqsubseteq & (\|x : S \bullet [A(x)]N(x))[AA(S)||A(c)]N(c) & (i) \\ & \sqsubseteq & (\|x : S' \bullet [A(x)]N(x)) & (ii) \end{array}$$

The monotonicity of the parallel operator and the inductive hypothesis give (i). The definition of indexed parallel gives (ii).

We want to establish (3.6) for any size of network T. Fortunately, both (3.7) and (3.8) can be proved using data-independence extended by predicate symbols and constants. This extension to data-independence theory allows us to treat S (the subset of the network present in a partially-built network) as a predicate symbol, and the particular node, c, being added onto the network as a constant. Therefore, we can reasonably hope to calculate a threshold on the size of T which will allow us to establish the two steps (3.7) and (3.8) for all values of their parameters. By checking that (3.7) and (3.8) hold for all sizes of T equal to the threshold and smaller, we can establish they hold for all T. We can thus conclude the truth of (3.7) and (3.8) (for arbitrary T, S and c), and that (3.6) holds for all nonempty S and finite T, from a finite collection of checks which can be verified automatically on FDR. 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.

Note that since $Abstraction(T,S)$ has to treat elements of the type T data independently, the connection scheme of the network must also be data-independent of the type. The topology of the network, and all sub-networks, is captured by the the predicate S. The only information this predicate can provide is the identities of nodes in the sub-network; therefore, their individual connection schemes must be identical.

Two possible network topologies are shown in Figure 3.3 below, and another in Figure 3.5:

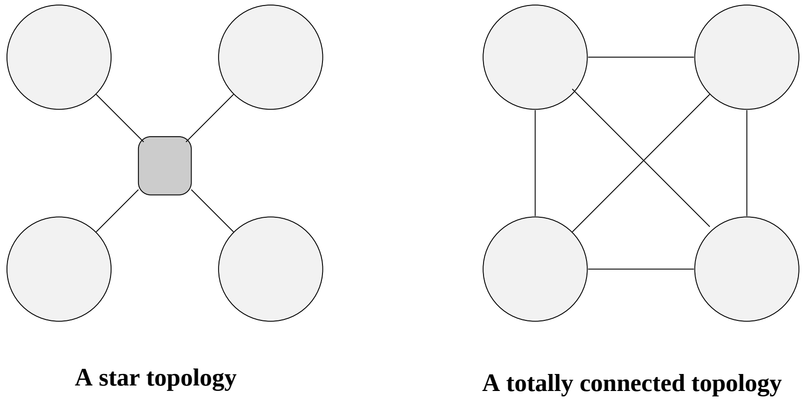


Figure 3.3: *Possible topologies.*

In the star topology shown in Figure 3.3, all the nodes are connected to a central hub, and only that hub. This type of topology can easily be captured using the induction scheme described above. Whenever a node is added on to the network it simply synchronises with the hub. The hub can be represented data-independently using a constant symbol. So the connection scheme is entirely independent in the type of node names.

In a totally connected topology the node being added on to the network doesn't need the knowledge of the network size. It simply needs to talk to everyone there. The set of nodes already in existence in the partial network can be represented using a predicate, making this type of connection scheme data-independent in the type of node names. Of course, these are not the only topologies which can be reasoned about in this manner.

Previously, data-independence theories have been used in establishing results for networks independent of the types of data they store and move around. Data Independent Induction not only addresses a different sort of data type, but it also provides a way of reasoning about networks with indexed parallel compositions over the type. It is appropriate for networks which are designed to be scalable over a type (where each component is data independent of the type), and where an appropriate $Abstraction(T, S)$ process can be constructed. In some cases it may be impossible to find a finite-state $Abstraction(T, S)$. The greatest challenge in building an

Abstraction(T, S) process appears to be keeping it sufficiently small to enable both finite thresholds and automated checking. For example, in some cases nodes might require information to be stored, and this might differ for each node over time. To capture this, different state would have to be stored in *Abstraction*(T, S) for each node, undoubtedly making the *Abstraction*(T, S) process too large to model check.

3.2 Small Examples

3.2.1 Example 1

Consider the following example of a distributed network which combines leadership election and token passing. This example has much in common with one of those given in [15, 83], 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_M description of such a process:

```
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})
```

The channel `leader?x` is used to decide a new leader. The channels `cr_start.x` and `cr_end.x` represents the node `x` beginning and ending the critical section. The channel `pass.x.y` passes the token from node `x` to node `y`. `diff(P,Q)` is the difference of the two sets `P` and `Q`. The parameter `T` is the data-type of node names. Note the function `others(i,T)`, whilst normally not permitted in a data-independent process in the data-type T, is permitted in this case. This is because the result of the function is only used in defining the events being offered as inputs on the `pass` channel. It is not possible to establish knowledge of T which might affect the behaviour of the process. In each case only one member of the type is actually input.

If a process is the *leader* then it behaves as `Leader(T,id)`. The identity of the process is `id`. The leader process is always willing to pass the leadership, `leader?j:T`. If the new leader is itself then it remains in the same state, otherwise it becomes a process which isn't the leader, and doesn't possess the token: `Empty(T,id,j)`.

The leader can always perform critical sections:

(`cr_start.id -> cr_end.id -> Leader(T,id)`),

unless it has passed the token to another process, in which case it must wait for the token to return to itself (`id`):

(`pass?j!id -> Leader(T,id)`).

A process (`id`) which doesn't possess the token, nor is the leader (`L'`), can either receive the token, `pass.L'.id-> Full(T,id,L')` and become full, or it is willing for the leadership to change: `leader?j':T`. If it becomes the new leader (`j'` $\equiv$ `id`) then it behaves as `Leader(T,id)`.

A process which is not the leader but possesses the token can either perform the critical section, or pass the token back to the leader. It cannot enter into a leadership change event since the token is obviously not with the leader.

If S is any sub-set of these nodes, represented using a predicate, and $L0$ is a constant representing the initial leader, then we can build the composition of all the nodes from T as:

$$\begin{array}{rcll} LNet(T,S,L0) & = & \|i : S \bullet [A(T,i)] & (\mathit{if}\, i = L0 \\ & & & \mathit{then} Leader(S,i) \\ & & & \mathit{else} Empty(S,i,L0)) \end{array}$$

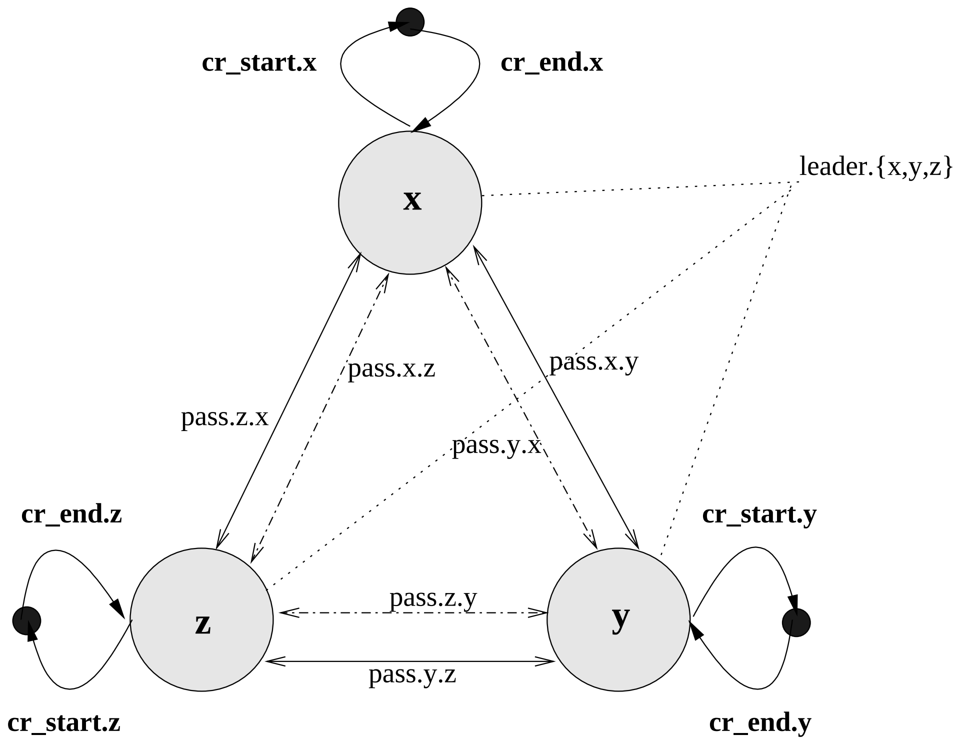


Figure 3.4: *A token passing ring*

$LNet(T, S, L0)$ 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.

Figure 3.4 shows an example network containing three processes. Note that for any number of nodes the network is entirely symmetric, and fully connected, as a consequence of the data-independent design.

The Induction

Ultimately, we want to establish that $LNet(T, S, L0)$ refines some property no matter how large the network is (i.e. for all T). Our first challenge is to find the Inductive Hypothesis, $Abstraction(T, S)$, which captures the behaviour of $LNet(T, S, L0)$, but is data-independent in T. It must be possible to establish this Inductive Hypothesis using data independent induction, as described above.

In order to give the reader some intuition into constructing such processes, the development has been broken down into phases:

Phase 1 First let us consider a well-behaved partial network `Phase1(T,S,L)`. (We are making an assumption that the leadership never passes outside S, and that therefore the leader is well behaved). In the next phase we will model a true partial network, where it is possible that the leadership passes outside of the network. The type of node names is `T`, the set of nodes in the partial network `S`, and the identity of the leader `L`. The processes in the network inevitably agree on who the leader is due to the synchronisation on `leader`.

The possible behaviours for such a network are: the leader performs the critical section, the leadership changes, or the token is passed to another process. If the token is passed to another process in `S` then either that process will perform the critical section, or the token will be passed back to the leader:

```
Phase1(T,S,L) =
        (cr_start.L -> cr_end.L -> Phase1(T,S,L))
        [] (leader?j:S -> Phase1(T,S,j))
        [] (pass!L?j:others(L,T) -> Phase1'(T,S,L,j))

Phase1'(T,S,L,j) =
        (cr_start.j -> cr_end.j -> Phase1'(T,S,L,j))
        [] (pass.j.L -> Phase1(T,S,L))
```

Note that this is exactly the behaviour of the network when it is complete. Since the leader must always be in the complete network, it must behave correctly.

Phase 2 Next we need to incorporate the possible behaviours of the network if the leader is outside `S`. Here 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. Our *Abstraction*(T, S) process requires a constant parameter `L` representing the leader, as well as the set of node names, predicate `T`, and the subset of nodes currently in the network, predicate `S`:

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

The leadership may change, and could pass to a node outside of the partial network `S`. If the leader is in `S` then it may perform the critical section. If the leader is in `S` it may pass the token to any other node (in or outside of `S`). If the leader is outside `S` we can view it passing the token to somebody inside `S`. Once the token has been passed outside to someone other than the leader our process behaves as `Abstraction'(T,S,L,j)`:

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

If the holder of token `j` is in `S` it may enter the critical section, and then behave as `Abstraction''`. Alternatively, the token may be passed back to the leader. If the leader is not in `S` we can't assume anything about it's behaviour; it could behave correctly, or refuse to pass the token again (modelled using `STOP`, or pass multiple tokens out, including when someone else is in the middle of the critical section, this is captured by `Abstraction''`:

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

If the leader passes multiple tokens out we simply don't care what happens next. We model this using the *CHAOS(set)* construct. This is the process that can perform any event in *set*, or can refuse to perform any event. In CSP_M the set `Events` represents all possible events. When we have a complete network it will not be possible for the leader to misbehave by giving out too many tokens, where the `CHAOS(Events)` and `STOP` options will have disappeared. (Wolper and Lovinfosse use an approach similar to this in [83]).

$\texttt{Abstraction}(T, S, L)$ is the inductive hypothesis we have been aiming for. The base case and the step case (corresponding to (3.7) and (3.8)) are:

$$\texttt{Abstraction}(\texttt{T}, \{\texttt{c}\}, \texttt{L0}) \quad \sqsubseteq \quad \texttt{N}(\texttt{T}, \texttt{L0}, \texttt{c}), \tag{3.9}$$

is the base case, and

$$\texttt{Abstraction}(\texttt{T}, \texttt{S}', \texttt{L0}) \quad \sqsubseteq \quad \texttt{Abstraction}(\texttt{T}, \texttt{S}, \texttt{L0})[\texttt{AA}(\texttt{S})||\texttt{A}(\texttt{c})]\texttt{N}(\texttt{T}, \texttt{L0}, \texttt{c}) \tag{3.10}$$

where S' is the union of S and $\{c\}$, the step case.

The result proved by the induction is:

$$\texttt{Abstraction}(\texttt{T}, \texttt{S}, \texttt{L0}) \quad \sqsubseteq \quad ||\texttt{i} : \texttt{S} \bullet [\texttt{A}(\texttt{T}, \texttt{i})]\texttt{N}(\texttt{T}, \texttt{L0}, \texttt{i}) \tag{3.11}$$

When $S = T$ the right-hand-side is the complete network `LNet(T,T,L0)`. As we will shortly see, by analysing (3.9) and (3.10) we can establish finite collections of checks which prove them, and hence (3.11), for all T and S.

Threshold calculations For checks where the processes contain no predicate symbols, the threshold is the greatest number of potentially distinct values of the type that might be present simultaneously during a symbolic comparison between the specification and implementation. As previously described, the formula we use to calculate such a threshold is:

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

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. $L_?^{Impl}$ is the largest number of values of type T that can be input in any *single* visible event of the Implementation and Specification, respectively. $L_\sqcap^{Impl}$ 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.

When there is a single predicate symbol, the threshold is the sum, over all values the predicate might take, of the largest number there might be in a comparison with that value (i.e. B). For example, if the predicate maps members of the type T to the set of Booleans, $\{true, false\}$, then a naive threshold would be $B \times 2$. This is naive because it is possible that certain members of T, represented as constants, may be fixed with respects to 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. The calculations below provide an example of this.

In the base case (3.9) we can apply formula (1.1) directly. This predicts a threshold of **5**.

$$\texttt{Abstraction(T,\{c\},L0)} \;\sqsubseteq\; \texttt{N(T,L0,c)} \qquad (3.9)$$

W^{Spec} is 2, (the `c` and the `L`). W^{Impl} is 2, (the `c` and the `L`). $max(L_?^{Impl}, L_\sqcap^{Impl})$ is 1. This is a naive calculation, since the `L` and `c` are the same in W^{Spec} and W^{Impl}, and hence we can deduct 2 from the threshold, giving a grand total of $B = \mathbf{3}$. We then have to check (3.9) for $T = 1$, $T = 2$ and $T = 3$. Note that when $T = 1$ `c` must be the leader, but when T is larger than 1 `c` may or may not be the leader.

The refinement checks which need to be performed in order to establish the base case for infinitely many inductions are:

```
Abstraction({1..3},{1},1) [FD=  Leader({1..3},1)
Abstraction({1..3},{1},1) [FD=  Empty({1..3},1,2)
Abstraction({1..2},{1},1) [FD=  Leader({1..2},1)
Abstraction({1..2},{1},1) [FD=  Empty({1..2},1,2)
Abstraction({1},{1},1)    [FD=  Leader({1},1)
```

In the step case (3.10) the threshold formula evaluates to $2 + 2 + 2 + 1 = \mathbf{7}$. However, the presence of the predicate symbol increases the second threshold to 13, a maximum of 6 in S and 7 outside S, (as by definition we know that the constant c does not satisfy the predicate).

To refine our calculation consider W^{Spec} and W^{Impl}; the leaders `L` 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` cannot 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 network 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. An example of a largest inductive step check we would need to do is:

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

Further checks (there are 24 including the above) need to be performed for T equal to 6 and smaller, for cases in which there are fewer objects inside S, and to deal with the cases where `L0` is inside S and equal to c, inside S and not equal to c, or neither. A listing of the code for this example, including a base case and step case check, can be found in Appendix F.

3.2.2 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 acknowledgments 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 3.5).

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

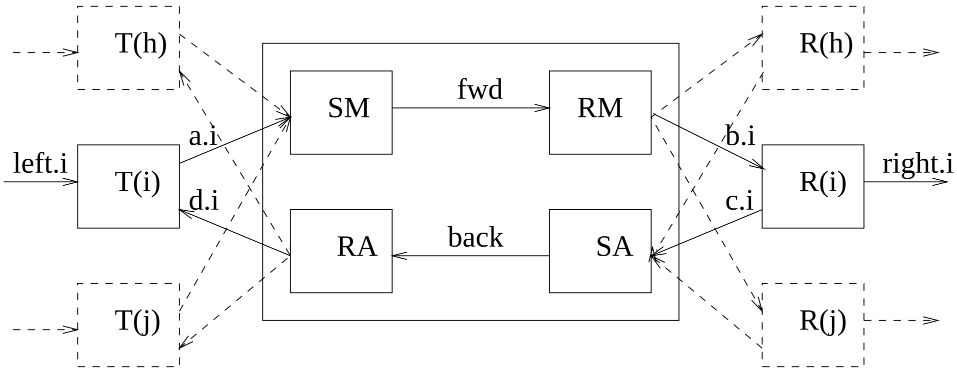


Figure 3.5: *Multiplexing Channels*

our network is equivalent to N independent one-place buffer processes for any N. The correctness of the protocol for any arbitrary number of multiplexed channels is therefore established.

There are actually two data-types in this example that are susceptible to data-independence arguments. The more obvious one is the type of data passed by the channels: the theorems of [54, 66] easily show that it is sufficient to set this type with size 2 to prove that the network behaves properly for any type. That argument is quite separate from the induction. The type which we study is that of `TAGS` (the channel names).

The network 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. The CSP$_M$ 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 network 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 network 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 representation of a partially constructed network is `Abstraction(S)`, where `S` $\subseteq$ `TAGS` and `C0` $\in$ `S`. It 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. Notice that, as in Example 1, this possibility for bad behaviour disappears when $S = T$. The abstraction gives us exactly what we want when the network is complete.

`Abstraction(S)` is built in three parts, one dealing mainly with forward transmission of messages of channels not yet present, one for corresponding acks, and one to record communications for C0, which act like COPY though it can refuse these communications if there are other messages/acks around:

```
SPEC1(S) = let SC = diff(TAGS,S) within
        left.C0?y -> SPEC1(S)
        [] right.C0?y -> SPEC1(S)
        [] a?i:SC?y -> SPEC1'(S,i,y)

SPEC1'(S,i,y) = let SC = diff(TAGS,S) within
        (STOP |~| (left.C0?z -> SPEC1'(S,i,y)
        [] right.C0?z -> SPEC1'(S,i,y)))
        [] b.i.y -> SPEC1(S)
        [] (STOP |~| a?j:SC?z -> SPEC1''(S,i,y,j,z))
```

```
SPEC1''(S,i,y,j,z) =
        (STOP |~| (left.C0?t -> SPEC1''(S,i,y,j,z)
                   [] right.C0?t -> SPEC1''(S,i,y,j,z)))
        [] b.i.y -> SPEC1'(S,j,z)

SPEC2(S) = let SC = diff(TAGS,S) within
        left.C0?y -> SPEC2(S)
        [] right.C0?y -> SPEC2(S)
        [] c?i:SC -> SPEC2'(S,i)

SPEC2'(S,i) = let SC = diff(TAGS,S) within
        (STOP |~| (left.C0?z -> SPEC2'(S,i)
                   [] right.C0?z -> SPEC2'(S,i)))
        [] d.i -> SPEC2(S)
        [] (STOP |~| c?j:SC -> SPEC2''(S,i,j))

SPEC2''(S,i,j) = (STOP |~| (left.C0?t -> SPEC2''(S,i,j)
        [] right.C0?z -> SPEC2''(S,i,j)))
        [] d.i -> SPEC2'(S,j)

SPEC3 = left.C0?x -> right.C0!x -> SPEC3

SPEC(S) = (normal(SPEC1(S)) [|{|left.C0,right.C0|}|] normal(SPEC2(S)))
          [|{|left.C0,right.C0|}|] normal(SPEC3)
```

The function `normal` is a compression performed by FDR. When utilised FDR removes all *tau* transitions from its LTS representation of the process at that point. Such actions represent hidden events or the presence of non-deterministic choices of events. See [30] and [66] for a description of how `normal` works and when to use it.

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[2], refines `SPEC(S)` (the specification of the network of `S` channels multiplexed).

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

[2]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 [66].

```
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}. The set `intevs` is the set of internal events.

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 network 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`).

3.3 Iterative Development

The development of the *Abstraction* process is an iterative process. An initial estimate is made, FDR used to verify a base or step case, and the failure of the refinement guides the adjustments. Often it is easy to overlook the behaviour of nodes outside of the partial network, which disappears when the network is complete. In particular, this behaviour can be 'incorrect' since there is no control over such nodes. It is wise to make sanity checks throughout the development process.

Hint: a useful check would be to ensure that the system is not deadlocked. This would not necessarily show up since STOP refines every process.

The two examples detailed above were constructed as demonstrations of the method, and have previously been published (in part) in [19]. The next two chapters demonstrate the use of Data Independent Induction (here-after referred to as DII) on industrial sized applications.

Chapter 4

The Time-Triggered Protocol

The *Time Triggered Protocol* (TTP) is a reliable and fault tolerant communications architecture for real-time embedded systems. It is designed to support safety and mission critical systems such as those found in fly-by-wire aircraft and drive-by-wire automobiles. Originally developed at the Technical University of Vienna with support from Mercedes-Benz, it is now the basis of a suit of products marketed by TTTech[1]. Current industrial partners include Honeywell, to develop future aircraft electronic systems, and Audi AG for safe auto electronics.

In this case study we consider one member of the TTP protocol stack: a group membership protocol designed to provide fault tolerance by facilitating the diagnosis of classes of communication faults, referred to hereafter as the TTP. In particular, it is intended that the following requirements are satisfied:

1. *Correct Fault Diagnosis* - Non-faulty processors agree on the membership of the group at every step, and the group excludes faulty processors.

[1] See `www.tttech.com`

2. *Self-Diagnosis* - A faulty processor will always eventually self-diagnose itself.

The TTP is a low-cost mechanism since it requires only one bit of information to be tagged onto existing regularly scheduled broadcasts. This makes it particularly suitable for bandwidth-constrained broadcast networks. We have taken the protocol specification from [44]. This paper reports on an attempt to verify the protocols correctness using the Murϕ model checker (and also contains a hand written 'proof' of the same properties). The CSP analysis described in this chapter identified an error in the protocol (in that it didn't meet part of its specification), contradicting the claims made in [44]. The error, a possible fix, and subsequent proof via data independent induction follow.

4.1 The Protocol

The n processor nodes (numbered 0,...,*n-1*) are arranged in a logical ring and attached to a broadcast bus, see Figure 4.1. Execution is synchronous, with a notional *time* variable increased by one at each step; this in turn defines a *slot* in the range 0,...,*n-1* as *time* mod n. Non-faulty processors broadcast whenever it is their slot, and the bus ensures that broadcasts reach all processors instantly.

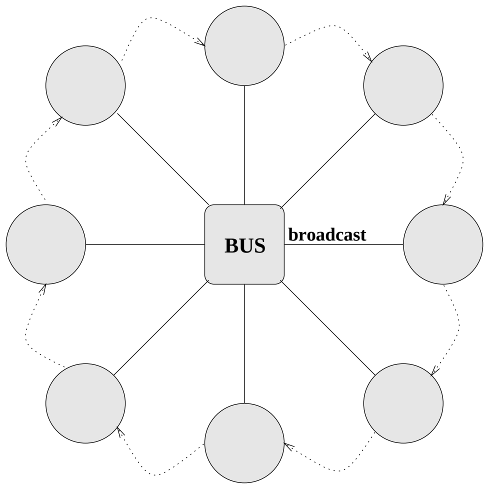


Figure 4.1: *The Time Triggered Protocol Architecture*

The version we are considering is that of [44], which ignores questions such as how the clocks are kept sufficiently synchronous. It is assumed that faults take one of two forms:

Send fault: A processor fails to broadcast. The guilty node is not immediately aware of the failure, however, and continues as though its broadcast has succeeded.
Receive fault: A processor fails to hear a broadcast. It cannot immediately tell when this occurs, since there is no way locally to distinguish this situation from a send fault in the scheduled sender process.

A *send* fault can only occur in a processor when it is the current broadcaster, and a receive fault when it is not. Messages cannot be corrupted, and the bus and channels connecting the bus to the network are assumed to work correctly. All non-receive-faulty processors receive the same message and no processor receives a message from a send-faulty broadcaster. Faults are intermittent: a faulty processor may behave correctly in some slots, and not in others, displaying faults of either sort.

The objective of the protocol 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. In order to do this [44] gives two assumptions which need to be made:

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

However, our analysis revealed that a further assumption is implicitly made in [44]:

- Nodes only become faulty (for the first time) on the slots of previously non-faulty processes.

Each processor has a *local membership set* (**lms**) and an **ack** bit (of type bool). The **lms** contains all the processors it believes to be non-faulty (initially containing all processors), and the **ack** bit indicates whether the processor believes the broadcaster in the previous slot to be faulty. 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.

- If p is the broadcaster b and is in its own **lms**, it broadcasts **ack**(b) and then updates **ack**(b) to *true*. The broadcaster always assumes it has broadcast correctly.
- For each processor p, if the current broadcaster b is not an element of p's **lms**, none of the local variables are changed. Thus, if p receives a message from an unexpected source it will ignore it, and if p has self-diagnosed itself as faulty it no longer updates its local variables since it has removed itself from the network.
- Otherwise, when p is not the broadcaster b, each field of p is updated as follows (notice that **ack**(p) is a local variable of p, and that **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)).

4.2 The CSP model of the TTP

The CSP model of the TTP is constructed from a process representing a protocol node in parallel, (synchronising on all common events), with the processes representing the assumptions under which the protocol is designed to operate. The semantics of CSP, (see Appendix A), dictate that a common event can only take place when all processes synchronising on it are willing to perform the event. Using this concept the assumptions of the protocol are implemented as constraining processes, limiting to the behaviour of the protocol via synchronised parallel composition. It is possible to build networks of `size` protocol nodes by putting `size` node processes in parallel with the protocol assumptions.

At this level of abstraction the notion of time is important because the protocol guarantees aspects of nodes behaviors with respect to discrete points in time. All processors must agree on whose turn it is to communicate: in our CSP model the agreement is abstracted into the processes communicating a special event *tock* at regular intervals (and then incrementing their 'slot' variables by *slot+1 mod size*). This event might correspond to 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 the TTP suite of protocols include mechanisms for clock synchronisation.)

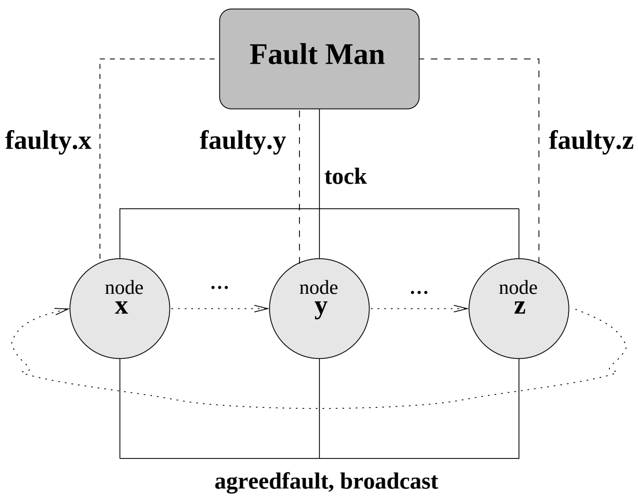


Figure 4.2: *CSP Model of the TTP*

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, a shell which provides us with channels on which we can observe its behaviour, thus allowing us to perform our analysis. What follows is the CSP_M description of a TTP protocol node, `NODE(i)`:

The node process possesses a slot variable which indicates whose turn it is to broadcast in the current time slot. The slot variable for each processor is incremented by `slot+1` *mod size* at each step (point in time). Since faults are intermittent, faulty processors may continue to broadcast, or attempt to. Once a node has become faulty it stays faulty and can no longer broadcast its local membership set details. This is because we are no longer interested in who it thinks is working or not. The protocol is designed to ensure that all NON-faulty processors agree on this fact at every discrete point in time. To make the process tractable using FDR a node process has been broken into two parts: `node(i,bslot,ack,flag)`, which captures the behaviour of a protocol node `i` being its name, `bslot` the expected broadcaster in the current time slot, `ack` it's ack bit and a boolean `flag` indicating if it is a faulty node. The second part of node, `SET(i,lms,fault,sd)`, keeps track of the nodes local membership set, `lms`, and whether the node has self-diagnosed itself as faulty, `sd`:

```
NODE(i) = (((node(i,0,true,false))
                  [[memb <- membership.lms | lms <- memset]])
          [| {| faulty,isin,take,membership |} |]
          SET(i,slot,false,false))\{|isin,take|}
```

The `node(i,bslot,ack,flag)` process controls the broadcasts a node makes. It can check who is in the nodes `lms` by communicating with `SET(i,lms,fault,sd)` on the `isin` channel. It can also remove members from the `lms` using the `take` channel. The `faulty` and `membership` are used to indicate when a node first becomes faulty, and to broadcast a node's `lms` respectively. All nodes synchronise, over the `faulty` channel, with a process which implements the fault rate assumptions (described below). The `membership` event was introduced to check that all non-faulty processors agree on who they believe to be non-faulty at every time slot. This is achieved by synchronising all nodes on a channel where they insist on outputting the value of their `lms`'s. Faulty nodes are always willing to agree on any value. Therefore, if a non-faulty node had an incorrect local membership set it would cause the system to deadlock. FDR provides a built in deadlock-freedom check which was utilised in checking this particular property.

A node process always has to broadcast its lms initially; however, the content of this broadcast is decided by the `SET(...)` process. If it is your own slot, and you are in your own lms then you can broadcast, or you may suffer a send fault and not broadcast:

```
node(i,bslot,ack,flag) = memb -> N(i,bslot,ack,flag)

N(i,bslot,ack,flag) =
(if bslot==i
then isin!i?r -> (if r
                  then
                    (broadcast!i!ack -> tock ->
                     node(i,(bslot+1)%size,true,flag))
                     [] (faulty!i -> tock ->
                     node(i,(bslot+1)%size,true,true))
                    [] flag & (tock ->
                     node(i,(bslot+1)%size,true,flag))
```

Note that nodes only perform the `faulty` event when they first become faulty, thus resulting in a significantly smaller state-space than if the event were performed every time a node exhibited such behaviour. The protocol does not allow for nodes to 'mend' themselves.

If it is your own slot but you are not in your own membership set then you can't broadcast; you must have diagnosed yourself as send faulty already:

```
                  else tock -> node(i,(bslot+1)%size,ack,flag))
```

If it is not your own slot, you may receive a broadcast. If it is from the expected broadcaster then you update your parameters appropriately, otherwise you simply ignore it:

```
else broadcast!bslot?a -> isin!bslot?r ->
        (if r then
```

The following deals with the situation under which you may remove yourself from your own membership set. The communication which informs `SET(...)` that `i` should be removed from the `lms` has to happen before the `tock` event, to ensure that this information is incorporated into the next membership broadcast.

```
            if a and (ack==false)
            then take!i -> tock ->
                node(i,(bslot+1)%size,true,flag)
```

This represents the analysis that the broadcaster `b` is faulty, so `b` is removed from the `lms`:

```
            else if (ack==true) and (a==false)
                 then take!bslot -> tock ->
                   node(i,(bslot+1)%size,false,flag)
```

There are no processors diagnosed as faulty, but the ack bit needs adjusting appropriately:

```
                 else tock -> node(i,(bslot+1)%size,
                   if (a==false) and (ack==true)
                   then false
                   else true,flag)
```

The broadcasting node is not a member of the local membership set, so ignore the broadcast:

```
       else tock -> node(i,(bslot+1)%size,ack,flag))
```

You suffer a receive fault and the expected broadcaster may or may not be send faulty. The `Ignore(S)` function is used to when nodes are receive faulty; it models them either allowing any broadcast to occur, but not acting upon it, or ignore the fact that no broadcast occurred:

```
Ignore(S) = ([]x:S @ x -> Ignore(S)) [] tock -> SKIP
```

Note: the `take` communications are taking place before the `tock` event:

```
[]
not flag & faulty!i -> isin!bslot?r ->
                   ((if r
                     then (if ack==false
                           then take!i -> take!bslot ->
                                        SKIP
                           else take!bslot ->
                                        SKIP)
                     else SKIP);
                   (Ignore({|broadcast|});
                        (if r
                         then node(i,(bslot+1)%size,false,true)
                         else node(i,(bslot+1)%size,ack,true))))
[]
flag &  isin!bslot?r ->  -- receive faulty when already faulty
        ((if r
         then (if ack==false
               then take!i -> take!bslot -> SKIP
               else take!bslot -> SKIP)
         else SKIP);
        (Ignore({|broadcast|});
             (if r
              then node(i,(bslot+1)%size,false,flag)
              else node(i,(bslot+1)%size,ack,flag))))
```

Or, you suffer no receive fault and no broadcast arrives:

```
[]
tock -> isin!bslot?r ->
    (if r
     then if ack==false
          then take!i -> take!bslot ->
                     node(i,(bslot+1)%size,false,flag)
          else take!bslot ->
                     node(i,(bslot+1)%size,false,flag)
    else node(i,(bslot+1)%size,ack,flag)))
```

```
SET(i,lms,fault,sd) =
     isin?x!member(x,lms) -> SET(i,lms,fault,sd)
      [] take?x -> SET(i,diff(lms,{x}),fault,sd)
      [] not fault & membership!lms -> SET(i,lms,fault,sd)
      [] fault & membership?Y -> SET(i,lms,fault,sd)
      [] not fault & faulty!i -> SET(i,lms,true,sd)
      [] ((not member(i,lms)) and not sd) &
  selfdiag!i -> SET(i,lms,fault,true)
```

The model of the TTP, or the `SYSTEM`, is constructed by synchronising (putting in parallel) all of the nodes on the `agreedfault`, `broadcast` and `tock` events. If at any point in time non-faulty nodes disagreed for the first 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. The `agreedfault` event is one introduced for the analysis, as explained above. A listing of this code is also given in Appendix E.1.

The assumptions given above were implemented in our model using two techniques. Firstly, 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 non-deterministically). Secondly, 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 protocol assumptions described on page 40:

```
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. It is easy to deduce which node is suffering which sort of fault from the broadcast slot in which the fault occurs. The protocol does not rely on faulty processor only subsequently committing certain types of faults.

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

4.2.1 Checking the TTP: modeling the specifications

Recall that there were properties we need to check: that the non-faulty nodes diagnose as faulty those nodes that are actually faulty, and that all non-faulty nodes agree on who is faulty at every step, property 1 *Correct Fault Diagnosis*, (though perhaps with some delay in diagnosing newly-faulty ones). And that after *size* slots (steps in time) a faulty processor *must* have self-diagnosed itself, property 2 *Self-Diagnosis*.

The process `RightLMS(badslots)` is a trace specification for part of property 1. It never incorrectly believes a node to be faulty, and after some delay all nodes are diagnosed. The process allows a delay of `count` time slots before insisting on a faulty node being diagnosed, and announced:

```
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)
```

The parameter `badslots` represents the set of nodes who are currently faulty. Initially there would be no faulty processors, so the parameter `badslots` would be equal to the empty set. The process is always willing to communicate the set of nodes it believes to be faulty on the `agreedfault` channel. Any node may exhibit faulty behaviour. If a node, `i`, previously considered not faulty becomes faulty then its identity is added to the set of faulty nodes, `badslots` and the process behaves as `RightLMS'(badslots,i,count)`:

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

By checking that our model refines `RightLMS(badslots)`, we check that the TTP has property 1:

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

`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)` doesn't check that all non-faulty nodes agree on who they believe to be faulty. We can check that this part of property 1 is true simply by checking that the model is deadlock-free. (This is described above). Recall that at the beginning of every slot every non-faulty node has to perform an *agreedfault* event, on which it outputs all nodes that have been removed on this turn from the *lms*.[3] Differing opinions would cause deadlock.

In order to check for self-diagnosis, (property 2), 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 events *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.

The following specification insists that faulty nodes self-diagnose themselves as faulty within one round of broadcasts. Until this has occurred the process is always willing to perform diagnosis of already self-diagnosed faulty nodes, and can refuse all other events.

```
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)
```

[3] This is, of course, not a real communication of the protocol, but 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 results in a smaller state space for FDR to check, resulting in better efficiency

```
     [] 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)))
```

The corresponding refinement check is:

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

4.3 The protocol error and a fix

The CSP analysis of the protocol shows that it does not meet its requirements. *Self-Diagnosis* says that a newly faulty node will remove itself from its local membership set, (and thereby diagnose itself), by *size* number of time slots after the node became faulty. This 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.

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 node 3 broadcasts, it wrongly diagnoses the node 3 as send faulty and removes it from its *lms* (and sets its *ack* bit to false). If node 1 broadcasts next (amongst the nodes not already agreed faulty), it sends its *ack* bit (false) (and sets its *ack* bit to true since nodes always assume that they have successfully broadcast in their own slot). Nodes 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. Node 2 will then broadcast its *ack* bit which will be false. This contradicts what 1 believes. Node 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 and hence 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. Node 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.

In [44] a 'proof' of a stronger property *Theorem 3 (Rapid Self-Diagnosis)* is presented. *Rapid Self-Diagnosis* insists on a faulty node diagnosing itself as faulty when the slots of at most two non-faulty nodes have been passed. This claim, and 'proof', are incorrect for the reason described above. The simple but understandable error is made of forgetting that the faulty process may have mis-diagnosed another correct process and thus ignore what it says. Discussion with one of the authors suggested that either the model checking had not been performed, or that the fault model was so restrictive that the system never exhibited the error. John Rushby has since successfully verified the protocol using the theorem prover PVS, see [73].

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 led to an alternative solution which is presented in [45].

4.4 The Induction

It is an obvious idea to try to use DII over the type of node names to prove that the TTP works. Unfortunately the calculations that nodes perform to decide whose turn it is to broadcast are highly dependent on how many nodes are present. They calculate whose turn it is to broadcast next using:

$$currentslot + 1 \bmod size$$

A node process is not data independent in the necessary type. However, the protocol does not depend crucially for its correctness on the cyclic distribution of slots, but 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.

So we can make nodes data-independent in the type of node names by generalising the protocol in such a way that it does not rely on the round-robin allocation of broadcasting turns. We will simply assume that there is some unspecified mechanism by which the nodes can agree whose go it is.

4.4.1 Making the protocol data independent

The model is modified so that the mechanism for deciding whose turn it is becomes 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 is claimed, informally, is that each new fault is diagnosed promptly provided that no second new fault is introduced during this process. This notion of "promptly" basically means 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 (part of specification property 1 *Correct Faulty Diagnosis*). 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. Note, 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 all correct nodes have diagnosed all incorrect ones and no others, all incorrect ones have self-diagnosed, and 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 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 is 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 commits a receive fault, becoming faulty for the first time, – *either* – the next slot is one of the faulty nodes 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. The behaviour of the system is essentially the same whatever order the nodes become faulty; it is completely symmetric which node goes faulty next. A different ordering of faults would have an effect on the system behaviour, but this effect is not significant to the properties we wish to check. So we can map any ordering of faults onto any other permutation on the ordering. This is not nearly so clear in the cyclic case, since you could imagine that it was significant what the 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 [28, 29, 41] 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. The code for a TTP node re-coded for this optimisation is given in Appendix E.2.

4.4.2 Abstraction

So long as we are able to find an appropriate Abstraction process we are now in a position to construct a proof of the type given in Section 3. Recall that our intention is inductively to prove that a sequential process Abstraction is an abstraction of the network constructed from the parallel composition of node processes. We then intend to establish this for all sizes of network. We need to establish the base and step case, where:

$$\mathtt{Abstraction}(\mathtt{T},\{\mathtt{c}\}) \sqsubseteq \mathtt{N}(\mathtt{T},\mathtt{c}), \tag{4.1}$$

is the base case, and

$$\mathtt{Abstraction}(\mathtt{T},\mathtt{S}') \sqsubseteq \mathtt{Abstraction}(\mathtt{T},\mathtt{S})[\mathtt{AA}(\mathtt{S})||\mathtt{A}(\mathtt{c})]\mathtt{N}(\mathtt{T},\mathtt{L0},\mathtt{c}), \tag{4.2}$$

the step case[4]. T is the datatype of node names, c the name of a particular node, and S and S' the predicates representing the set of nodes present in a partial network.

A naive encoding of `Abstraction` might involve keeping state which represents the set of all faulty nodes. This would pose a problem when we try to establish a threshold on the number of node names we need to consider. The set of faulty nodes would vary in size over time. The equation we use to calculate the threshold contains a component representing the maximal number of unique values of the type needed to be stored at any one time. Clearly this would be unboundedly large in such a case, making it impossible to establish a threshold. Whilst it is easy to construct both a `Abstraction` and a `Node` process which are data independent in the type, it is not trivial to construct them in the manner we require.

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. Our aim will be to run checks in which the new set variables are bounded in size.

The replacement has to occur in both the `Node` and `Abstraction` processes:

- `Node(i,oldbads, newbads, ack)`: node `i` which was initialised with the information that the set of nodes satisfying the predicate `oldbads` 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.

[4] These are copied directly from Section 3

- `SDNode(i)` a simplified representation of a node that has already self-diagnosed.
- `Abstraction(S,oldbads,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`.
- `Abstraction'(S,oldbads,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.

The assumptions, and to some extent what to do about any improper behaviour of "nodes" outside `S`, are dealt with in `Abstraction` 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. (A similar approach was used in the first of the examples in Chapter 3). Code for the Abstraction process can be found in Appendix E.3.

The checks which need to be performed are:

```
assert ABSTRACTION({0},{}) [F= Processor(0,{})
```

This is the base case. And:

```
assert ABSTRACTION({0,1,2},{}) [F= (ABSTRACTION({0,1},{})
                        [| {| agreedfault,broadcast,tock,whosego |} |]
                                         Processor(2,{}))

assert ABSTRACTION({0,1,2,3},{}) [F= (ABSTRACTION({0,1,2},{})
                        [| {| agreedfault,broadcast,tock,whosego |} |]
                                         Processor(3,{}))

assert ABSTRACTION({0,1},{}) [F= (ABSTRACTION({0},{})
                        [| {| agreedfault,broadcast,tock,whosego |} |]
                                         Processor(1,{}))
```

which are the necessary inductive steps. All four refinement relations can be automatically checked using FDR. By data independent reasoning it is the case that

Abstraction is a correct abstraction of the network of nodes, for all sizes of network. Proving properties of true of the network, independent of size, now falls into standard data independence techniques.

A description of the work presented in this chapter was also published as a Technical Report of The University of Oxford [21].

Chapter 5

RSVP: a resource ReSerVation Protocol

In [17, 18] a structural induction technique was used to prove various end-to-end properties of arbitrary binary tree networks which modelled aspects of the resource reservation protocol RSVP[1]. This was a structural induction using CSP and FDR, as described in Chapter 2. Here we will demonstrate how DII can be used to provide proofs of the same property for tree networks with arbitrary degrees of branching.

5.1 The Protocol

The resource ReSerVation Protocol (RSVP)[2] is a protocol designed to support resource reservation for high-bandwidth multicasts over IP networks. This protocol addresses those requirements associated with the latest generation of applications, such as remote video and multimedia conferencing, which are sensitive to the quality of service provided by the network. These applications depend on certain levels of resource (bandwidth, buffer space, etc.) allocation in order to operate acceptably. One strategy for dealing with large demands for bandwidth is to "multicast" from sources to receivers, with transmissions along a number of intermediate links shared

[1] This work on structural induction applied to RSVP develops ideas reported by Reed, Jackson, Deianov and Reed in [67]

[2] A description of RSVP can be found at `http://www.isi.edu/div7/rsvp/.index.html`.

by "downstream" nodes. The RSVP approach is to create and maintain resource reservations along each link of a previously determined multicast route.

Routes consist of multiple sources and receivers connected by arbitrary numbers of intermediate nodes. Messages requesting amounts of bandwidth originate at receivers and are passed upstream towards the source. At any node, if a resource reservation is in place for an amount of bandwidth then it may be shared by all receivers downstream, removing any unnecessary duplication and reducing network traffic. If at any intermediate node a request is rejected a reject message is passed downstream and the request discarded. Otherwise, requests are propagated as far as the closest point along the way to the source where a reservation level greater than or equal to it has been made.

It is this traffic-reducing property which is modelled in [17, 18]. Here the protocol nodes are binary in that they possess two downstream channels and one upstream; in contrast a source process only has a downstream channel (since it is the ultimate destination). The property proven is that the receivers always receive a response to each unique request for bandwidth (or that the interface presented by the network to a receiver at any point has the behaviour of a source process independently of how many intermediate nodes lie along the path between source and receiver). It is established that this correct behaviour is presented on a particular interface no matter what events, if any, are performed elsewhere on the network. This is achieved by lazily abstracting all the network's behaviour which is not on the path between the source and receiver under consideration. So in Figure 5.1 the correct behaviour presented on the downstream branch at A is independent of the behaviour of the rest of the network at the points marked $*$. Dotted boxes containing $*$ represent the lazy abstraction of all events on that channel. In this case there would be two refinement relations corresponding to the inductive step, one for each of the downstream channels of a *Node* where the other has been lazily abstracted away.

Using this type of structural induction it would be possible to prove properties of an arbitrary tree constructed from certain nodes. However, in such a proof the maximum degree of branching will be fixed, and separate inductive cases need to be model-checked for each branching degree. In order to prove properties hold independently of the branching of trees we combine structural and data independent induction.

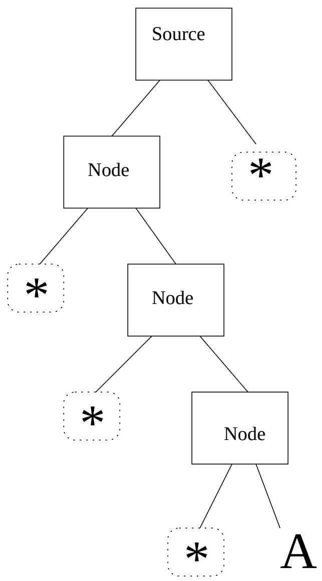


Figure 5.1: *A Binary Tree.*

5.2 Arbitrary Branching Trees

We take as our starting point the protocol model in [18]. Our aim is to show that any arbitrary tree network, where the branching degree varies over the network, will offer correct behaviour, (specified by property P), on any interface to a receiver. In the case of the binary trees we lazily abstracted one of the two downstream channels of any intermediate node in the network. To achieve N-branching node trees we need to establish the structural induction for all-but-one of any number of channels lazily abstracted. We need to establish refinement relations (5.1) and (5.2) below for a *Node* with any N *down* channels, where *AlphDown* is the set of all downstream communication *except* that labeled by a particular constant C (the name of the arbitrary channel we are leaving visible).

$$P \sqsubseteq Source \tag{5.1}$$

$$P \sqsubseteq \mathcal{L}_{Alphdown}(P[down \longleftrightarrow up]Node), \tag{5.2}$$

$\mathcal{L}_A(P)$ represents the lazy abstraction of the events in A from P. Superficially we cannot do this using a finite state model checker like FDR, since it requires an infinite number of checks, whose state spaces grow with N. Careful use of data

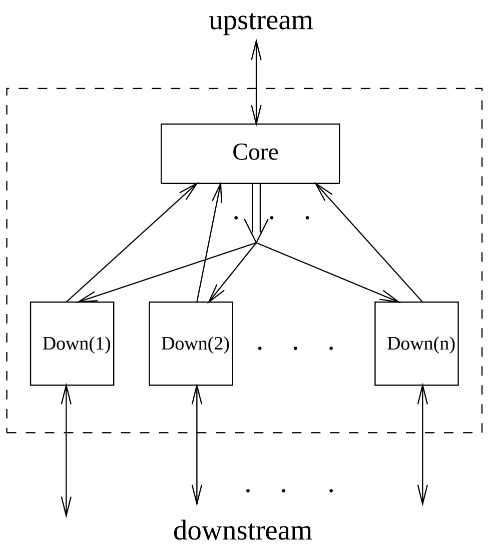


Figure 5.2: *An n-branching node.*

independence, however, enables it.

Let the type T under consideration represent the names of the downstream channels of a node. If P and *Node* can be constructed in such a manner as to be data independent of T, it may be possible to establish a threshold on the size of T for (5.1) and (5.2). FDR can then be used to check the refinement relations corresponding to (5.1) and (5.2), for all sizes smaller than or equal to the threshold. Thus, establishing the structural induction for all possible sizes of T, and so for any arbitrary number of N-branching nodes.

A traffic reducing Node

The *Node* process is constructed from a process representing its core behaviour ($Core$) and the parallel combination of ($Down(x)$) processes, each of which service a downstream channel. Each of the $Down(x)$ processes must be data independent in the type T, (each $Down(x)$ process stores its name as a constant symbol). Figure 5.2 shows a node with n downstream channels. (Note that all events internal to the node, inside the dashed box, are hidden).

Each of the $Down(x)$ processes is responsible for maintaining state on the reservations already in place on the node, and can issue responses to requests as appropriate. If a request is to be propagated upstream then the $Down(x)$ process sends the request to the $Core$, on a local request channel. The $Core$ process then propagates this request upstream.

In order to perform effective merging of requests, and so achieve the desired traffic reduction properties, each of the $Down(x)$ processes needs to know of responses to requests made on the other downstream channels of the node. Figure 5.2 indicates how this is achieved; all replies from upstream are sent to the $Core$ which in turn forwards these over the one reply channel to all of the node's $Down(x)$ processes (this is thus a *broadcast* communication, as indicated by the different way this channel is treated in this and later figures). All of the internal events of a node are then hidden, since no other processes in the network will ever need to synchronise on them.

We are unable to use this construction of a node in (5.2) because it contains a parallel composition indexed over the data independent type (the names of the collection of $Down(x)$ processes). To perform our structural induction we need to use a process in (5.2) which is equivalent to a node with all but one of its downstream channels lazily abstracted, which is not itself constructed of the parallel composition of $Down(x)$ processes.

The lazy abstraction of events on a channel allows any possible set of events to occur, including the empty set. Therefore it would seem intuitive that we could push the lazy abstraction higher up into $Node$, and instead lazily abstract all the events which $Down(x)$ uses to send its requests to $Core$. If we could show that a node with all but one of its downstream channels lazily abstracted is equivalent to one which is not fully constructed, in that not all of the $Down$ processes are present on the local request channels of $Core$, where all possible requests which could have come from those $Down(x)$ processes have been lazily abstracted, then we would have a data independent representation of a N-branching node which could be used as $Node$ in (5.2).

In order to make the node replacement in our structural induction we need to show that the nodes (a) and (b) in Figure 5.3 are equivalent. We can't use data independence directly for the reasons discussed above. However, we can show this using a combination of induction and data independence:

Lemma 1 *Let the node (a) in Figure 5.3 be $NODEa$, and node (b) be $NODEb$, then $NODEa \equiv NODEb$.*

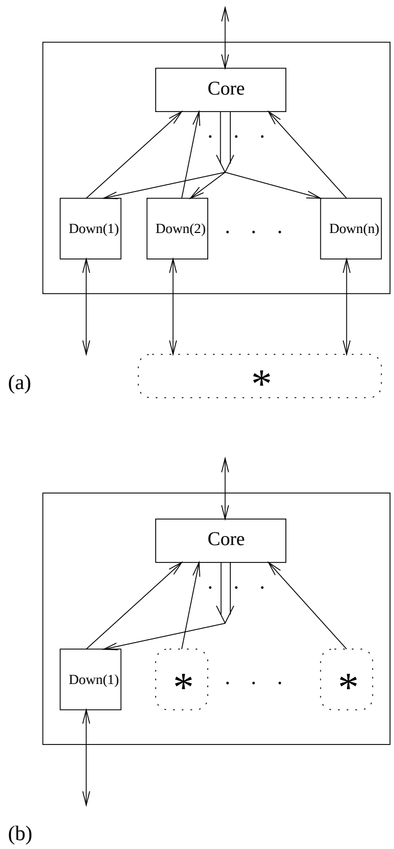


Figure 5.3: *(a) A node with n downstream channels, (b) A node with n local request channels*

All but one of the downstream channels in (a) has been lazily abstracted. In (b) all but one of the local request channels for *Core* have been lazily abstracted away, the remaining local request having a *Down* process attached to it. In both (a) and (b) the channel which *Core* uses to propagate replies to the *Down* processes is represented by a large single arrow. We prove the two nodes are equivalent using data independent induction. We will use the induction to show that if we have a *Core* process with a set of n $Down(x)$ processes on a proper subset of its downstream request channels, then the process that results from abstracting the downstream events of this collection of $Down(x)$ processes is the same as results from lazily abstracting the same set of channels from $Core$ directly, (i.e. without the $Down$ process). The result shown in Figure 5.3 is then implied by the case where there are $Down$ processes on all but 1 of the local downstream request channels.

The base case ($n = 0$) of this induction is trivial as the processes which the induction claims are equal are syntactically the same. The step case is proved by showing that a single $Down$ process, whose external interface is abstracted, can be removed and replaced by the abstraction of its request channel to $Core$. See Figure 5.4.

We can replace the lazy abstraction of a downstream channel from $Down$ by the removal of $Down$ and the lazy abstraction of all events on its local request output channel to $Core$. Lazy abstraction of channels is denoted by dotted curved cornered rectangles. In (b) ABS represents a set of request channels already abstracted:

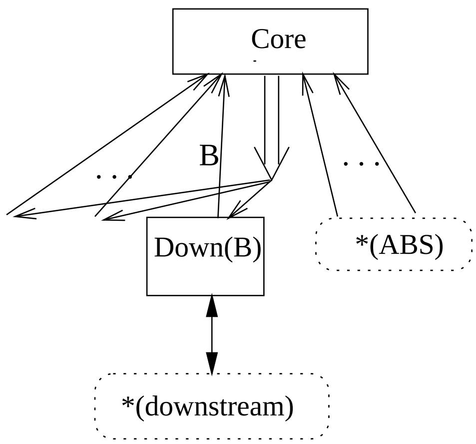
In order to prove equivalence of two processes in CSP it is sufficient to show that each refines the other in the *failures/divergences* model. We use data independence theorems to generate a threshold, this time on the size of T, where ABS is treated as a predicate mapping each member to true if it is in the set. If we can then show that the following refinements hold for all sizes of T, for all possible configurations of ABS, equal to the threshold and below it, then we have shown the two to be equivalent for all T:

$$HI(T, ABS, B) \sqsubseteq_{FD} LOW(T, ABS, B) \tag{5.3}$$

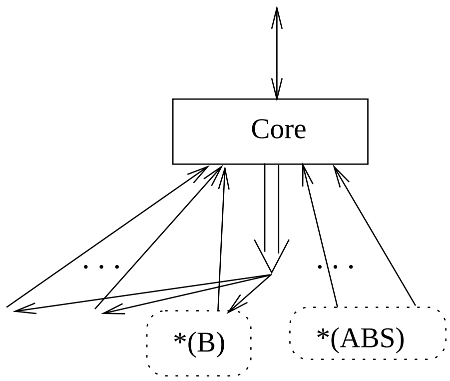
$$LOW(T, ABS, B) \sqsubseteq_{FD} HI(T, ABS, B) \tag{5.4}$$

As with all the examples in the paper, the threshold is small and is 3: the constant B outside ABS, one member of T inside ABS and one outside. This actually only leaves two checks for each of (5.3) and (5.4) (which handle the cases of ABS being empty and of size one). Establishing (5.3) and (5.4) easily justifies the induction, so proving Lemma 1, which in turn allows us to use the $NODEb$ in (5.2).

Thanks to the replacement of $NODEa$ by $NODEb$, (5.1) and (5.2) now become properties which can be proved by data independent reasoning (again with a small threshold). Thus a few refinement checks combined with our earlier results show that any branching width of node can validly be used in our RSVP structural induction.



(a) LOW(ABS,B)



(b) HI(ABS,B)

Figure 5.4: *(a) LOW(ABS,B), (b) HI(ABS,B)*

Hence, a much wider range of network topologies have been proven to have the desired property.

5.3 Example Code

This script is a modification of that developed for [17].

Nodes are now data independent in the type which represents the identities of the downstream channels `DWNS`. A node process is constructed of a local resource process and the appropriate down processes, (all of which are data independent in the type `(DWNS)`). The code corresponding to the local resource for a node:

```
S(T,q) =
let S'(T,q,{}) = localreq ?x:T ! request ? v -> S'(T,q,{v})
    S'(T,q,seen) = (localreq ?x:T ! request ? vv ->
S'(T,q,union(seen,{vv}))
   []
   (|~| v: seen @
                        if v <= q
then upstream . request . v ->
                                 S'(T,q,seen)
else localrep . reply ! reject . v ->
S'(T,q,diff(seen,{v})))
                   []
                   (upstream . reply ? a ? r ->
                      localrep . reply ! a ! r ->
                         S'(T,q,diff(seen,{r}))))

within S'(T,q,{})

LOCALRESOURCE(DS) = |~| q:RESOURCE @ S(DS,q)
```

`RESOURCE` represents the amount of bandwidth the node has available locally.

Model of a downstream channel:

```
Down(id,seen,req,maxacc,minrej) =
  downstream.request?v ->
(if (not(member(v,seen)) and not(member(v,req)))
then Down(id,union({v},seen),req,maxacc,minrej)
else Down(id,seen,req,maxacc,minrej))
  []
  card(seen)>0 &
    (|~| x:seen @ (if x<=maxacc
  then downstream.reply.accept!x ->
    Down(id,diff(seen,{x}),req,maxacc,minrej)
  else (if x>=minrej
then downstream.reply.reject!x ->
     Down(id,diff(seen,{x}),req,maxacc,minrej)
```

```
else localreq.id.request!x ->
     Down(id,diff(seen,{x}),union({x},req),
                                     maxacc,minrej))))
[]
card(req)>0 &
(|~| x:req @ (x<=maxacc) & downstream.reply.accept!x ->
       Down(id,seen,diff(req,{x}),maxacc,minrej)
     []
       (x>=minrej) & downstream.reply.reject!x ->
         Down(id,seen,diff(req,{x}),maxacc,minrej))
[]
localrep.reply?rr?vv ->
(if (rr==accept)
then Down(id,seen,req,max(vv,maxacc),minrej)
else Down(id,seen,req,maxacc,min(vv,minrej)))

DOWN(x) = Down(x,{},{},0,4)
```

A `LOWNODE(S,ABS,ID)` using interface parallel is given below. Note that `S` is the set of all downstream connections from the localsource. `ABS` is the set of all downstreams already abstracted. `ID` is the channel we are concentrating on.

```
LOWNODE =
(LAbs((LAbs(LOCALRESOURCE({A,B}))({|localreq.A|}))
     [| {| localreq.B, localrep |} |]
      DOWN(B))({|downstream|}))
```

The `HIGHNODE` abstracts away the `DOWN(B)` channel higher up:

```
HIGHNODE =
(LAbs(LAbs(LOCALRESOURCE({A,B}))({|localreq.A|}))({|localreq.B|}))
```

The function `LAbs(P)(X)` lazily abstracts the set of events `X` from the process `P`:

```
LAbs(P)(X) = (P [|X|] normal(CHAOS(X)))\X
```

To establish *Lemma 1* we need to check that the two node constructions refine each other in the failures divergences model of CSP:

```
assert LOWNODE [FD= HIGHNODE

assert HIGHNODE [FD= LOWNODE
```

Once we have proven that we may utilise a `HIGHNODE`, we can perform a structural induction using a `HIGHNODE` with a downstream channel on `A`:

```
source(q) =
let   source'(q,{}) = downstream . request ? v -> source'(q,{v})
      source'(q,seen) =
        (downstream . request ? vv ->
                          source'(q,union(seen,{vv}))
        []
        (|~| v: seen @ if v <= q
       then downstream . reply ! accept . v ->
                                 source'(q,diff(seen,{v}))
       else downstream . reply ! reject . v ->
source'(q,diff(seen,{v}))))

within source'(q,{})

SPEC = sbisim(diamond(|~| q:RESOURCE @ normal(source(q))))

NODE = ((LAbs(LOCALRESOURCE({A,B}))({|localreq.A|}))
     [| {| localreq.B, localrep |} |]
      DOWN(B)) \ {| localreq, localrep |}
```

The base case is:

```
assert SPEC [FD= SPEC
```

The step case is:

```
assert SPEC [F= SPEC [downstream <-> upstream] NODE
```

So we have achieved what we set out to do: prove the traffic reducing property of RSVP for any arbitrary tree network. This example is unique since it combines structural induction and data independent induction. This case study was also published in the proceedings of *The International Conference on Parallel and Distributed Processing Techniques and Applications '99* [20].

Chapter 6

Arbitrary Ring Topology

All the examples of DII presented so far follow the principle that, with the exception of the indexed parallel composition that we induct through, the entire system is data independent. If we stick to this restriction then the range of network topologies that can arise is extremely limited, for the simple reason that the selection of which of the indexed class any process talks to is itself then data independent.

Assume for simplicity that the network has the form `|| x:T @ [A(x)] P(x)` (i.e. one node for each member of T, and no others). Each node `P(x)` has to treat any two others `P(y)` and `P(z)` exactly alike: `y` and `z` must not be differentiated by equality with any constant symbol or `x`, or by any predicate symbol. Thus, if it can talk to one of them it can talk to both. The only networks that can be created with no constant symbols or predicates are the fully connected one or the totally disconnected one. With a single constant we can have topologies in which a single node is special, and with a predicate symbol we can have networks in which the nodes are divided up into several distinguished classes.

It is not possible to construct common network structures like chains, rings and grids, since in these each node `P(x)` is connected to a few others whose identity depends functionally upon `x`. For example, in a ring, `P(x)` is connected to `P(pred(x))` and `P(succ(x))`, respectively its predecessor and successor. The problem is that operations over T like `pred(x)` and `succ(x)` are illegal in data independent programs. Fortunately, just as the technique of DII was evolved to get round the restriction that we were not allowed indexed parallel compositions over T, we can adapt it so as to avoid this problem some of the time.

6.1 The Basic Method

Consider the example of a ring shown in Figure 6.1. In order to build a ring of nodes, each connected only to its neighbours, we can first build a simple chain and then connect the last node to the first. Each of the nodes in the ring is essentially the same, in that it can communicate with its neighbours, but differs in that the set of behaviours for each node is different since they each have different sets of neighbours. (If nodes did not know each others' names we could build a ring easily using structural induction, but as nodes do need to know each others names, we build a ring using Data independent induction.)

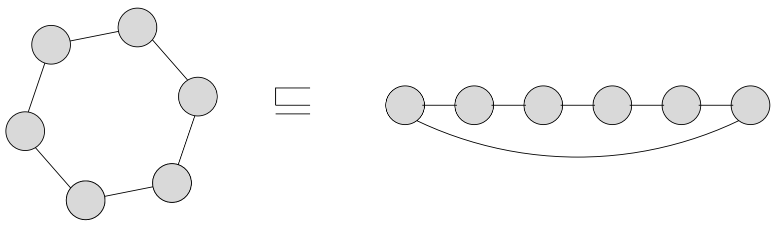


Figure 6.1: *The construction of a ring topology is a chain with its last member connected to its first member in the obvious way.*

The Abstraction process is constructed which represents a chain of nodes, either the partially constructed system or the ring. As before, we use induction to show that the Abstraction process is indeed equivalent to the intuitive parallel composition of nodes, (in the appropriate manner). Data independence reasoning then allows us to make the claim that this equivalence is true no matter how many nodes are in the chain. However, we require more inductive steps than in previous examples since our topology is more complex. A *Node* process will take the form of:

$$\texttt{Node(id,left,right)} \tag{6.1}$$

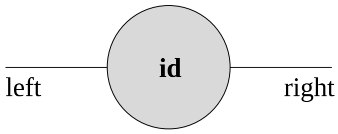


Figure 6.2: *A chain node.*

The `id` of a node is its unique identity, the `left` and `right` parameters the channel names on which it is willing to communicate, see Figure 6.2.

The *Abstraction* process has five parameters:

$$\texttt{Abstraction(S,f,l,bf,al)}. \tag{6.2}$$

The `S` parameter of `Abstraction` is the predicate on the datatype T which represents the nodes in the partial chain. In order to perform the induction we typically need to differentiate up to four different nodes. These nodes are represented as constants in the datatype `T`. They are: the first (`f`) node in the partially constructed chain `S`, the last node (`l`), the node outside of `S` which is before the first (`bf`), and the node outside of `S` which will be connected after the last (`al`). When $\texttt{S} = T$ then the system is complete and the ring has been formed. If the ring is of less than 4 then special cases of all the relevant steps are formed: these cases will be described later.

The base case of the induction is simple: a one node network.

$$\texttt{Abstraction(\{f\},f,f,bf,al)} \;\sqsubseteq\; \texttt{Node(f,bf,al)} \qquad (i)$$

Since there is only one node in `S`, the constants representing the first and last nodes

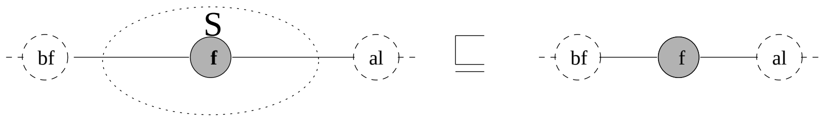


Figure 6.3: *The base case (i).*

in this partially constructed chain must be identical (they are both labeled as `f` in (i)). Figure 6.3 is a graphical representation of (i). Nodes are solid circles, and those inside a dotted ellipse satisfy the predicate with which it is labeled. Dashed lines and circles are not in existence but are known about; this is required since we are considering partially built chains.

To build a two node chain from a one node chain a special case of the inductive step is required. As in (*i*), this is because we have to assume that certain nodes are identical, namely that the first and last nodes in `S` are identical:

```
Abstraction(S',f,al,bf,aal)  ⊑  Abstraction(S,f,f,bf,al)
                                [AA(S)||A(al)]
                                Node(al,f,aal)                (ii)
```

where `S'`=`S`∪{`al`}. See Figure 6.4 for a graphical representation.

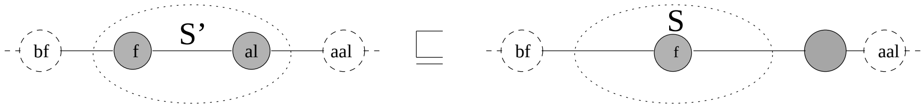


Figure 6.4: *Moving from 1 to 2 (ii).*

To create chains of nodes longer than two we need to assume that `f` and `l` in `S` are different. Three different cases are required: one representing the inductive step and two which represent the adding on of the second to last and last nodes in the chain. The inductive step is similar to the special case in Figure 6.4 except that `l` and `r` in `Abstraction` are different in both cases:

```
Abstraction(S',f,al,bf,aal)  ⊑  Abstraction(S,f,l,bf,al)
                                [AA(S)||A(al)]
                                Node(al,l,aal)                (iii)
```

Figure 6.5 below is a graphical representation:

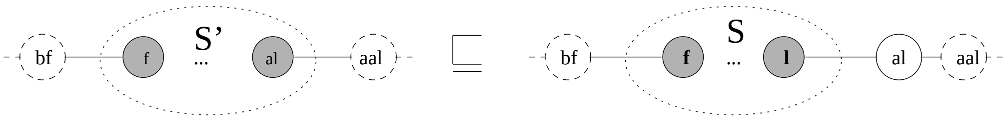


Figure 6.5: *The general step case (iii).*

The step which establishes the addition of the second to last node assumes that `bf` and `aal` in (iii) are the same:

```
Abstraction(S',f,al,bf,bf)  ⊑  Abstraction(S,f,l,bf,al)
                               [AA(S)||A(al)]
                               Node(al,l,bf)                    (iv)
```

The final step creates the ring. This is achieved by a slightly different parallel construction to those above, in that the renamings in the final nodes `al` alphabet should create a synchronization on its right channel with the left channel of the first node `f`, and this composition shall refine the desired data-independent specification:

```
Specification  ⊑  Abstraction(S,f,l,al,f)
                  [AA(S)||AAA(al)]
                  Node(al,l,f)                                  (v)
```

Note that in this case the `bf` node and the `al` node are identical, implying that `aal = f`. See Figure 6.7:

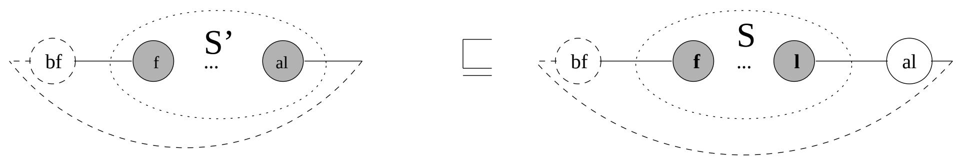


Figure 6.6: *The addition of the second to last node (iv).*

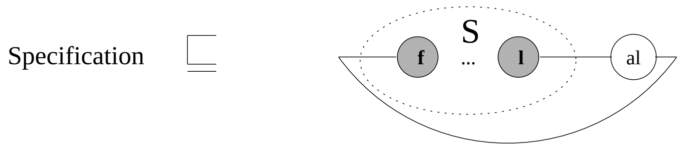


Figure 6.7: *The completion of the ring (v).*

Note that to construct ring networks of four nodes or less each of the refinement checks might contain more constants which are identical to each other. For example, in the trivial case of a ring containing only one node, all of the constants, `f`, `l`, `bf`, and `al`, in `Abstraction` would be identical since we know that the network is complete and not partial, i.e. the cardinality of T is 1. We would require no further checks than this special instance of the base case. However, a ring of size four or less is certainly within the realms of standard model checking. In such cases it would be sensible to model the network directly using indexed parallel composition over the type of node names (data-independence is not required, nor is induction). The effort required to construct a *Abstraction* would not be justified.

If an appropriate data-independent `Abstraction` process can be found then the limitation on the types of network which can be addressed using this technique is that the inductive steps are of a size which can be model checked using FDR, and that the order structure of nodes can be captured using constants and/or uninterpreted predicates. The more constants required to capture a topology, the greater the size of the state space that FDR has to explore. The use of more predicates would increase the size of the threshold on the type T (at worse doubling it for every extra predicate), again increasing the state space dramatically. The sorts of properties one can prove of the ring are limited by the need to be finite space, and also must be data-independent of the type of node names. This would mean they cannot be aware of the order of processes within the ring, since that might provide a means for determining the size of the type.

However, finding a `Abstraction` can be at the very least tricky. It is possible that different `Abstraction` processes would be used to prove different properties of the same system. Each one would be an Abstraction of the system which is strong enough to imply the property, but which abstracts away other behaviour in order to make the process data independent in the type concerned, and as small as possible.

6.2 Example

Consider a simple ring network of *size* nodes, in which each node has the right to broadcast one message at a time to all the other nodes. Messages are passed in one direction round the ring, on the `c` channel, and each one that arrives is output locally and passed on unless it was originated here, in which case it is removed. A node can only accept a new message to broadcast when it has no current expected broadcast. `N(id)` is a typical node:

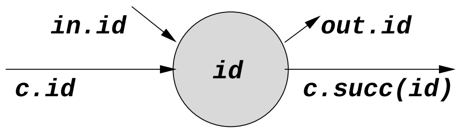


Figure 6.8: *A node.*

The channels `in` and `out` are used for communicating outside of the ring. The node inputs from the network on channel `c.id` where `id` is the nodes name. Nodes can output onto the ring on channel `c.succ(id)`, where `succ(id)` is a node's neighbour on its right:

```
succ(y) = (y+1)%size
```

```
N(x) = in.x?m -> c.succ(x).x!m -> E(x)
       [] c.x?y?m ->
           if x==y
           then N(x)
           else out.x!y.m ->
                   c.succ(x)!y.m -> N(x)

E(x) = c.x?y?m ->
           if x==y
           then N(x)
           else out.x!y.m ->
                   c.succ(x)!y.m -> E(x)
```

If these node processes are put together in parallel they can deadlock: if all users simultaneously insert a message no further progress can be made. Deadlock can be avoided by extra buffering. One way to do this, which would hardly change the following analysis at all, would be to put two processes at each node `N(x)`, in which the `c.succ(x)` has been replaced by `d.x`, in parallel with a one-place buffer from `d.x` to `c.succ(x)`.

In either case we would evidently expect the broadcasts sent by each process to arrive in turn at each individual destination. A complete description of the ring's (and partial rings') behaviour would not be something we can verify by our method because the specification would not be data independent: the broadcasts emerge from the nodes in ring order, and that is not something we can capture.

However the behaviour that links a pair of nodes is accessible. (Of course, having proved the property for an arbitrary pair of nodes, we will really have proved it for all such pairs.) Suppose `p` and `q` are two nodes with `p` $\neq$ `q`. Since clearly we can build a ring of identical processes starting from any point, we can make life a bit easier for ourselves by starting it at `p`. The *Abstraction* process we define comes in two parts: one that describes behaviour relevant to communications between `p` and `q`, and one that deals with all the rest.

The first part has alphabet consisting of those events directly relevant to communications from `p` to `q`, namely `{|in.p,out.q.p|}` together with the related ring communications: for the segment of the ring whose first element is `p` and whose last is `pred`(`al`), these are `{|c.p.p,c.al.p|}`. If the union of these two sets is `ASS`, the first part of the *Abstraction* process is defined as `SS(S,p,q,al)`. In the following definitions, the three uses of `CHAOS(ASS)` all deal with cases that can arise in partially completed rings, but disappear once the whole system is put together.

```
SS(S,p,q,al) = c.p.p?m -> CHAOS(ASS)
               [] in.p?z ->
                   if S(q)
                   then SS'(S,p,q,al,z)
                   else SS''(S,p,q,al,z)

SS'(S,p,q,al,y) = c.p.p?z -> CHAOS(ASS)
                  [] out.q.p.y ->
                         SS''(S,p,q,al,y)

SS''(S,p,q,al,y) = c.p.p?z -> CHAOS(ASS)
                   [] c.al.p.y ->
                          SS'''(S,p,q,al)

SS'''(S,p,q,al) = c.p.p?z -> SS(S,p,q,al)
```

The other part of *Abstraction* is just `CHAOS(A)`, where A is the complete alphabet of the partial ring. The two are put in parallel, synchronising on `ASS`. The corresponding specification for the completed network is just the process

```
Spec = in?p?z -> out.q!p.z -> Spec
```

having alphabet `{| in.p,out.q.p|}`, in similar parallel composition with `CHAOS({| in, out |})`. The scheme of induction we set out above, hiding the channels `c.x` as they become internalised within S, then goes through precisely. There are in fact two data independent types here: the important one is the one of node names, but the type of message contents is also one and falls easily within the scope of the results which allow a threshold of 2. We can then use normal data independent reasoning to prove properties of Abstraction which hold independently of T, and so independently of the size of ring. A full listing of this script, including thresholds and all of the refinement checks, can be found in Appendix H. The results described in this chapter have been published (in part) in the proceedings of *The International Conference on Parallel and Distributed Processing Techniques* 2000 [22].

Chapter 7

A Toroidal Topology

7.1 Technique 1

Grid structures are used widely in networks. A torus is a special type of grid where opposite ends are connected, see Figure 7.1. Parallel computers often have mesh or torus interconnection networks for efficiency, examples include the MPP from Goodyear Aerospace [6] and the J-Machine from MIT [60]. The performance of a parallel computer is largely dependent on the performance of its communication network, making the development of efficient routing algorithms critical, see [33] for an example of such an algorithm. It might seem intuitive that we should be able to use some kind of data independence reasoning or induction to prove properties of toroid, but it is not necessarily straightforward. Let us imagine that we want to be able to send messages from one point on a torus to another. This would require the nodes which form the torus to both have unique identities, and to have knowledge of other nodes' identities. Each of these nodes would also have to know who its neighbours were, as in the ring topology above. But in the torus nodes have four neighbours, not two. As with all the examples considered in this thesis, we cannot use data independence or structural induction directly, but we can use data independent induction.

There are multiple ways of building a torus using DII; here we will start with the top most row and then work along each subsequent row below it, from left to right. A torus can be decomposed into a series of rings, see Figure 7.3, connected in the

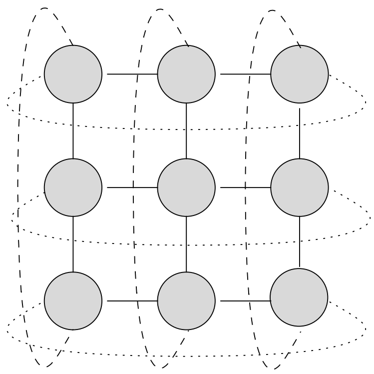


Figure 7.1: *A small torus.*

obvious way (along either the X or Y axis). Each row of the torus can be constructed using the data independent induction scheme for rings, described in Chapter 6 above.

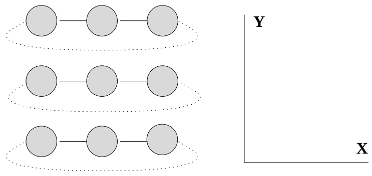


Figure 7.2: *A torus can be constructed from many rings connected together.*

The *Abstraction* process will need four constants identifying key nodes in the ring/row under construction. However, in order to connect rows together, predicates will be needed which not only identify the nodes present in the network as a whole, S, but also which nodes are in the top row, TP, or the bottom row, BT, of the torus. So the *Abstraction* for a torus should look like:

$$Abstraction(S, fc, bfc, lc, alc, TP, BT) \tag{7.1}$$

The constants fc, bfc, lc and alc represent the first node, the node before the first, the last node and the node after the last in the current row. These constants are required to construct the current row/ring, see Chapter 6 and Figure 7.3 below.

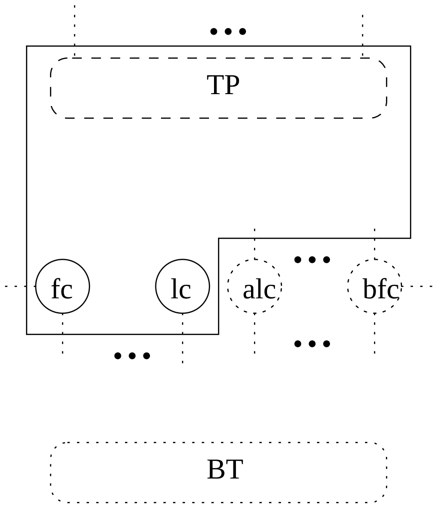


Figure 7.3: *The abstraction process representing a partial torus.*

The first row can be achieved using exactly the same method as in Chapter 6. The base case is a one node network:

$$Abstraction(S, fc, bfc, lc, alc, TP, BT) \sqsubseteq Node(id, u, d, l, r) \tag{7.2}$$

where the node's parameters represent its name, and the names of the nodes neighbours: above it, below it, to its left and right respectively. The step case establishes that adding a node onto a Abstraction process is as good as a Abstraction process containing that node:

$$\begin{aligned} Abstraction(S', fc, bfc, lc, aalc, TP, BT) \;\sqsubseteq\; & Abstraction(S, fc, bfc, lc, alc, TP, BT) \\ & [|\{|l|\}|] Node(id, u, d, l, r) \end{aligned}$$

Notice that the node is only synchronising with the network on its `l` channel; this is because we are constructing the top row (there are no nodes in other rows for it to synchronise with on other channels). Notice also that *TP* must equal *S* and *S'* in both of the *Abstraction* processes, and none of the nodes satisfy *BT* (otherwise we would only be constructing a ring as before)[1].

[1] As when constructing a ring, there are many more equalities between parameters when constructing rows of size four or less. See Chapter 6 for a detailed discussion. Also, there will be

Rows which are not the top or bottom row require a slightly different set of refinement relations. S' and S might contain many complete rows of the torus, and would look something like Figure 7.4 below:

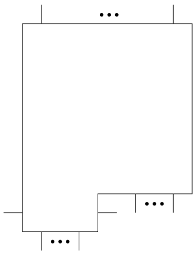


Figure 7.4: *A partially completed torus.*

The addition of a node on to the current row must synchronise both with its neighbour on the row (plural if it is the last node in the row), and with its neighbour above it:

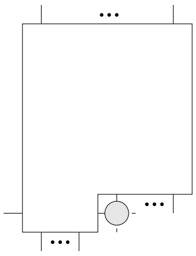


Figure 7.5: *The addition of a node to a partial torus where the current row is neither the top or bottom row.*

$$
\begin{aligned}
Abstraction(S', fc, bfc, lc, aalc, TP, BT) \;\sqsubseteq\; & Abstraction(S, fc, bfc, lc, alc, TP, BT) \\
& [|\{|l, u|\}|]Node(id, u, d, l, r)
\end{aligned}
$$

If the row under construction is the bottom row `BT`, then nodes being added onto `S` must synchronise with nodes in the top row `TP`, in the appropriate manner:

equalities between predicates, for example, if the torus only contains two rows then nodes in the row under construction must either satisfy TP or BT

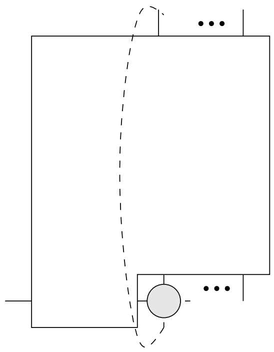


Figure 7.6: *Constructing the bottom row of the torus.*

$$\begin{aligned} Abstraction(S', fc, bfc, lc, aalc, TP, BT) &\sqsubseteq Abstraction(S, fc, bfc, lc, alc, TP, BT) \\ &\quad [||l, u, d||] Node(id, u, d, l, r) \end{aligned}$$

The addition of the final node in the bottom row should complete both the ring of that row, but also the torus itself (see Figure 7.7):

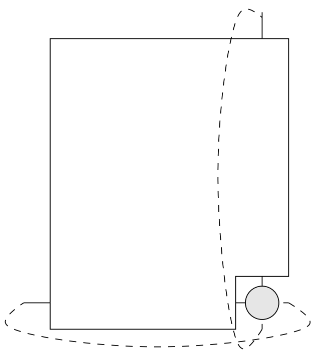


Figure 7.7: *Completing the torus.*

$$\begin{aligned} Abstraction(S', fc, bfc, lc, alc, TP, BT) &\sqsubseteq Abstraction(S, fc, bfc, lc, alc, TP, BT) \\ &\quad [||l, u, d, r||] Node(id, u, d, l, r) \end{aligned}$$

The proof that the parallel composition of nodes in a torus can be represented by a suitable *Abstraction* process requires the following checks:

- A base case of one node, where that node is in the top row.
- Step cases representing the completion of the top row, using the ring construction method described above.
- Step cases representing the construction of all rows which are neither the top or bottom rows. Again, these should be achieved using the ring construction technique, but with an additional synchronisation with nodes in the row above.
- Step cases representing the construction of the bottom and final row. Again, using the ring construction method, only with synchronisations with nodes above, below and to the left of the node.
- A final case where the last node is added to the bottom row; this node synchronises in all four directions.

To establish that the *Abstraction* representation holds true no matter how large the torus is, thresholds must be established on all steps, and then the steps checked for all sizes of the datatype equal to and less than the threshold. Finally, assuming that the desired property is data independent in the type of node names it may be possible to establish a threshold on the check that the *Abstraction* refines the property. However, for even relatively small examples this technique results in thresholds currently too large to be checkable using the FDR tool, (due to the presence of multiple predicates). What follows is an alternative method which breaks the routing problem down into two inductions.

7.2 Technique 2 - Smaller thresholds

Imagine that we want to prove a routing property of our torus, rather like the example presented of a ring topology in Section 6. We want to show that no matter where you input a message into the torus, the message will eventually be output at the desired destination. We cannot show this for all possible senders and receivers at once, the model would become intractably large. Instead, as we did for the ring topology, we shall consider one fixed but arbitrary sender/receiver pair: p and q. The process of routing messages from p to q can be split into two parts:

- Locating the correct position on the x-axis.

- Locating the correct position on the y-axis, and then outputting the message.

The basic method is to establish the location of the point on the x-axis using one induction, then to use a second induction to establish any possible location on the y-axis.

The location of the correct point on the x-axis is basically an induction identical to that for a ring topology. Data independent induction can be used to prove the desired property for any length of row (which is a ring since we are constructing a torus). The property required for a routing protocol is basically that the row behaves as a weak one-place buffer, (from its environment's point of view). If the recipient is on the same row as the message was input, then we would expect the message to be output. If the message recipient is on a different row, then the message will be passed down onto the next row, at the correct x-coordinate. By *weak* we mean that if a second message is input at point p, destined for q, while the first message is still passing through the buffer (somewhere on the row), then the buffer will break. This is necessary to keep the model finite and tractable. Essentially we are assuming the presence of some flow control on messages from p to q, namely that there can only be one outstanding message at any point in time. Figure 7.8 shows a completed row. Recall that as nodes are added to the ring their internal connections

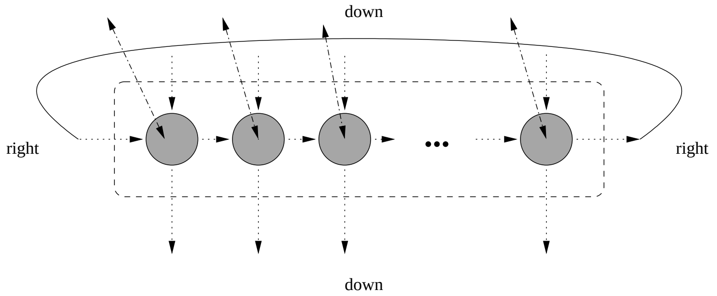


Figure 7.8: *One row of the torus, the x-axis.*

are hidden. What remains are all the communications nodes may make to outside of the row. In this example there are more possibilities since we are working in a different dimension.

When performing the data-independent induction to construct the row it is sufficient to fix the possible number of unique y-coordinates to be 2, i.e. one for the current row, and one for the destination row. If the destination is on the current row then a size of one would be sufficient. The predicate used in the induction identifies those nodes in the set of x-values present in the current row, hence the constants representing the y values will have no effect on the threshold calculations, (they will never satisfy the predicate). The actual values of any explicit y-coordinates in the model can be represented using constants.

The desired property of the row is `DP` where:

```
Spec(S,(px,py),(qx,qy),al,bx) =
  (in.(px,py)?q -> (if q==(qx,qy)
                    then Spec'(S,(px,py),(qx,qy),al,bx)
                    else Spec(S,(px,py),(qx,qy),al,bx)))

Spec'(S,(px,py),(qx,qy),al,bx) =
  if (member((qx,qy),S))
  then (out.(qx,qy).(px,py) ->
          Spec(S,(px,py),(qx,qy),al,bx))
  else (down!bx!(qx,qy)!(px,py) ->
          Spec(S,(px,py),(qx,qy),al,bx))

DP = Spec(S,(px,py),(qx,qy),al,bx)
     [| {in.p.q, out.q.p } |] CHAOS({| in, out |})
```

where the coordinates `(pa,pb)` are those of the point of input, and `(qa,qb)` the destination nodes position. `S` is the set of nodes present in the row; it is only concerned with the x element or the coordinates. This specification says that whenever a message is input at `p` it is eventually output at `q`, or it sent to the next row on channel `down`. It specifies that the x-coordinate at the point it is passed to the row below is the desired x coordinate of the destination: `qa`. If this is the case then we know the message will arrive at subsequent rows already in the correct position on the x-axis.

In order to perform a data independent induction the specification must also allow for messages which have originated at nodes other than p, to be passed onto the row, and likewise messages which have originated at p, but are not destined for q. In all of these cases the specification process ignores the messages and does not change it's state. The specification also captures the buffer breaking when an additional message from p to q is input, before the previous message has left the buffer. We model the buffer breaking by the process willing to do anything, *CHAOS*, since we do not care what happens past this point. The specification must also allow for the events occurring on the `right` channel when the row is incomplete, and all of the possible events on the `down` channel when the torus is incomplete. The specification used in the data independent induction becomes:

```
WBuff(S,(px,py),(qx,qy),al,bx) =
  (in.(px,py).(qx,qy) -> WBuff'(S,(px,py),(qx,qy),al,bx))
  [] -- falls off rhs, only when row incomplete, IGNORE:
  (right!al?dest?s -> WBuff(S,(px,py),(qx,qy),al,bx))
  [] -- message from void (disappears when row complete) IGNORE:
  (right?(px,py)?dest?s -> WBuff(S,(px,py),(qx,qy),al,bx))


WBuff'(S,(px,py),(qx,qy),al,bx) =
  ((member(qx,S) and py==qy) & (out.(qx,qy).(px,py) ->
          WBuff(S,(px,py),(qx,qy),al,bx)))
  [] -- falls off rhs, only when row incomplete:
  (right!al?dest?s -> WBuff(S,(px,py),(qx,qy),al,bx))
  [] -- passed down to row beneath:
  ((member(qx,S) and py!=qy) &
       (down!bx!(qx,qy)!(px,py) ->
              WBuff(S,(px,py),(qx,qy),al,bx)))
  [] -- break the buffer:
  (in.(px,py).(qx,qy) -> CHAOS({| in, out |}))
  [] -- message from void (disappears when row complete) IGNORE:
  (right!(px,py)?dest?s -> WBuff'(S,(px,py),(qx,qy),al,bx))

Abstraction(S,p,q,al,bx) =
   WBuff(S,p,q,al,bx)
   [| {| in.p.q, out.q.p |} |]
   CHAOS({| in, out |})
```

A corresponding node process looks like:

```
N(n,al,d) =
  (in.n?dest -> N'(n,dest,n,al,d))
  []
  (right!n?dest?sender -> N'(n,dest,sender,al,d))
  []
```

```
  (down!n?dest?sender -> N'(n,dest,sender,al,d))

N'(n,dest,s,al,d) =
  (if getx(n)!=getx(dest)
   then right!al!dest!s -> N(n,al,d)
   else (if gety(n)!=gety(dest)
         then down!d!dest!s -> N(n,al,d)
         else out!n!s -> N(n,al,d)))

alph(n,r,d) = {| in.n, out.n, right.n,
               down.n, right.r, down.d |}
```

Notice, when the row is complete all of the events on the `right` channels in the row disappear. If we lazily abstract all of the `down` events that we are not concerned with, we can show that our row refines the desired specification.

We now need to add on the other rows, starting from the row `b` which we have just constructed. In fact, we can view the addition of the rows as the creation of another ring along the y-axis. In particular, the nodes which will be linked are precisely those with the x coordinate matching that of the destination.

Consider Figure 7.9 below:

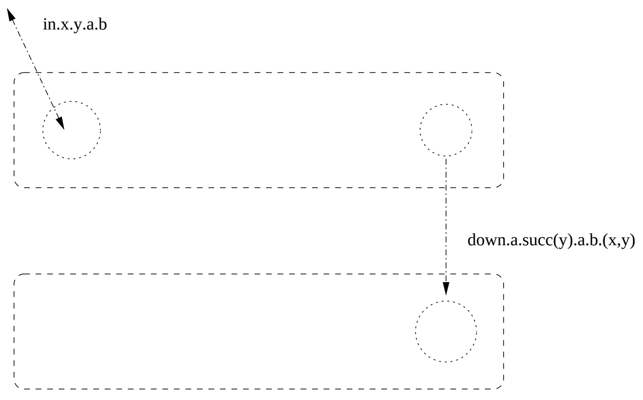


Figure 7.9: *A message is passed to the row below.*

Because we can rely on the output from row `b` to be on the desired point on the x-axis we only need to connect the two rows in one position: at `b`. We know that the message will not be passed around the row, it will either be output (since it has reached its destination, or it will be passed down a further row). This pattern will continue until the node reaches its destination. Linking the rows only at the necessary point on the x-axis results in a ring over the y-axis, where the nodes in the ring are whole rows:

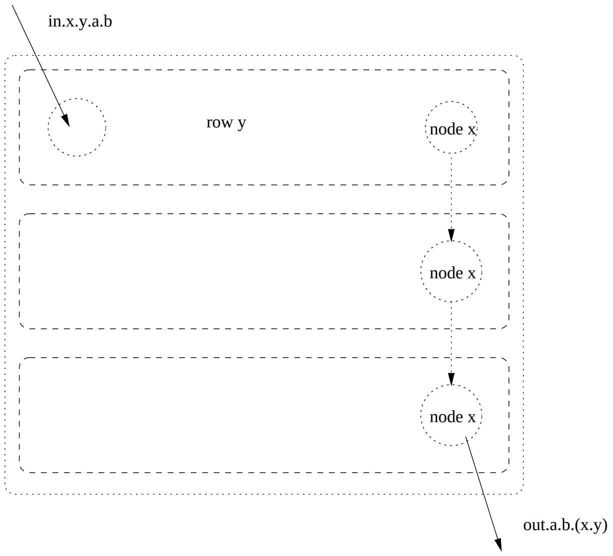


Figure 7.10: *The connection of the rows in a torus for a particular destination node.*

Clearly, if we show this for any arbitrary sender and receiver, and use data independence results to prove this no matter how large, in either dimension, the torus is, then we have shown the routing property is true of any arbitrary torus.

We construct a second data independent induction, again identical in its construction to that of a ring. However, we will have a new predicate containing the names of all the rows present in the partial system (corresponding to the y elements of the coordinates of nodes in the column). The nodes in this induction are processes representing rows. Their behaviour is actually very simple since only one node will ever be active. It is sufficient to connect just one node in the correct column, and lazily abstract all of its possible communications on its row: `right`. The lazy Abstraction means that we are assuming nothing about what messages may or may not be passed to the node along the row. The node will take in the message on the channel connecting it to the row above, and either output the message since the message has reached its destination, or the node will send the message to the row below on channel `down`. We then perform a second data independent induction creating a ring of nodes in the y-axis. An example inductive step would be:

```
lazyNODE((id,al,d)) =
  (N(id,al,d)
  [| {| right.id, right.al |} |]
  CHAOS({| right.id, right.al |})) \ {| right.id, right.al |}

assert Abstraction({0,1},(0,0),(3,3),(0,0),(3,2)) [T=
            (Abstraction({0},(0,0),(3,3),(0,0),(3,1))
            [ down.(3,1) <-> down.(3,1) ]
            lazyNODE((3,1),(4,1),(3,2)))
```

where `lazyNODE((id,al,d))` is the lazy Abstraction of the `right` channels of the node being added to the network. When we have added the final node, completing the ring on the y-axis, we can lazily abstract all the remaining `down` events, and our network will refine `DP`.

It is interesting to note that we have shown by induction that the overall torus refines:

```
Spec(S,(px,py),(qx,qy),al,bx) ||| CHAOS(A)
```

and by symmetry:

```
Spec(S,(qx,qy),(px,py),al,bx) ||| CHAOS(B)
```

where `A` and `B` are all the external events of the torus not devoted to communications from `p` to `q` or `q` to `p`. It follows that, so long as we implement a flow control on the messages passing between `p` and `q` ensuring that only one message may be present on the network at any one time, the torus refines:

```
(Spec(S,(px,py),(qx,qy),al,bx) |||
   Spec(S,(qx,qy),(px,py),al,bx)) |||
      CHAOS(inter(A,B))
```

where `inter(A,B)` is the intersection of the sets `A` and `B`.

As with any data independent induction, the applications this may be applied to are limited by process size. Certainly this method for constructing toroid is preferable to the previous, since this technique requires only one predicate to be active in the induction at any one time. Further, the *Abstraction* used in the induction presented above is bespoke for the routing algorithm. In fact, we do not construct a complete torus, but simply the connections relevant to the property in question. One could imagine many other properties which would not allow such an abstraction, and would be currently out of scope of the tool support. Grids could be constructed using a similar method.

Chapter 8

Conclusions and further work

8.1 The Value of Data Independent Induction

The work presented in this thesis is an attempt to address one of industry's concerns: that formal methods cannot produce results for systems of realistic size and complexity. Model-checking is an important formal technique, since it is a form of tool support not only capable of positive verifications of system specifications, and also because where proofs fail it provides de-bugging information. This information makes model-checking applicable throughout the development life-cycle; it can help remove errors early in system development. But model-checkers can only reason about systems of a finite size, making it impossible to model check a network of arbitrary size directly. This thesis has presented a technique that allows the FDR model-checking tool to produce results for networks of arbitrary size. The Data Independent Induction method uses the model checker to discharge proof obligations of arguments which combine induction and data-independence results.

The mathematical notion of induction is a powerful one, as demonstrated by the structural induction technique for CSP described in Chapter 2. Many large structures constructed from identical components can be described in terms of a set of base and step cases iteratively combined together. Structural induction can handle a finite range of different components (as witnessed in the monument example). When using CSP to describe networks and the nodes they contain, we have to also describe the events which these nodes are willing to communicate to each other. This makes induction over components who need to know of each others' identities, (to address messages to each other for example), impossible. When new nodes are added to the network the induction cannot provide the knowledge of their existence to the nodes already present. Data Independent Induction fuses induction and data-independent reasoning together to facilitate reasoning about such systems. In particular, Data independent induction produces results for arbitrary sized networks over certain classes of network topology. As systems are increasingly formed from networked processors, to connect desktop computers or embedded control systems, the ability to produce results which will hold as these networks grow becomes more useful.

The two main challenges in producing a successful Data Independent Induction appear to be the construction of a suitable *Superstate* process, and the calculation of thresholds small enough to allow automatic verification using FDR. Current work on developing FDR capabilities for automatically generating thresholds on checks involving data-independent processes should eventually eliminate the need to calculate thresholds. (Either the FDR tool will automatically calculate thresholds, or it may be that a symbolic system is developed which will mean that the data-independent type need not be instantiated at all.)

8.2 Further Work

More complex topologies This thesis has demonstrated how Data Independent Induction might be used to produce results for various network topologies: totally connected, star topology, multiplexed buffers, ring, grids and torus. For applicable properties (such as deadlock-freedom) it should be possible to combine systems with different topologies together, either using structural induction or data independent induction. In Chapter 5 data independent induction was used to prove properties of a node independent of the branching degree on its downstream channel. This was then combined with structural induction to obtain proofs for all arbitrary branching multi-cast trees. In a manner similar to this one could treat networks as components of larger networks, see Figure 8.1:

Automatic generation of *Abstractions* Currently the construction of the network invariant which can be used in the data independent induction to imply the desired overall property is the result of ingenious coding. In many cases clever

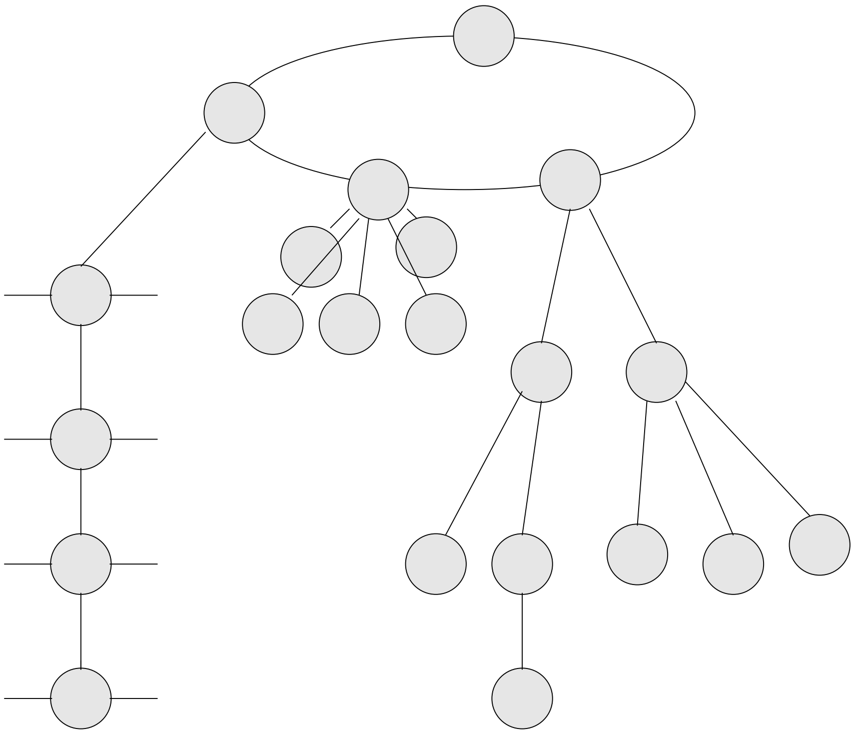


Figure 8.1: *Networks of networks.*

modeling and coding is required to make the systems small enough for FDR to handle. Constant development of the FDR tool, and computers with larger memories and faster processing speeds, will ensure that more and more complex proofs will be possible. Perhaps this will have side effects leading to a reduction in the need for so much ingenuity in the coding. It is also possible that for certain families of properties and applications, a tool which automatically generates a form of invariant might be constructed, just as the Casper tool generates CSP models of cryptographic protocols for FDR to process.

The invariant process *Abstraction* can be a complete description of the sub-network (modulo any don't care conditions you need), or very abstract. The more abstract it is, the harder it will be to automatically generate. One could try observing the sub-networks working, identify the states they go through, use these to construct the invariant, and then add any don't care clauses required (while the network is incomplete).

Alternatively, one could start from a description of one node and try to generalise it. This is the approach taken in the construction of the *Abstraction* process used in the Time-Triggered Protocol analysis. In general, if the refinement fails then you can locate the failure, and introduce a don't care clause or modify *Abstraction*. There will always be a certain amount of human ingenuity required in the construction of the *Abstraction* process.

Recent work by Rushby [73] on using diagramatic methods to construct models of whole systems behaviour, rather like the *Superstate* constructed in the TTP case study, seems promising. His visual method uses a form of Labelled Transition System which could conceivably easily be converted to CSP. This would make the technique more accessible to those who would rather think in pictures than numbers.

More complex nodes Recent extensions to the theory of data independence will facilitate the capture of more complex node behaviours. An extension of the theory to allow data independent programs to contain arrays indexed over the type, and containing members of any other type which may or may not be the same, is presented in [52]. This work could lead to the nodes in systems being able to differ in more subtle ways than just their identities. Currently Data Independent Induction requires that certain aspects of node's behaviour are similar (we may not be concerned with all behaviours of a node), distinguishing exceptions through the use of constants and predicates. One application of arrays might be to allow nodes to vary their behaviour with respect to each other, using the array to capture state information for each node. This would mean that nodes need not be in identical states. Where fault models are used different nodes might be in varying states of failure. For example, in communication protocols incorporating a concept of rounds it may be possible to vary the rounds that each node is in with respect to the protocol.

The application domain of Data Independent Induction will constantly be expanded whilst there is development of the theory of data independence. As the theory enables more complex data structures to be used in processes, so will it allow a wider variety of systems to be reasoned about and be proven true independent of their size, using Data Independent Induction.

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)`.

Appendix B

CSP$_M$ for The Monument

```
--Monument example induction - last modified 15-12-97

--The induction will show that arbitrary configurations of roadway made
--up of certain types of road will still maintain the property of
--deadlock-freedom over the whole network.

--To establish the Basecase FDR is used to check that the bit of road
--that goes around the monument is deadlock free.

--Using FDR it can also be shown that:

--'A deadlock free network' || **

--is deadlock free.  Which will establish, by normal inductive
--reasoning, that the addition of any number of road blocks of type **
--will still produce a deadlock-free system.

--The Basecase will be the road around the monument.

--Letting the road around the monument be made up of only road segment
--of type A.




-- Check1: DF [F= Basecase. Do not need an 'assert' command, we can
--just get FDR to check that A is deadlock-free.
```

```
Basecase = A


--To get the inductive step need a process which represents deadlock
--freedom:

DF =    in -> DF
|~|
out -> DF

--Each of the following checks produces rules for the construction of
--allowable road networks.  In each case the actual refinement check is
--performed by asking FDR to check whether the implementation is
--deadlock free, which if true allows us to claim [F= DF.

--Check2: DF [F= DF || EmptyA
--This allows the road to be made up of segments of type A.

Acase = DF [in <-> out] A


--Check3: DF [F= DF || EmptyB
--This allows the road to be made up of segments of type B.

Bcase = DF [out <-> in] B


--Check4: DF [F= DF || CD
--This allows the road to be made up of segments of type C, the narrow
--ones, so long as there is a one way of type D attached to its
--leftwards out direction.

CDcase = CD [leftout <-> in, leftin <-> out] DF


--Check5:  DF [F= DF || DF
--Any two allowable networks joined together where the out-road of one
--is joined to the in road of the other.

DFNETcase = normal(DF [ in <-> out ] DF)
```

```
-- FDR code for the road segments.

channel right, left, leftin, leftout, in, out
channel OK

transparent normal

--process describing segment of type A.

A  = in -> out -> A

-- process describing segment of type B.

B  = in -> out -> B

--Process describing segment of type C: the pass between the rocks.
--An extra communication of OK is required before a car can move
--leftwards into segment C, this confirms that D is empty.  Note that it
--only ever moves back into the C state after an OK event, and therefore
--only takes cars in on its ‘in‘ channel after this comm.

C  = leftin -> out -> C
     []
     in -> leftout -> C’
C’ = OK -> C
     []
     leftin -> C’’
C’’ = out -> C’
      []
      OK -> out -> C

--Segment D is defined so that it communicates OK as soon as it
--becomes empty, and before it is willing to accept another input. This
--way it is possible to restrict segment C from taking in another car to
--only when it has been informed that segment D is empty, by an OK
--event.

D = in -> out -> OK -> D

-- CD segment:

CD = C [leftout <-> in, OK <-> OK] (D [[out <- leftout]])
```

Appendix C

CSP$_M$ for Leadership Election

```
-- Consider a set of
-- processes that require a leader to exist from amongst them, whose job it
-- is in turn to distribute a token 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 token "idle" is in the leader.....

-- The parameter P in the following processes is the type of nodes in use;
-- kept variable as we need nodes to know who is and who is not present
-- in a given check.


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

Empty(P,id,j) = pass.j.id-> Full(P,id,j)
             [] leader?j' -> (if j'==id then Leader(P,id) else Empty(P,id,j') )

Full(P,id,j) = pass.id.j -> Empty(P,id,j)
             [] cr_start.id -> cr_end.id -> Full(P,id,j)

Alpha(P,id) = Union({{|leader|},
                   {cr_start.id, cr_end.id},
                   {pass.id.j, pass.j.id | j <- P}})

-- If R 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 R as

LNet(P,R,L0) = || i:R @ [Alpha(P,i)] (if i==L0 then Leader(P,i)
                                       else Empty(P,i,L0))
```

Chapter C $\mathbf{CSP}_M$ for Leadership Election

```
-- Channel etc declarations...

Players = {1..8}

channel leader, cr_start, cr_end:Players

channel pass:Players.Players

others(i,P) = diff(P,{i})



-- This model completely abstracts away from the leadership election process,
-- which we delegate here to a somewhat unrealistic multi-way synchronisation.

-- We might reasonably hope that the complete system is equivalent
-- to the following process:

State(P,L) =  leader?j:P -> State(P,j)
             [] cr_start.L -> cr_end.L -> State(P,L)
             [] pass!L?j:others(L,P) -> State'(P,L,j)

State'(P,L,j) =  cr_start.j ->  cr_end.j -> State'(P,L,j)
             [] pass.j.L -> State(P,L)

-- and so it is for a complete system with all the nodes present, for
-- example

assert State({1..5},1) [FD= LNet({1..5},{1..5},1)

assert State({1..5},1) [FD= LNet({1..5},{1..3},1)

-- Unfortunately a partially constructed network can get quite
-- badly behaved, in particular allowing any number of nodes to
-- get into critical sections at once, so we have to be a little
-- cunning in devising our inductive hypothesis, which essentially is
-- that things stay fine as long as no imaginary leader outside R
-- hands out too many tokens.....the following process deals with this
-- and otherwise reflects all the proper behaviour in the alphabets
-- of processes in R

Abstraction(P,R,L) =  leader?j:P -> Abstraction(P,R,j)
             [] member(L,R) & cr_start.L -> cr_end.L -> Abstraction(P,R,L)
             [] member(L,R) & pass!L?j:others(L,P) -> Abstraction'(P,R,L,j)
             [] not member(L,R) & pass!L?j:others(L,R) -> Abstraction'(P,R,L,j)

Abstraction'(P,R,L,j) =  member(j,R) & cr_start.j -> Abstraction''(P,R,L,j)
                 [] pass.j.L -> Abstraction(P,R,L)
```

```
                 [] not member(L,R) & (STOP |~| pass.L?k:R -> CHAOS(Events))

Abstraction''(P,R,L,j) =  cr_end.j -> Abstraction'(P,R,L,j)
                 [] not member(L,R) & (STOP |~| pass.L?k:R -> CHAOS(Events))


-- The inductive claim is then that the partially completed network
-- refines this:

-- assert Abstraction(P,R,L0) [FD=
--        Abstraction(P,diff(R,{k}),L0)[Alphas(diff(R,{k})||Alpha(P,k)] Empty(P,k,L0)


Alphas(P,R) = Union({Alpha(P,n) | n <- R})

-- Threshold for the inductive step is 6, 3 in S, 3 outside.
-- The biggest of a long series of checks we apparently
-- have to do is then:

assert Abstraction({1..6},{1..4},1) [FD=
       Abstraction({1..6},{1..3},1)
       [Alphas({1..6},{1..3})||Alpha({1..6},4)] Empty({1..6},4,1)

-- The base case we require is

-- assert Abstraction(P,{L0},L0) [FD=  Leader(P,L0)

-- The threshold is 3. Example check:

assert Abstraction({1..3},{1},1) [FD=  Leader({1..3},1)

-- Which readily leads to an inductive proof that for all finite P,R,L0
-- (L0 in R)

-- Abstraction(P,R,L0) is refined by LNet(P,R,L0)

-- which in turn gives us what we want as clearly Abstraction(P,P,L0) = State(P,L0)
```

Appendix D

CSP$_M$ for Multiplexed Buffers

```
-- mbuff with data independent induction

-- Our objective is to prove that the buffer works
-- whatever the type of data and however many channels are
-- multiplexed.....

datatype DATA = Zero | One

N = 7

TAGS = {1..N}

channel left, a, b, right : TAGS . DATA
channel c, d : TAGS
channel fwd : TAGS . DATA
channel back : TAGS

-- The following four processes form the core of the system
--
--
--    --> SM -->  ....fwd....   --> RM -->
--
--    <-- RA <--  ...back....   <-- SA <--
--
-- SM and RM send and receive tagged messages, while
-- SA and RA send and receive acknowledgements.

SM = [] i : TAGS @ (a . i ? x -> fwd ! i . x -> SM)

RM = fwd ? t ? x -> b . t ! x -> RM
```

```
SA = [] i : TAGS @ (c . i  -> back ! i -> SA)

RA = back?t?x -> d . t ! x -> RA

-- These four processes communicate with equal numbers of  transmitters (T)
-- and receivers (R), which in turn provide the interface with the environment.

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: here without added
-- intermediate buffering.

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

-- Our objective is to prove that the complete system is the composition
-- of a COPY for each tag.  A direct proof of this does not lead to
-- finite thresholds, but we can show that the abstraction to an
-- arbitrary lane is right by induction, and deduce the overall result from
-- that.  The lane we concentrate has tag C0

C0 = 1

-- We then progressively add further T(i)/R(i) combinations onto the
-- following obvious base system.

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

-- The following is intended to be a specification of how a partially
-- constructed system looks once all the external channels it has, other
-- than C0 ones, are lazily abstracted: it is built in three parts, one
-- dealing mainly with forward transmission of messages of channels not
-- yet present, one for corresponding acks, and one to record
-- commmunications for C0, which act like COPY though it can refuse
-- these communications if there are other messages/acks around.

SPEC1(S) = let SC = diff(TAGS,S) within
        left.C0?y -> SPEC1(S)
        [] right.C0?y -> SPEC1(S)
        [] a?i:SC?y -> SPEC1'(S,i,y)

SPEC1'(S,i,y) = let SC = diff(TAGS,S) within
        (STOP |~| (left.C0?z -> SPEC1'(S,i,y)
        [] right.C0?z -> SPEC1'(S,i,y)))
        [] b.i.y -> SPEC1(S)
        [] (STOP |~| a?j:SC?z -> SPEC1''(S,i,y,j,z))
```

```
SPEC1''(S,i,y,j,z) =
        (STOP |~| (left.C0?t -> SPEC1''(S,i,y,j,z)
                   [] right.C0?t -> SPEC1''(S,i,y,j,z)))
        [] b.i.y -> SPEC1'(S,j,z)

SPEC2(S) = let SC = diff(TAGS,S) within
        left.C0?y -> SPEC2(S)
        [] right.C0?y -> SPEC2(S)
        [] c?i:SC -> SPEC2'(S,i)

SPEC2'(S,i) = let SC = diff(TAGS,S) within
        (STOP |~| (left.C0?z -> SPEC2'(S,i)
                   [] right.C0?z -> SPEC2'(S,i)))
        [] d.i -> SPEC2(S)
        [] (STOP |~| c?j:SC -> SPEC2''(S,i,j))

SPEC2''(S,i,j) = (STOP |~| (left.C0?t -> SPEC2''(S,i,j)
        [] right.C0?z -> SPEC2''(S,i,j)))
        [] d.i -> SPEC2'(S,j)

SPEC3 = left.C0?x -> right.C0!x -> SPEC3

SPEC(S) = (normal(SPEC1(S)) [|{|left.C0,right.C0|}|] normal(SPEC2(S)))
          [|{|left.C0,right.C0|}|] normal(SPEC3)

-- Above 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 T on the a/b or c/d channel.  Could be formulated as a
-- single sequential process with quite a lot of states.

--  The claim is that a network constructed from CORE plus all the T(i) and
--  R(i) from i in R, all internal communication hidden and all
--  channels in R other than C0 lazily abstracted, refines SPEC(R).

-- Base case, threshold is 8:

assert SPEC({C0}) [F= BASE

-- The inductive step required to prove this is
-- that if we add a further T(K)/R(K), hiding internal and abstracting
-- external communications appropriately, we preserve SPEC, threshold is 7:

assert SPEC(St) [F= (LAbs(SPEC({1}) [| intevs |]
                     (T(K)|||R(K)))({|left.K,right.K|})) \ intevs


intevs = {|a.K,b.K,c.K,d.K|}
St = {1,2}
```

```
K = 2


transparent normal

LAbs(P)(X) = (P [|X|] normal(CHAOS(X)))\X

-- Finally, it is as well to show that the complex definition of SPEC above
-- reduces to COPY when the predicate encompasses all lanes....

assert COPY(left.C0,right.C0) [FD= SPEC(TAGS)

COPY(in,out) = in?x -> out!x -> COPY(in,out)
```

Appendix E

CSP$_M$ for the TTP

E.1 TTP node

```
slot = {0..(size-1)}

size = 4


-- local membership sets are sets of processors, where processors are
-- identified by their slot values.


memset = Set(slot)

channel broadcast, isin  : slot.Bool
channel tock
channel membership: memset
channel memb
channel take: slot
channel faulty: slot
channel selfdiag: slot


Ignore(S) =     ([]x:S @ x -> Ignore(S))
                [] tock -> SKIP


transparent normal

compress(P) = normal(P)
```

```
NODE(i) = (compress((compress(node(i,0,true,false))
                             [[memb <- membership.lms | lms <- memset]]
          [| {| faulty,isin,take,membership |} |]
          SET(i,slot,false,false))\{|isin,take|}))

node(i,bslot,ack,flag) = memb -> N(i,bslot,ack,flag)


N(i,bslot,ack,flag) =
         (if bslot==i then isin!i?r -> (if r then
                                (broadcast!i!ack -> tock ->
                                 node(i,(bslot+1)%size,true,flag))
                                [] (faulty!i -> tock ->
                                 node(i,(bslot+1)%size,true,true))
[] flag & (tock ->
 node(i,(bslot+1)%size,true,flag))
                             else tock ->
                                 node(i,(bslot+1)%size,ack,flag))
        else  broadcast!bslot?a -> isin!bslot?r -> (if r then
                                   if a and (ack==false)
                                   then take!i -> tock ->
                                        node(i,(bslot+1)%size,true,flag)
                                   else if (ack==true) and (a==false)
                                         then take!bslot -> tock ->
                                         node(i,(bslot+1)%size,false,flag)
                                        else tock -> node(i,(bslot+1)%size,
                                            if (a==false) and (ack==true)
                                                then false else true,flag)
                                  else tock ->
                                  node(i,(bslot+1)%size,ack,flag))
                []
                not flag & faulty!i -> isin!bslot?r ->
                                ((if r
                                 then (if ack==false
                                      then take!i -> take!bslot ->
                                        SKIP
                                      else take!bslot ->
                                        SKIP)
                                 else SKIP);
                               (Ignore({|broadcast|});
                                (if r
                                 then
                                 node(i,(bslot+1)%size,false,true)
                                 else node(i,(bslot+1)%size,ack,true))))
  []
                 flag &  isin!bslot?r ->  -- receive faulty when already faulty
                                ((if r
                                 then (if ack==false
                                      then take!i -> take!bslot ->
```

```
                                    SKIP
                                  else take!bslot ->
                                    SKIP)
                             else SKIP);
                           (Ignore({|broadcast|});
                            (if r
                             then
                             node(i,(bslot+1)%size,false,flag)
                             else node(i,(bslot+1)%size,ack,flag))))
                []
                tock -> isin!bslot?r -> (if r
                                         then if ack==false
                                              then take!i -> take!bslot ->
                                               node(i,(bslot+1)%size,false,flag)
                                              else take!bslot ->
                                               node(i,(bslot+1)%size,false,flag)
                                         else node(i,(bslot+1)%size,ack,flag)))


SET(i,lms,fault,sd) =      isin?x!member(x,lms) -> SET(i,lms,fault,sd)
                        [] take?x -> SET(i,diff(lms,{x}),fault,sd)
                        [] not fault & membership!lms -> SET(i,lms,fault,sd)
                        [] fault & membership?Y -> SET(i,lms,fault,sd)
                        [] not fault & faulty!i -> SET(i,lms,true,sd)
[] ((not member(i,lms)) and not sd) &
 selfdiag!i -> SET(i,lms,fault,true)
```

E.2 Optimised TTP node

```
slot = {0..(size-1)}

size = 3

memset = Set(slot)

channel broadcast, isfaulty  : slot.Bool
channel tock
channel notmembers: memset
channel memb
channel add: slot
channel faulty: slot
channel whosego: slot
channel selfdiag: slot

Ignore(S) =     ([]x:S @ x -> Ignore(S))
                [] tock -> SKIP
```

```
transparent normal

compress(P) = normal(P)

NODE(i) = (compress((compress(node(i,true,false))
                [[memb <- notmembers.faultyones | faultyones <- memset]]
          [| {| faulty,isfaulty,add,notmembers |} |]
          SET(i,{},false,false))\{|isfaulty,add|}))


node(i,ack,flag) = memb -> N(i,ack,flag)

N(i,ack,flag) =
            ( whosego?bslot ->
                if bslot==i then isfaulty!i?r -> (if r==false then
                                (broadcast!i!ack -> tock ->
                                 node(i,true,flag))
                                [] (faulty!i -> tock ->
                                 node(i,true,flag))
[] (flag & tock ->
node(i,true,flag))
                             else tock ->
                                 node(i,ack,flag))
        else  broadcast!bslot?a -> isfaulty!bslot?r -> (if r==false then
                                   if a and (ack==false)
                                   then add!i -> tock ->
                                        node(i,true,flag)
                                   else if (ack==true) and (a==false)
                                         then add!bslot -> tock ->
                                         node(i,false,flag)
                                        else tock -> node(i,
                                            if (a==false) and (ack==true)
                                                then false else true,flag)
                                  else tock ->
                                  node(i,ack,flag))
                []
                not flag & faulty!i -> isfaulty!bslot?r ->
                                ((if r==false
                                 then (if ack==false
                                      then add!i -> add!bslot ->
                                        SKIP
                                      else add!bslot ->
                                        SKIP)
                                 else SKIP);
                               (Ignore({|broadcast|});
                                (if r==false
                                 then
                                 node(i,false,true)
```

```
                                else node(i,ack,true))))
                []
flag & isfaulty!bslot?r ->
                               ((if r==false
                                then (if ack==false
                                     then add!i -> add!bslot ->
                                       SKIP
                                     else add!bslot ->
                                       SKIP)
                                else SKIP);
                              (Ignore({|broadcast|});
                               (if r==false
                                then
                                node(i,false,flag)
                                else node(i,ack,flag))))
[]
                tock -> isfaulty!bslot?r -> (if r==false
                                           then if ack==false
                                                then add!i -> add!bslot ->
                                                 node(i,false,flag)
                                                else add!bslot ->
                                                 node(i,false,flag)
                                           else node(i,ack,flag))
        )


SET(i,faultyones,fault,sd) =
   isfaulty?x!member(x,faultyones) ->
                       SET(i,faultyones,fault,sd)
   [] add?x -> SET(i,union(faultyones,{x}),fault,sd)
   [] not fault & notmembers!faultyones ->
                       SET(i,faultyones,fault,sd)
   [] fault & notmembers?Y -> SET(i,faultyones,fault,sd)
   [] not fault & faulty.i -> SET(i,faultyones,true,sd)
   [] ((not sd) and member(i,faultyones)) & selfdiag!i ->
                      SET(i,faultyones,fault,true)
   [] ((not sd) and card(faultyones)>(size-2)) & selfdiag!i ->
                  SET(i,union(faultyones,{i}),fault,true)


FAULTORDER({},0) = faulty.0 -> FAULTORDER({0},1)

FAULTORDER(seen,bigg) = faulty.bigg ->
               FAULTORDER(union(seen,{bigg}),(bigg+1)%size)
```

E.3 TTP abstraction process

```
ABSTRACTION(S,oldbads) = Abstraction(S,oldbads) [| {| faulty |} |] CHAOS({|faulty|})


Abstraction(S,oldbads) =
let
NOTFAULTY(oldbads,newbads,ack,repfault) =
if card(newbads)>1 then BOT else
 (if card(union(oldbads,newbads))>size-2 then BOT else
(if S<=union(oldbads,newbads) then
agreedfault?_ -> NOTFAULTY'(oldbads,newbads,ack)
else agreedfault!repfault -> NOTFAULTY'(oldbads,newbads,ack)))

-- NOTFAULTY is the state when there are no faulty
-- processors waiting to selfdiagnose.

NOTFAULTY'(oldbads,newbads,ack)=
        whosego?bslot -> NOTFAULTY''(oldbads,newbads,bslot,ack)

NOTFAULTY''(oldbads,newbads,whose,ack) =
       (member(whose,S)) &    -- no faults
          (if member(whose,union(oldbads,newbads))
           then tock -> NOTFAULTY(oldbads,newbads,ack,{})
           else broadcast!whose!ack -> tock ->
             NOTFAULTY(oldbads,newbads,true,{}))
       [] (member(whose,S)==false) &
          (if not(member(whose,union(oldbads,newbads))) then
           (broadcast!whose?b -> tock ->   -- no fault
             (if b!=ack
              then (if (ack==false and b)
                    then BOT
                    else (if (ack and b==false)
     then NOTFAULTY(oldbads,union(newbads,union(oldbads,{whose})),
false,{whose})
                         else NOTFAULTY(oldbads,newbads,false,{})))
                    else NOTFAULTY(oldbads,newbads,true,{}))
                [] tock -> (if ack
       then NOTFAULTY(oldbads,union(newbads,union(oldbads,{whose})),
false,{whose})
                                        else BOT))
                                -- send fault
                else (tock -> NOTFAULTY(oldbads,newbads,ack,{})
  [] broadcast!whose?a -> tock -> NOTFAULTY(oldbads,newbads,ack,{})))
        [] faulty?m:diff(S,union(oldbads,newbads)) ->
     (if not ack then BOT else   -- a new faulty
  (if member(whose,union(oldbads,newbads)) -- don't care if
                    then BOT      -- you go faulty in a faulty slot.
```

```
                else ((m!=whose) & (
                   let HF = (
                   faulty?x:S -> BOT  -- only one fault allowed.
                   [] SKIP)
                    within HF ;
               (if member(whose,S)
                 then broadcast!whose!ack -> tock -> -- receive fault
                  FAULTY(oldbads,newbads,{whose},true,m,false,
         union(union(oldbads,{whose}),newbads),{})
                  else (broadcast!whose?b -> tock -> -- receive fault
                         (if b!=ack
                           then BOT   -- going faulty in a faulty slot
                              else
       FAULTY(oldbads,newbads,{whose},true,m,false,
union(union(oldbads,{whose}),newbads),{}))
                    [] tock ->   -- whose is faulty
              BOT)))  -- can't go faulty in a faulty slot, so don't care.
                   [] (m==whose) & ((faulty?x:S -> BOT
                                    [] SKIP);
                                tock -> -- send fault
   FAULTY(oldbads,union({m},newbads),{},false,
                   m,true,union(oldbads,newbads),{m})))))


-- if the fault was a send fault then the processor is already in the set
-- bads.  However,if the fault was a receive fault then this won't occur
-- until after the next broadcast slot of the processor, since that is the
-- only way that the diagnosis can be made.


FAULTY(oldbads,newbads,newbads',ack,dying,ack',bads',repfault) =
if card(newbads)>1 then BOT else
(if card(union(oldbads,newbads))>size-2 then BOT else
(if (member(dying,newbads) and member(dying,bads'))
then NOTFAULTY(oldbads,union(newbads,{dying}),ack,repfault)
              -- only return to notfaulty if selfdiagnosis and diagnosis
                        -- have both occurred.
        else (if card(bads') >= size-1 and not member(dying,bads') then
           FAULTY(oldbads,newbads,union(newbads',{dying}),ack,dying,
ack',union(bads',{dying}),repfault)
 else
         (if S <= union(union(oldbads,newbads),{dying}) then
        agreedfault?_ -> FAULTY'(oldbads,newbads,newbads',
                                     ack,dying,ack',bads')
        else
        agreedfault!repfault ->
 FAULTY'(oldbads,newbads,newbads',
                                ack,dying,ack',bads')))))
```

```
FAULTY'(oldbads,newbads,newbads',ack,dying,ack',bads') =
        whosego?bslot -> if member(bslot,union(oldbads,newbads))
             then FAULTY'''(oldbads,newbads,newbads',
                                  bslot,ack,dying,ack',bads') -- bad
             else FAULTY''(oldbads,newbads,
                           newbads',bslot,ack,dying,ack',bads') -- good


FAULTY''(oldbads,newbads,newbads',whose,ack,dying,ack',bads') =
         (if (whose==dying)   --dying slots turn, not diagnosed yet.
          then  (if not(member(dying,bads'))
                then ((broadcast!whose!ack' -> tock ->
                        (if (ack and (ack'==false))
             then FAULTY(oldbads,union(newbads,{whose}),newbads',
                                false,dying,true,bads',{whose})
               else FAULTY(oldbads,newbads,newbads',
                               true,dying,true,bads',{})))
                      []  tock ->
                  FAULTY(oldbads,union(newbads,{whose}),
                               newbads',false,dying,true,bads',{whose}))
      else tock -> FAULTY(oldbads,union(newbads,{whose}),
                         newbads',false,dying,true,bads',{whose}))
          else (if member(whose,S)
                then
-- DON'T NEED TO PREVENT FAULTY PROCESSORS FROM
-- BROADCASTING SINCE IF IT WERE
-- A FAULTY PROCESSORS GO WE WOULD BE IN FAULTY'''.
          broadcast!whose!ack -> tock ->
             (if member(whose,bads')
              then FAULTY(oldbads,newbads,newbads',true,
                                 dying,ack',bads',{})
                     else    (if (ack and (ack'==false))
                              then (if not member(dying,bads')
   then selfdiag!dying ->
FAULTY(oldbads,newbads,union(newbads',{dying}),
                      true,dying,true,union(bads',{dying}),{})
     else
FAULTY(oldbads,newbads,union(newbads',{dying}),true,
                      dying,true,union(bads',{dying}),{}))
                              else    (if (ack' and (ack==false))
                                                then
FAULTY(oldbads,newbads,union(newbads',{whose}),true,
                             dying,false,union(bads',{whose}),{})
                                                else
     FAULTY(oldbads,newbads,newbads',true,
                      dying,if (ack' and (ack==false))
                            then false
```

```
                              else true,bads',{}))))
                else (broadcast!whose?a -> tock ->
                          -- good slots turn outside of S
                    (if a==ack
                    then  (if member(whose,bads')
                           then FAULTY(oldbads,newbads,newbads',
                                 true,dying,ack',bads',{})
                           else    (if (ack and (ack'==false))
                              then (if not member(dying,bads')
                                   then selfdiag!dying ->
FAULTY(oldbads,newbads,union(newbads',{dying}),
              true,dying,true,union(bads',{dying}),{})
else FAULTY(oldbads,newbads,newbads',
true,dying,true,bads',{}))
                          else    (if (ack' and (ack==false))
                                then
           FAULTY(oldbads,newbads,union(newbads',{whose}),
                    true,dying,false,union(bads',{whose}),{})
                           else
               FAULTY(oldbads,newbads,newbads',true,dying,
                           if (ack' and (ack==false))
                           then false
                           else true,bads',{})))))
                    else BOT) -- new processor faulty is not allowed
                    [] tock -> (if member(dying,S)
                                then BOT
                             else FAULTY(oldbads,union(bads,{whose}),
                                union(newbads',{dying}),
false,whose,true,union(bads',{dying}),{})
       ))))  -- dying node outside of S assume selfdiagnosis, and

         []
         faulty?m:S -> BOT


FAULTY'''(oldbads,newbads,newbads',whose,ack,dying,ack',bads') =
    -- bad nodes slot
          faulty?x:S ->BOT
        [] (dying!=whose) & tock ->
                 (if member(whose,bads')
                  then FAULTY(oldbads,newbads,newbads',
                             ack,dying,ack',bads',{})
                  else    (if (ack and (ack'==false))
                           then (if not member(dying,bads')
                                 then selfdiag!dying ->
FAULTY(oldbads,newbads,union(newbads',{dying}),
ack,dying,true,union(bads',{dying}),{})
   else  FAULTY(oldbads,newbads,union(newbads',{dying}),
ack,dying,true,union(bads',{dying}),{}))
```

```
                 else    (if (ack' and (ack==false))
                          then
           FAULTY(oldbads,newbads,union(newbads',{whose}),
ack,dying,false,union(bads',{whose}),{})
                           else
         FAULTY(oldbads,newbads,newbads',ack,dying,
if (ack' and (ack==false))
                then false
                else true,bads',{}))))
        [] (dying!=whose) and not member(whose,S) &
                     broadcast!whose?_ ->
                    tock ->        (if member(whose,bads')
   then FAULTY(oldbads,newbads,newbads',ack,dying,ack',bads',{})
             else    (if (ack and (ack'==false))
        then (if not member(dying,bads') then selfdiag!dying ->
          FAULTY(oldbads,newbads,union(newbads',{dying}),
ack,dying,true,union(bads',{dying}),{})
  else FAULTY(oldbads,newbads,union(newbads',{dying}),
ack,dying,true,union(bads',{dying}),{}))
            else    (if (ack' and (ack==false))
                           then
          FAULTY(oldbads,newbads,union(newbads',{whose}),
ack,dying,false,union(bads',{whose}),{})
               else
           FAULTY(oldbads,newbads,newbads',ack,dying,
if (ack' and (ack==false))
                              then false
                              else true,bads',{}))))
        [] (dying==whose) &
             ((if member(whose,S)
               then (broadcast!whose!ack' -> tock ->
                     FAULTY(oldbads,newbads,newbads',
ack,dying,true,bads',{})
|~| tock ->
FAULTY(oldbads,newbads,newbads',
ack,dying,true,bads',{}))
                  else broadcast!whose?a -> tock ->
              FAULTY(oldbads,newbads,newbads',
ack,dying,true,bads',{}))
                -- dying hasn't self diagnosed
            [] (not(member(whose,S))) & tock ->
           FAULTY(oldbads,newbads,newbads',
ack,dying,true,bads',{}))
within NOTFAULTY(oldbads,{},true,{})

-- the inductive step checks:

assert ABSTRACTION({0,1,2},{}) [F= (ABSTRACTION({0,1},{})
                        [| {| agreedfault,broadcast,tock,whosego |} |]
```

```
                                        Processor(2,{}))

assert ABSTRACTION({0,1,2,3},{}) [F= (ABSTRACTION({0,1,2},{})
                          [| {| agreedfault,broadcast,tock,whosego |} |]
                                        Processor(3,{}))

assert ABSTRACTION({0,1},{}) [F= (ABSTRACTION({0},{})
                          [| {| agreedfault,broadcast,tock,whosego |} |]
                                        Processor(1,{}))

-- check the base case.

assert ABSTRACTION({0},{}) [F= Processor(0,{})
```

Appendix F

CSP$_M$ for Arbitrary Branching Trees

```
{-
This script is a modification of that which corresponds to CR98.

Nodes are now data independent in the type which is actually the
identities of the downstream channels. A node process is constructed
of a local resource process and the appropriate down processes. All of which
are data independent in the type (DWNS).
-}


MAX_R = 2
RESOURCE = {1..2}

datatype DWNS = A | B | C | D | E

datatype RESULT = accept | reject
datatype MESSAGE = request . RESOURCE
                 | reply . RESULT . RESOURCE

channel localreq: DWNS . MESSAGE
channel upstream, downstream, localrep: MESSAGE
channel downstreamA, downstreamB: MESSAGE


min(x,y) = if x > y then y else x
max(x,y) = if x > y then x else y

transparent normal, sbisim, diamond
```

```
-- ----------------------------------------------------------
-- Model of the local resource for a node:
-- Note: it gives priority to incoming replies to mimise traffic
-- upstream.
--

S(T,q) =
let     S'(T,q,{}) = localreq ?x:T ! request ? v -> S'(T,q,{v})
        S'(T,q,seen) = (localreq ?x:T ! request ? vv ->
S'(T,q,union(seen,{vv}))
[]
(|~| v: seen @ if v <= q
  then upstream . request . v ->
                                S'(T,q,seen)
          else localrep . reply ! reject . v ->
S'(T,q,diff(seen,{v})))
                        []
                        upstream . reply ? a ? r ->
                        localrep . reply ! a ! r -> S'(T,q,diff(seen,{r})))

within S'(T,q,{})

LOCALRESOURCE(DS) = |~| q:RESOURCE @ S(DS,q)


-- ------------------------------------------------------------
-- Model of a downstream channel:



Down(id,seen,req,maxacc,minrej) =
downstream.request?v ->
(if (not(member(v,seen)) and not(member(v,req)))
then Down(id,union({v},seen),req,maxacc,minrej)
else Down(id,seen,req,maxacc,minrej))
[]
card(seen)>0 &
(|~| x:seen @ (if x<=maxacc
then downstream.reply.accept!x ->
     Down(id,diff(seen,{x}),req,maxacc,minrej)
else (if x>=minrej
then downstream.reply.reject!x ->
     Down(id,diff(seen,{x}),req,maxacc,minrej)
else localreq.id.request!x ->
     Down(id,diff(seen,{x}),union({x},req),
                                    maxacc,minrej))))
```

```
[]
card(req)>0 &
(|~| x:req @ (x<=maxacc) & downstream.reply.accept!x ->
     Down(id,seen,diff(req,{x}),maxacc,minrej)
[]
(x>=minrej) & downstream.reply.reject!x ->
     Down(id,seen,diff(req,{x}),maxacc,minrej))
[]
localrep.reply?rr?vv ->
(if (rr==accept)
then Down(id,seen,req,max(vv,maxacc),minrej)
else Down(id,seen,req,maxacc,min(vv,minrej)))

DOWN(x) = Down(x,{},{},0,4)


-- --------------------------------------------------------------------
-- PART ONE: That a HIGHNODE is equivalent to a LOWNODE.
-- --------------------------------------------------------------------
-- A LOWNODE(S,ABS,ID) using interface parallel:
-- Note: S is the set of all downstream connections from the
-- localsource. ABS is the set of all downstreams already abstracted.
-- T is the predicate type(not shown).
--  ID is the channel we are concentrating on.
--
-- The HIGHNODE abstracts away the DOWN(B) channel higher up.


HIGHNODE =
(LAbs(LAbs(LOCALRESOURCE({A,B}))({|localreq.A|}))({|localreq.B|}))



LOWNODE =
(LAbs((LAbs(LOCALRESOURCE({A,B}))({|localreq.A|}))
     [| {| localreq.B, localrep |} |]
      DOWN(B))({|downstream|}))

LAbs(P)(X) = (P [|X|] normal(CHAOS(X)))\X


----------------------------------------------------------------------
-- The checks:

assert LOWNODE [FD= HIGHNODE

assert HIGHNODE [FD= LOWNODE
```

```
-- -----------------------------------------------------------------
-- PART TWO: The structural induction using HIGHNODE with a DOWN on A.
----------------------------------------------------------------------
-- This is a model of a source:

source(q) =
let   source'(q,{}) = downstream . request ? v -> source'(q,{v})
      source'(q,seen) = (downstream . request ? vv ->
                              source'(q,union(seen,{vv}))
                     []
                     (|~| v: seen @ if v <= q
    then downstream . reply ! accept . v ->
                                   source'(q,diff(seen,{v}))
    else downstream . reply ! reject . v ->
source'(q,diff(seen,{v}))))

within source'(q,{})



SPEC = sbisim(diamond(|~| q:RESOURCE @ normal(source(q))))


-- -------------------------------------------------------------
-- The induction: that the interface presented by a node on any
-- downstream channel has the behaviour of a source, no matter
-- how many nodes stand between it and the source. A source will
-- always give a response for requests.

-- The base case is trivial:
--
-- SPEC [FD= SPEC

-- The inductive step:

assert SPEC [F= SPEC [downstream <-> upstream] NODE


-- Note: We only need to use the failures model here since we know
-- that the process can quite rightly diverge on the lazily abstracted
-- channels of the HIGHNODE.


NODE = ((LAbs(LOCALRESOURCE({A,B}))({|localreq.A|}))
     [| {| localreq.B, localrep |} |]
      DOWN(B)) \ {| localreq, localrep |}

-----------------------------------------------------------------
```

Appendix G

CSP$_M$ for a ring topology

```
-- Small example for a ring.

{- Nodes are in a ring. Communicate with neighbours and
outside world. A node gets a message from the outside world
and broadcasts it to everyone in the ring. It waits for the message to
return to it, then it knows that everyone has seen it, and disguards it.

To ensure that the nodes don't deadlock (caused by everyone inputing
a message and wannting to pass it on at once), we put a buffer of size
one on the output channel of a node.

Nodes can only deal with one message from the outside world at a time,
obviously using buffering could increase this capacity, (if multiple
messages are required than the buffering between nodes would have to
be increased accordingly, and some tagging system be introduced to ensure
that messages are unique.
-}
transparent normal

size = 8
tags = {0..(size-1)}
mess = {0,1}

-- currently their are 4 nodes in the ring


channel in: tags.mess
channel out: tags.tags.mess
channel c, d: tags.tags.mess

-- the channels in and out are used for communicating outside
```

```
-- of the ring. The label of the point at which the message
-- enters the ring, and the message, are communicated on the in
-- channel. When a message is output the point at which it is output
-- followed by the point of input and message are passed along the
-- channel.

-- the c channel is used for passing messages along the ring,
-- d is identical, but represents the buffering on the output
-- c channel.

N(x) = in.x?m -> d.x.x!m -> E(x)
       [] c.x?y?m -> ((x!=y) & out.x!y.m ->
      d.x!y.m -> N(x))

E(x) = c.x?y.m -> (if (x==y)
   then N(x)
   else out.x!y.m ->
           d.x!y.m -> E(x))

Buff(x) = d.x?y.m -> c.succ(x)!y.m -> Buff(x)

Node(x,bf,al) = normal(N(x) [| {| d.x |} |] Buff(x)) \ {| d.x |}


alph(x) = {| in.x, out.x, c.x, c.succ(x) |}

-- succ(x) is the function which works out the label
-- of the next node in the ring (who you should output to).

succ(y) = (y+1)%(size)

Ring = || x:tags @ [alph(x)] Node'(x)
Node'(x) = normal(N(x) [| {| d.x |} |] Buff(x)) \ {| d.x |}


--------------------------------------------------------------------
--------------------------------------------------------------------

-- Abstraction for a partialring:

SS(S,p,q,al,f,l,bf) =
  (c.p.p?m -> CHAOS(ASS(p,q,al,f,l,bf)))
  []
  (in.p?z -> if member(q,S)
     then SS'(S,p,q,al,z,f,l,bf)
     else SS''(S,p,q,al,z,f,l,bf))

SS'(S,p,q,al,y,f,l,bf) =
  (c.p.p?z -> CHAOS(ASS(p,q,al,f,l,bf)))
```

```
  []
  (out.q.p.y -> SS''(S,p,q,al,y,f,l,bf))

SS''(S,p,q,al,y,f,l,bf) =
  (c.p.p?z -> CHAOS(ASS(p,q,al,f,l,bf)))
  []
  (c.al.p.y -> SS'''(S,p,q,al,f,l,bf))

SS'''(S,p,q,al,f,l,bf) =
  c.p.p?z -> SS(S,p,q,al,f,l,bf)


ASS(p,q,al,f,l,bf) = {| in.p, out.q.p, c.al.p, c.p.p |}

A(al, f) = {| in, out, c.al, c.f |}

Abstraction(S,p,q,al,f,l,bf) =
SS(S,p,q,al,f,l,bf)
[| ASS(p,q,al,f,l,bf) |]
CHAOS(A(al,f))

-----------------------------------------------------------------
-- basecase: distinguish three nodes in superstate:
--           node in the network, bf and al
--           Node must be in S.

-- Threshold is 7 on the basecase and 8 on the step:

-- assert Abstraction({0},0,2,1,0,2,3) [T= Node(0,3,1)

-- step case

-- with S' size 2:
-- q outside S'
{-
assert Abstraction({0,1},0,4,2,0,1,5) [T=
(Abstraction({0},0,4,1,0,0,5)
[| {| c.1 |} |]
Node(1,0,2)) \ {| c.1 |}
-}
-- q inside S'
{-
assert Abstraction({0,1},0,1,2,0,1,5) [T=
(Abstraction({0},0,1,1,0,0,5)
[| {| c.1 |} |]
Node(1,0,2)) \ {| c.1 |}
-}
-- -----------------------------------
-- with S' size 3:
```

```
-- q outside S'
{-
assert Abstraction({0,1,2},0,4,3,0,2,5) [T=
(Abstraction({0,1},0,4,2,0,1,5)
[| {| c.2 |} |]
Node(2,1,3)) \ {| c.2 |}
-}
-- q inside S' and not Node
{-
assert Abstraction({0,1,2},0,1,3,0,2,5) [T=
(Abstraction({0,1},0,1,2,0,1,5)
[| {| c.2 |} |]
Node(2,1,3)) \ {| c.2 |}
-}
-- q inside S' and Node
{-
assert Abstraction({0,1,2},0,2,3,0,2,5) [T=
(Abstraction({0,1},0,2,2,0,1,5)
[| {| c.2 |} |]
Node(2,1,3)) \ {| c.2 |}
-}
-------------------------------------
{-
-- with S' size 4:
-- q outside S'
assert Abstraction({0,1,2,3},0,4,4,0,3,7) [T=
(Abstraction({0,1,2},0,4,3,0,2,7)
[| {| c.3 |} |]
Node(3,2,4)) \ {| c.3 |}

-- q inside S' and not Node
assert Abstraction({0,1,2,3},0,1,4,0,3,7) [T=
(Abstraction({0,1,2},0,1,3,0,2,7)
[| {| c.3 |} |]
Node(3,2,4)) \ {| c.3 |}

-- q inside S' and Node
assert Abstraction({0,1,2,3},0,3,4,0,3,7) [T=
(Abstraction({0,1,2},0,3,3,0,2,7)
[| {| c.3 |} |]
Node(3,2,4)) \ {| c.3 |}

-------------------------------------
-- with S' size 5
-- q outside S'
assert Abstraction({0,1,2,3,4},0,6,5,0,4,7) [T=
(Abstraction({0,1,2,3},0,6,4,0,3,7)
[| {| c.4 |} |]
Node(4,3,5)) \ {| c.4 |}
```

```
-- q inside S' and not Node
assert Abstraction({0,1,2,3,4},0,3,5,0,4,7) [T=
(Abstraction({0,1,2,3},0,3,4,0,3,7)
[| {| c.4 |} |]
Node(4,3,5)) \ {| c.4 |}

-- q inside S' and Node
assert Abstraction({0,1,2,3,4},0,4,5,0,4,7) [T=
(Abstraction({0,1,2,3},0,4,4,0,3,7)
[| {| c.4 |} |]
Node(4,3,5)) \ {| c.4 |}


------------------------------------
-- with S' size 6

-- q outside S'
assert Abstraction({0,1,2,3,4,5},0,6,6,0,5,7) [T=
(Abstraction({0,1,2,3,4},0,6,5,0,4,7)
[| {| c.5 |} |]
Node(5,4,6)) \ {| c.5 |}

-- q inside S' and not Node
assert Abstraction({0,1,2,3,4,5},0,3,6,0,5,7) [T=
(Abstraction({0,1,2,3,4},0,3,5,0,4,7)
[| {| c.5 |} |]
Node(5,4,6)) \ {| c.5 |}

-- q inside S' and Node
assert Abstraction({0,1,2,3,4,5},0,5,6,0,5,7) [T=
(Abstraction({0,1,2,3,4},0,5,5,0,4,7)
[| {| c.5 |} |]
Node(5,4,6)) \ {| c.5 |}

-}


{-
-------------------------------------
-- with S' size 7
-- adding on the second to last node
-- q outside S', q==aal==bf

assert Abstraction({0,1,2,3,4,5,6},0,7,7,0,6,7) [T=
(Abstraction({0,1,2,3,4,5},0,7,6,0,5,7)
[| {| c.6 |} |]
Node(6,5,7)) \ {| c.6 |}
```

```
-- q inside S' and not Node, aal==bf
assert Abstraction({0,1,2,3,4,5,6},0,4,7,0,6,7) [T=
(Abstraction({0,1,2,3,4,5},0,4,6,0,5,7)
[| {| c.6 |} |]
Node(6,5,7)) \ {| c.6 |}

-- q inside S' and Node
assert Abstraction({0,1,2,3,4,5,6},0,6,7,0,6,7) [T=
(Abstraction({0,1,2,3,4,5},0,6,6,0,5,7)
[| {| c.6 |} |]
Node(6,5,7)) \ {| c.6 |}
-}
--------------------------------------
-- with S' size 8
-- (adding on the last node closing the ring)
-- q must be inside S
-- aal==f, al==l, bf==al



-- q isn't Node

assert SPEC(0,5) [T=
(Abstraction({0,1,2,3,4,5,6},0,5,7,0,6,7)
[| {| c.7,c.0 |} |]
Node(7,6,0)) \ {| c.7,c.0 |}

-- q is Node
assert SPEC(0,7) [T=
(Abstraction({0,1,2,3,4,5,6},0,7,7,0,6,7)
[| {| c.7,c.0 |} |]
Node(7,6,0)) \ {| c.7,c.0 |}


--------------------------------------

Spec(p,q) = in.p?z -> out.q!p.z -> Spec(p,q)
SPEC(p,q) = Spec(p,q) [| {| in.p, out.q.p |} |] CHAOS({|in,out|})
```

Appendix H

CSP$_M$ for Torus

```
-- Working version:
-- Small example for a toroid

xsize = 6
ysize = 5

xtags = {0..(xsize-1)}
ytags = {0..(ysize-1)}

nametype send  = (xtags,ytags)
nametype receive = (xtags,ytags)

-- point it enters grid. destination:

channel in: send.receive

-- point it leaves grid. orignal sender:
channel out: receive.send

-- messages are input on the senders right channel,
-- labelled with destination, and the original point
-- of entry:

channel right,down: send.receive.send

getx((x,y)) = x
gety((x,y)) = y

-- a node:

N(n,al,d) =
```

```
  (in.n?dest -> N'(n,dest,n,al,d))
  []
  (right!n?dest?sender -> N'(n,dest,sender,al,d))
  []
  (down!n?dest?sender -> N'(n,dest,sender,al,d))

N'(n,dest,s,al,d) =
  (if getx(n)!=getx(dest)
   then right!al!dest!s -> N(n,al,d)
   else (if gety(n)!=gety(dest)
         then down!d!dest!s -> N(n,al,d)
         else out!n!s -> N(n,al,d)))

alph(n,r,d) = {| in.n, out.n, right.n,
               down.n, right.r, down.d |}



-------------------------------------------------------------------
-- isolate a case of sending a message between two nodes p and q.
-- superstate is a weak 1-place buffer between p and q: i.e. that
-- the torus behaves as a buffer between p and q, so long as
-- no more than one message is put in it, otherwise it breaks
-- and behaves chaotically.


-- when behaviour isn't pertinent to the buffer
-- we are investigating we IGNORE it, this is done
-- be recursing to the state we were in immediately before.

WBuff(S,(px,py),(qx,qy),al,bx) =
  (in.(px,py).(qx,qy) -> WBuff'(S,(px,py),(qx,qy),al,bx))
  [] -- message from void (disappears when row complete) IGNORE:
  (right!(px,py)?dest?sender -> WBuff(S,(px,py),(qx,qy),al,bx))
--  [] -- message from void (dissappears when torus complete) IGNORE:
--  (down!(px,py)?dest?sender -> WBuff(S,(px,py),(qx,qy),al,bx))
  [] -- falls off bottom (only happens when torus incomplete), IGNORE:
  ([](x,y):{(p,q)|p<-S,q<-ytags} @
       (down?(x,y)?dest?s -> WBuff(S,(px,py),(qx,qy),al,bx)))
  [] -- falls off rhs, only when row incomplete, IGNORE:
  (right!al?dest?s -> WBuff(S,(px,py),(qx,qy),al,bx))


WBuff'(S,(px,py),(qx,qy),al,bx) =
  ((member(qx,S) and py==qy) & (out.(qx,qy).(px,py) ->
          WBuff(S,(px,py),(qx,qy),al,bx)))
  [] -- falls off rhs, only when row incomplete:
  (right!al?dest?send -> (
      if (dest==(qx,qy) and send==(px,py))
```

```
      then WBuff(S,(px,py),(qx,qy),al,bx)
      else WBuff'(S,(px,py),(qx,qy),al,bx)))
--  [] -- passed down to row beneath:
--  ((member(qx,S) and py!=qy) &
--       (down!bx!(qx,qy)!(px,py) ->
--               WBuff(S,(px,py),(qx,qy),al,bx)))
  [] -- break the buffer:
  (in.(px,py).(qx,qy) -> CHAOS(alphbuff((px,py),(qx,qy),al,bx)))
  [] -- message from void (disappears when row complete) IGNORE:
  (right!(px,py)?dest?sender -> WBuff'(S,(px,py),(qx,qy),al,bx))
  []
  ([](x,y):{(p,q)|p<-S,q<-ytags} @ down?(x,y)?dest?send ->
      (if ((x,y)==bx and dest==(qx,qy) and send==(px,py))
       then WBuff(S,(px,py),(qx,qy),al,bx)
       else WBuff'(S,(px,py),(qx,qy),al,bx)))


alphbuff((px,py),(qx,qy),al,bx) =
     {| in,out,right.al,right.(px,py),down |}

--  {| in.(px,py).(qx,qy),
--     out.(qx,qy).(px,py),
--     right.al, right.(px,py),
--     down.bx.(qx,qy).(px,py) |}

Superstate(S,p,q,al,bx) =
   WBuff(S,p,q,al,bx)
   [| {| in.p.q, out.q.p |} |]
   CHAOS(alphCOMMS(p,q,al,bx))

alphCOMMS(p,q,al,bx) = {| in, out |}


------------------------------------------------------------

-- FIRST INDUCTION

----------------------------------------------------------------
-- DI Induction to establish that row b is a buffer
-- between the input of a message and its output
-- or its forwarding to a different row on the correct
-- x co-ordinate.
------------------------------------------------------------------
-- BASECASE
-- Super(S,p,q,al,bx) [T= N(id,al,d)

-- threshold: on ytags= (py,qy,aly,bxy,idy,dy)+2
--                        but py=aly,py=idy,bxy=dy
--  = py,qy,dy + 2 = 5
```

```
--              xtags = (px,qx,alx,dx)+2 = 6
-- idx is a member of S. so preidcate has no effect on threshold.

-- three possible configurations:
--     q=bx, q=al, q!=bx and !=al

-- make these checks for all
-- sizes of types between 2 and threshold (inclusive).
-- NOTE: we don't check for size one since then we would
-- only require a ring topology.


------------------------------
-- x = 5, y = 6

-- 1. q=bx  Super(S,p,q,al,q) [T= N(id,al,bx)

assert Superstate({0},(0,0),(0,1),(1,0),(0,1))
             [T= N((0,0),(1,0),(0,1))

-- 2. q=al: Super(S,p,q,q,bx) [T= N(id,q,bx)
assert Superstate({0},(0,0),(1,0),(1,0),(0,1))
             [T= N((0,0),(1,0),(0,1))

-- 3. q!=al and q!=bx Super(S,p,q,al,bx) [T= N(id,al,bx)
assert Superstate({0},(0,0),(2,1),(1,0),(0,1))
             [T= N((0,0),(1,0),(0,1))

----------------------------
-- x = 5, y = 5

-- 1. q=bx  Super(S,p,q,al,q) [T= N(id,al,bx)

--assert Superstate({0},(0,0),(0,1),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

-- 2. q=al: Super(S,p,q,q,bx) [T= N(id,q,bx)
--assert Superstate({0},(0,0),(1,0),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

-- 3. q!=al and q!=bx Super(S,p,q,al,bx) [T= N(id,al,bx)
--assert Superstate({0},(0,0),(2,1),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

----------------------------
-- x = 5, y = 4

-- 1. q=bx  Super(S,p,q,al,q) [T= N(id,al,bx)
```

```
--assert Superstate({0},(0,0),(0,1),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

-- 2. q=al: Super(S,p,q,q,bx) [T= N(id,q,bx)
--assert Superstate({0},(0,0),(1,0),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

-- 3. q!=al and q!=bx Super(S,p,q,al,bx) [T= N(id,al,bx)
--assert Superstate({0},(0,0),(2,1),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

----------------------------
-- x = 5, y = 3

-- 1. q=bx  Super(S,p,q,al,q) [T= N(id,al,bx)

--assert Superstate({0},(0,0),(0,1),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

-- 2. q=al: Super(S,p,q,q,bx) [T= N(id,q,bx)
--assert Superstate({0},(0,0),(1,0),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

-- 3. q!=al and q!=bx Super(S,p,q,al,bx) [T= N(id,al,bx)
--assert Superstate({0},(0,0),(2,1),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

---------------------------
-- x = 5, y = 2

-- 1. q=bx  Super(S,p,q,al,q) [T= N(id,al,bx)

--assert Superstate({0},(0,0),(0,1),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

-- 2. q=al: Super(S,p,q,q,bx) [T= N(id,q,bx)
---assert Superstate({0},(0,0),(1,0),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))

-- 3. q!=al and q!=bx Super(S,p,q,al,bx) [T= N(id,al,bx)
--assert Superstate({0},(0,0),(1,1),(1,0),(0,1))
--             [T= N((0,0),(1,0),(0,1))




-------------------------------------------
```

```
-- STEPCASE -  building ring row py:
-- threshold: y = 5
--             x = 5, but q may or may not be in S,

-- note: the channels a node and superstate are connected on
-- are hidden:

assert Superstate({0,1},(0,0),(3,0),(2,0),(3,1)) [T=
            Superstate({0},(0,0),(3,0),(1,0),(3,1))
            [ right.(1,0) <-> right.(1,0) ]
            N((1,0),(2,0),(1,1))


------------------------------------------
-- 2ND INDUCTION... completing column 3,
-- lazy abstracting all messages from other columns..
-------------------------------------------
{-
-- stepcase adding on nodes in the px column:

-- lazy abstraction of messages on new nodes right channels:

lazyNODE((id,al,d)) =
  (N(id,al,d)
  [| {| right.id, right.al |} |]
  CHAOS({| right.id, right.al |})) \ {| right.id, right.al |}

-- now S represents the rows present

assert Superstate({0,1},(0,0),(3,3),(0,0),(3,2)) [T=
            (Superstate({0},(0,0),(3,3),(0,0),(3,1))
            [ down.(3,1) <-> down.(3,1) ]
            lazyNODE((3,1),(4,1),(3,2)))

-}
```

Appendix I

Statistics

The table below contains statistics for each example script. The checks were performed on a Pentium III processor, running at 1500MHz with 2Gb of RAM. For each example one base case and one step case was considered, and in all cases the largest required size of data type was considered. (The size of the data type of node names is not relevant to the monument example from Section 2 and the arbitrary branching trees (RSVP) example from Section 5, since both were structural inductions. In the case of RSVP the base case was trivially true, no check was performed.

Chapter I **Statistics**

Statistics for checks				
Example	**Base/Step**	**Size of Type**	**States**	**Time**
Monument (2)	Base		2	<1sec
	CD Step		21	<1sec
Election (3.2.1)	Base	11	37	1sec
	Step	11	958	3sec
Multiplex (3.2.2)	Base	7	44,697	5sec
	Step	7	69,507	6sec
TTP (4)	Base	4	765	3sec
	Step	4	13,803	5sec
RSVP (5)	Base			
	Step		7770	1sec
Ring (6)	Base	7	1521	1sec
	Step	8	8402	7sec
Torus (7)	Base	x=5,y=6	258	2mins
	Step	x=5,y=5	25,607	5mins

Bibliography

[1] P.A. Abdulla and B. Jonsson, Verifying programs with unreliable channels, *Proceedings of the 8th IEEE International Symposium on Logic in Computer Science*, 1993.

[2] P.A. Abdulla and M. Kindahl, Decidability of simulation and bisimulation between lossy channel systems and finite state systems, *Proceedings of CONCUR'95 The 6th International Conference on Concurrency Theory*, LNCS 962, 333-347, Springer Verlag, 1995.

[3] R. Alur, C. Courcoubetis, and D. Dill, Model Checking for real-time systems, *Proceedings of the 5th IEEE International Symposium on Logic in Computer Science*, Philadelphia, 1990.

[4] R. Alur and T. Henzinger, A really temporal logic, *Proceedings of the 30th Annual Symposium Foundations of Computer Science*, 1989.

[5] S. Berezin, S. Campos and E. Clarke, Compositional Reasoning in Model Checking, *Proceedings of COMPOS'97.*

[6] Kenneth E. Batcher, Design of a massively parallel processor, *IEEE Transactions on Computers*, 29(9):836-840, September 1980.

[7] 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.

[8] Alan Bundy, *The Computer Modelling of Mathematical Reasoning*, Academic Press Inc, 1985.

[9] Julain Bradfield and Colin Stirling, Local Model Checking for Infinite State Spaces, *Theoretical Computer Science*, **96**, (1992), 157-174.

[10] Ramesh Bharadwaj and Steve Sims, Salsa: Combining Decision Procedures for Fully-Automatic Verification.

[11] R. Bharadwaj, A. Felty and F. Stomp, Formalizing INductive Proofs of Network Algorithms, *Proceedings of the 1st Asian Computer Science Conference*, Thailand, December 1995.

[12] K. Cerans, Decidability of bisimulation equivalence for parallel timer processes, *Proceedings of the Workshop on Computer Aided Verification*, LNCS 663, 302-315, 1992.

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

[14] E.M. Clarke, O. Grumberg and D. Long, Model checking and abstraction, *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, 1992.

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

[16] E.M. Clarke, O. Grumberg and D.A. Peled, *Model Checking*, The MIT Press, 1999.

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

[18] S.J. Creese and Joy Reed, Verifying End-to-End Protocols Using Induction with CSP/FDR, *Proceedings of the Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'99)*, April 1999, San Juan, Puerto Rico. Published by Springer as volume 1586 of the Lecture notes in Computer Science series.

[19] S.J. Creese and A.W. Roscoe, Verifying an infinite family of inductions simultaneously using data independence and FDR, In *Formal Methods for Protocol Engineering and Distributed Systems, the proceedings of Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV'99)*, October 5-8 1999, Beijing, China. Published by Kluwer Academic Publishers.

[20] S.J. Creese and A.W. Roscoe, Formal Verification of Arbitrary Network Topologies, em Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 28 - July 1 1999, Las Vegas, Nevada, USA. Volume II. Published by CSREA Press.

[21] S.J. Creese and A.W. Roscoe, *TTP: A case study in combining induction and data independence*, Oxford University Computing Laboratory Technical Report, PRG-TR-1-99.

[22] S.J. Creese and A.W. Roscoe, Data Independent Induction over structured networks, *Proceedings of the International Conference on Parallel and Distributed*

Processing Techniques and Applications (PDPTA '00), June 25 - 30 2000, Las Vegas, Nevada, USA.

[23] The Coq proof assistant, http://coq.inria.fr/.

[24] Formal Verification of a Partial-Order Reduction Technique for Model Checking, Ching-Tsun Chou and Doron Peled, *Proceedings of TACAS'96*, Passau, Germany, March 1996.

[25] E. M. Clarke and J. M. Wing, Formal Methods: State of the art and future directions, *ACM Computing Surveys*, 28(4):626-643, December 1996.

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

[27] D. Dams, O. Grumberg, and R. Gerth, Abstract interpretation of reactive systems *Proceedings of IFIP working conference on Programming Concepts, Methods and Calculi (PROCOMET'94)*, 1994.

[28] 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.

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

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

[31] *First order logic and Automated theorem proving*, 2nd edition, Springer-Verlag, 1996.

[32] Yuri Gurevich and Raghu Mani, *Group Membership Protocol: Specification and Verification, Specification and Validation Methods for Programming Languages and Systems*, Oxford University Press, 1994.

[33] L. Gravano, G. Pifarre, P. Berman, and J. Sanz, *Adaptive Deadlock- and Livelock-Free Routing with All Minimal Paths in Torus Networks*, IEEE Transactions on Parallel and Distributed Systems, vol 5, no. 12, Dec. 1994.

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

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

[36] R. Hojati and R. K. Brayton, Automatic Datapath Abstraction In Hardware Systems, *CAV'95: 7th International Conference on Computer Aided Verification*, LNCS 939, 98-113, Springer Verlag, 1995.

[37] The HOL Theorem Proving System, http://lal.cs.byu.edu/lal/hol-documentation.html.

[38] Gerard J. Holzmann, Spin Model Checking, Dr. Dobb's Journal, October 1997.

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

[40] G.J. Holzmann and D. Peled, The state of spin, *CAV'96: 8th International Conference on Computer Aided Verification*, LNCS 1102, 385-389, Springer, 1996.

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

[42] P. Jancar and F. Moller, Checking regular properties of petri nets, *Proceedings of CONCUR'95 the 6th International Conference on Concurrency Theory*, LNCS 962, 348-362, Springer Verlag, 1995.

[43] B. Jonsson and J. Parrow, Deciding bisimulation equivalences for a class of non-finite-state programs, *Information and Computation*, 107(2), 272-302, December 1993.

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

[45] Updated version of [44] to be found at www.csl.sri.com/wdag97.html.

[46] H. Kopetz and G. Grünsteidl, TTP –a protocol for fault-tolerant real-time systems, *IEEE Computer*, 27(1):14-23, January 1994.

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

[48] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, E. Shanker, Symbolic Model Checking with Rich Assertional Languages, October 1998.

[49] Y. Kesten and A. Pnueli, Modularization and Abstraction: The Key to Practical Formal Verification, *Proceedings of the 23rd Int. Symp. on Mathematical Foundations of Computer Science MFCS-98*, LNCS 1450, Springer Verlag, 1998.

[50] M. Leuschel and T. Massart, Inifinite State Model Checking by Abstract Interpretation and Program Specialisation.

[51] 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.

[52] R.S. Lazič, T. C. Newcomb and A. W. Roscoe, *On model checking data-independent systems with arrays without reset (abstract)*, Proceedings of the 2nd International Workshop on Verification and Computational Logic (VCL 2001), Technical Report, Department of Electronics and Computer Science, University Southampton, 2001.

[53] R.S. Lazić and A.W. Roscoe, Data Independence with Generalised Predicate Symbols, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, June 28 - July 1 1999, Las Vegas, Nevada, USA. Volume I, 319-325. Published by CSREA Press.

[54] 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.

[55] C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani and S. Bensalem, Property preserving abstractions for the verification of concurrent systems, *Formal Methods in System Design*, (6):11-44, 1995.

[56] K.L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic, 1993.

[57] R. Milner, *A calculus of communicating systems*, Springer, LNCS 92, 1980.

[58] R. Milner, *Communication and concurrency*, Prentice Hall, 1989.

[59] Carroll Morgan, *Programming from Specifications*, Prentice Hall, 1990.

[60] Michael Noakes and William J. Dally, System design of the J-Machine, *Proc. Sixth MIT Conf. on Advanced Research in VSLI*, 179-194, 1990.

[61] Tino Pyssysalo, An Induction Theorem for Ring Protocols of Processes Described with Predicate/Transition Nets, *Research Report No. 37*, Digital Systems Laboratory, Helsinki University of Technology, April 1996.

[62] PRobabilistIc Symbolic Model checker, http://www.cs.bham.ac.uk/∼dxp/prism/.

[63] The PVS Specification and Verification System, http://pvs.csl.sri.com/whatispvs.html.

[64] Yves-Marie Quemener, Thierry Jeron, Finitely Representing Infinite Reachability Graphs of CFSMS with Graph Grammars, *Publication Interne No. 994*, IRISA, France.

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

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

[67] 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.

[68] S. Owre, S. Rajan, J. M. Rushby, N. Shanker, and M. Srivas, PVS: Combining Specification, Proof Checking, and Model Checking, *Proceedings of CAV'96.*

[69] S. Rajan, N. Shanker, and M. Srivas, An Integration of Model Checking with AUtomated Proof Checking, *Proceedings of CAV'95.*

[70] John Rushby, Mechanized Formal Methods: Where Next?, *FM'99: The World Congress on Formal Methods*, LNCS 1708, pp. 48-51, September 1999.

[71] John Rushby, Integrated Formal Verification: Using Model Checking With Automated Abstraction, Invarient Generation, and Theorem Proving, *Invited paper presented at 5th SPIN workshop*, Trento, Italy, 5th July 1999.

[72] John Rushby, Theorem Proving for Verification, *Proceedings of Modelling and Verification of Parallel Processes: MoVEP2k*, Nantes, France, June 2000.

[73] John Rushby, Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification, *Proceedings of CAV'00*, 2000.

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

[75] Natarajan Shanker, Unifying Verification Paradigms (Extended Abstract), *Proceedings of FTRTFT'96.*

[76] Hassen Saidi and Natarajan Shanker, Abstract and Model Check while you Prove, *Proceedings of CAV99*, Trento, Italy, July 7-10, 1999.

[77] N. Shanker, Machine-Assisted Verification Using Theorem Proving and Model Checking, To appear in *Mathematical Programming Methodology*, ed. M. Broy.

[78] N. Shanker, Combining Theorem Proving and Model Checking through Symbolic Analysis, Invited paper at *CONCUR2000.*

[79] Symp: Symbolic Model Prover, http://www.cs.cmu.edu/ modelcheck/symp.html.

[80] Tom Newcomb, Transfer of Status Dissertation, Oxford University Computing Laboratory, 2000.

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

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

[83] 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.

[84] 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.