Optimising Communication Structure for Model Checking

Peter Saffrey[1] and Muffy Calder[2]

[1] Dept. of Computer Science, University College London
`P.Saffrey@ucl.ac.uk`
[2] Dept. of Computing Science, University of Glasgow
`muffy@dcs.gla.ac.uk`

Abstract. Model checking is an effective tool in the verification of concurrent systems but can require skillful use. The choice of representation for a particular system can make a substantial difference to whether the verification will prove tractable. We present a method for improving the choice of representation by effective use of *communication structure*. The main contribution is a technique for selecting a communication structure which yields a reduced search space whilst preserving the essential behaviour of a representation. We illustrate our method with examples based on the model-checker Spin.

1 Introduction

Concurrent systems consisting of a number of communicating processes are present in many real world applications. However, the complexity inherent in communication and parallelism makes it difficult to build concurrent systems that behave as intended without errors or failures.

One technique to aid in the construction of reliable concurrent systems is *model checking* [2]. Model checking attempts to *verify* the behaviour of a system by exploring all possible behaviours of that system, the *state space*, by checking each behaviour against a set of *properties* which are expected to hold, or be violated. This procedure can be expensive and for some systems the state space may be too large for a complete search: the verification is thus intractable.

When verifying a real world system using model checking, a particular *representation* (i.e. a model) of that system must be chosen. It is usually possible to choose a variety of representations, any one of which would accurately represent the behaviour of that system.

An illustration of this phenomenon is shown in figure 1. Each representation consists of a model and an associated set of specified properties. Each representation results in a state space, often of differing size.

Choosing, or developing a representation for a system is analogous to the act of converting a specification into a piece of software, often called *programming*. In most cases, a large number of programs can be written to conform to a single specification, and they may vary widely in time efficiency, space efficiency or other measure of quality.

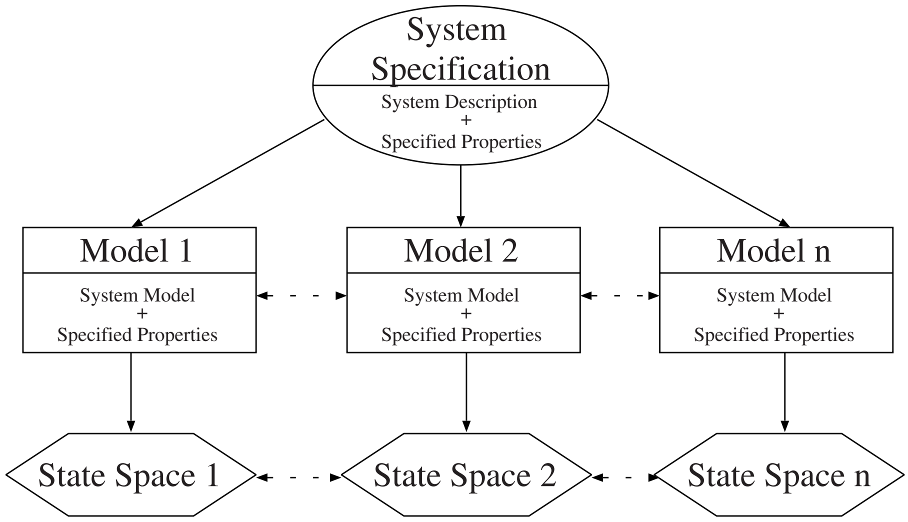

System Specification
System Description + Specified Properties
Model 1
System Model + Specified Properties
Model 2
System Model + Specified Properties
Model n
System Model + Specified Properties
State Space 1
State Space 2
State Space n

Fig. 1. Alternative models and their state spaces.

Although these observations are relevant to all aspects of representing a system for model checking, we will apply these notions to one specific area: that of *communication structure*. In a concurrent system, the communication structure is the method by which information passes between components. More precisely, it refers to the data structures used for communication, and which processes may access those data structures (to read or write). Specification languages for model-checkers provide a number of constructs to represent communication structure. How these constructs are used can have a considerable influence over the size of the resulting state space.

Communication structure is most pertinent to asynchronous communication since in synchronous systems, messages are transmitted instantaneously and thus the choice of structure has little impact on the state-space. We therefore concentrate on asynchronous communication and the use of *channels*. The focus of this work is not generic, internal data structures for model checking, but rather the specific, source language data structures used in problem representation. The inspiration to explore the relationship between communication structure representation and state space was provided by [6] in which the author demonstrated a reduction in state space for an example based on the logical linked protocol.

The main contribution of this paper is a technique for selecting a communication structure which preserves the essential behaviour of a representation and yields a tractable search space. We assume the starting point is an intractable initial representation. Therefore, our technique must be applicable *without* model checking the initial representation.

The remainder of the paper is organised as follows. We open with a motivating example before justifying the model checker we have chosen to illustrate our technique (sections 2 and 3). Sections 4, 6, 7, 8 and 9 outline our technique for demonstrating property preservation between communication structures and how good communication structures can be chosen. Section 10 provides a case study. Finally, sections 11, 12 and 13 provide related work, further work and conclusions.

2 Communication Structure

To illustrate how communication structure can influence the size of a resulting state space, we present a simple example.

Consider the system shown in figure 2 with its two alternative communication structures. Two users may send messages to one another along communication channels. In the first case, two *dedicated* channels are used, a channel for each direction. In the second case, communication in either direction is mediated by a single *shared* channel.

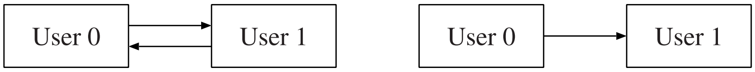


Fig. 2. A simple system with alternative communication structures.

Assume that each user can only send one type of message and that each channel can contain a maximum of one message. Table 1 enumerates the message combinations for the shared and dedicated configurations: [] denotes all channels empty, (user0,user1) a message from user0 to user1 and (user1,user0) a message from user1 to user0. Note that a state where both messages are being passed simultaneously is only possible for the dedicated structure; its state space size is greater by 1.

Table 1. Enumeration of combinations for simple system.

Dedicated	Shared
[]	[]
(user0,user1)	(user0,user1)
(user1,user0)	(user1,user0)
(user0,user1), (user1,user0)	—

The simple system can be extended to three users, where the dedicated structure has 6 channels (2 between each user) and the shared structure 3 channels (1 between each user). Here the dedicated structure has 64 possible states while the shared structure only 27. As the topology becomes more complex, the difference in the number of combinations which can achieved with various communication structures tends to increase.

It is clear that a change from a dedicated to a shared structure alters the size of the state space in this simple example. However, these systems are not isomorphic: one state is only reachable with the dedicated structure. There are also differences in the possible series of events that that can occur. For example, in the dedicated system, it is possible for a send event from user0 to immediately follow a send event from user1; in the shared configuration, there must be a receive event to clear the channel.

This example demonstrates how a change to communication structure can affect state space size but with a loss of certain behaviour. Whether such differences in behaviour influence the verification of specified properties for this system is a primary question addressed by our work. A further question is which representation will result in the smallest state space. We will consider some guidelines for selecting a representation later, in section 9. For now we concentrate on the more problematic issue of altering a representation, i.e. traversing the horizontal arrows in figure 1. For alteration to be applicable, we must provide some demonstration of equivalence between the initial and the altered system.

3 Spin and Promela

Our method is designed to apply to any systems that are modelled with asynchronous communication. To validate the techniques we have constructed an implementation based on the model checker Spin [5] and its accompanying specification language Promela. Spin applies state of the art, on the fly, model checking techniques. Promela is a succinct and easy to use language that supports the use of channels for communication. All the examples presented in this paper were represented in Promela and model checked using Spin.

4 Communication Structure Alteration

In this section, we address communication structure alteration: altering the communication structure of a representation to result in a smaller state space.

Communication structure alteration takes as input an *initial representation* with an initial communication structure. The aim is to produce an *alternative representation* which results in a smaller state space and which preserves the *specified* properties of the initial representation. It is important to note that only the specified properties must be preserved, other properties might change.

The procedure follows the following stages:

1. The initial representation is analysed including the extraction of the communication structure.
2. Using a best communication structure framework (see section 9), a new communication structure is selected that results in a smaller state space. The initial representation with this new communication structure is referred to as the *candidate representation.*
3. The initial and candidate representations are compared to test for property preservation. Details of this procedure are in section 6. After comparison,
   – If the candidate representation preserves the properties of the initial representation, the candidate representation, with its smaller resulting state space, can be used to verify specified properties of that system.
   – If alteration is not property preserving, return to stage 2 and choose another communication structure.

A communication structure alteration only alters the communication structure: no other part of the system – for example, the number of components, the behaviour of those components or which components interact with each other – is altered.

Communication structure alteration should only be applied when the specified properties can be preserved by the alteration. Our method for testing whether or not properties are preserved is described in section 6. Recall that in most cases, the initial representation is intractable.

To illustrate the basic principles behind our technique, we apply them to a small worked example. The example is described in the next section.

5 The Simple System in Promela

The example is taken from figure 2, with two processes User 0 and User 1, instantiations of a generic process given in Promela by the following:

```
mtype = { send, receive, u0, u1 };
mtype lastaction[2];
proctype user(chan inchan, outchan;mtype myid, oppid)
{
        do
        ::outchan!(myid,oppid) ->
                lastaction[myid] = send
        ::inchan?eval(oppid),eval(myid) ->
                lastaction[myid] = receive
        od
}
```

Promela has a C-like syntax. Briefly, `mtype` is a Promela keyword for an enumerated type; so `send`, `receive`, etc. are constants. `c!m` denotes write `m` on channel `c`, `c?m` denotes read `m` from channel `c` (destructive read). Statements (e.g. assignments denoted by "=") can be guarded by other statements, with the form statement `->` statement. The second statement executes only if the guard is not blocked. The eval functions ensure that incoming messages must match these variable values rather than overwriting them.

All channels are parameters: this allows us to instantiate the process with either communications structures from figure 2 without having to alter the body of the process. Promela representations to be used with our method must be encoded in this way to allow the communication structure to be altered without other changes to the representation.

The whole system consists of two instantiations of the process `user`, thus assuming that `chans` is an array of channels, the initial representation would use a separate channel for each parameter as shown below:

```
        run user(chans[0], chans[1], u0, u1);
        run user(chans[1], chans[0], u1, u0)
```

and the candidate representation would use the same channel thus:

```
        run user(chans[0], chans[0], u0, u1);
        run user(chans[0], chans[0], u1, u0).
```

Note also that the send and receive statements in the process definition include variables denoting the intended recipient of the message. This is to prevent a user process receiving its own message in a shared channel configuration. This annotation of messages is necessary to ensure messages arrive as intended regardless of communication structure.

5.1 Specified Properties

In Spin, properties are expressed using linear temporal logic (LTL) [11]. The properties are as follows:

– *User0 will eventually receive a message.* This is expressed in LTL as $\Box\Diamond p$ where p is the boolean expression lastaction[0]==receive. We refer to this property as **user0 receive**.
– *There exists a reachable state where the last message action for both User0 and User1 are receive actions.* This is expressed in LTL as $\Diamond(p \wedge q)$ where p is the boolean expression lastaction[0]==receive and q is the boolean expression lastaction[1]==receive. We refer to this property as **both receive**.

We now return to the task of testing for property preservation.

6 Property Preservation Testing

The communication structure alteration procedure described in section 4 requires that we determine whether a specified property is preserved between two alternative communication structures. In this section we present an overview of our procedure for testing for property preservation which involves comparing traces in *message automata*.

6.1 Message Automata

To compare alternative representations we use *message automata*, an abstraction we have devised to reason about communication structure. The message automata for example system are shown in figure 3, the initial model on the left and the candidate model is on the right.

Message automata consist of *message states* linked by *message statements*. A message state is labelled with a name and the messages that are present on all channels, messages that have been sent but not yet received, at that state. We are only interested in whether messages have been sent or received: which channels are used for their transit is irrelevant. A message statement is a Promela statement that sends or receives a message, here prefixed by its (local) process name.

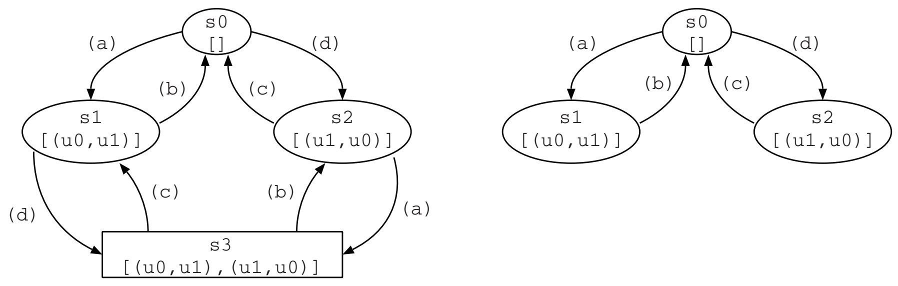
The two message automata in figure 3 differ by only a single state: the state s3. This non-shared state is known as a *difference state* (denoted by a rectangle) and is crucial to determining whether properties are preserved between the two communication structures.

6.2 Traces

The key idea behind testing for property preservation is comparing *traces* through the message automaton. In particular, we are concerned with *difference traces* and *emulating traces*. A difference trace is a trace which exists in the message automaton for one communication structure, but not for the other. In the simple example, the trace s0s1s3 is a difference trace, since it can only be achieved by the message automaton for the initial system. By definition, every difference trace contains at least one difference state. An emulating trace is a trace which *emulates* the behaviour of a difference trace with respect to a specified property. To emulate a difference trace, the emulating trace must match both the initial and final message states and must also match the *effect* of the trace on a specified property. We will describe the exact meaning of *effect* in section 6.4, in the next section we discuss how to reduce the number of traces under consideration.

6.3 Difference Sub-traces

To reduce the emulation effort, we will emulate only difference *sub*-traces, illustrated in figure 4. The figure shows two disjoint sets of states (think of them as rings, this is not a Venn diagram): an inner set containing the states shared by the two message automata and an outer set containing the difference states. In our example, the shared set would contain s0, s1 and s2 and the difference set s3.



```
(a) user0:chans[0]!u0,u1              (a) user0:chans[0]!u0,u1
(b) user1:chans[0]?eval(u0,u1)        (b) user1:chans[0]?eval(u0,u1)
(c) user0:chans[1]?eval(u1,u0)        (c) user0:chans[0]?eval(u1,u0)
(d) user1:chans[1]!u1,u0              (d) user1:chans[0]!u1,u0
```

Fig. 3. Message automata for the simple system. (left) Initial communication structure. (right) Candidate communication structure.

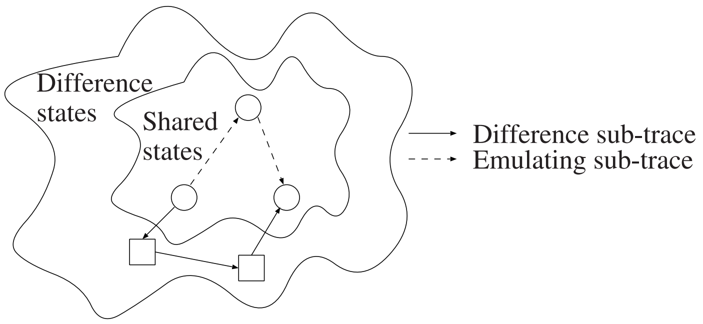


Fig. 4. Emulating sub-trace illustration.

When emulating a trace, any sub-trace which exists only within the shared set can be emulated by simply copying the appropriate transitions: the states are common to both automata. Only when a trace enters the difference set is more sophisticated emulation required. We can take advantage of this observation by only emulating the difference sub-traces, the sections of a trace which enter the difference set. Once the trace re-enters the shared set, direct emulation is possible: an emulating trace for this section already exists since the shared set of states and their transitions is identical in the two message automata.

From the simple example, consider the trace s0s1s3s2s0. The subtraces s0s1 and s2s0 use only shared states and can be emulated directly. We need only find an emulating sub-trace for the difference sub-trace s1s3s2. This emulating sub-trace must not only match the effect on a specified property, but also the start and end states of the difference sub-trace. This would allow the emulating sub-trace to form a direct substitution for the difference sub-trace as part of a longer trace. In the example above, assume s1s0s2 is an emulating sub-trace for the difference sub-trace s1s3s2. We can now use the emulating sub-trace to substitute in the complete trace. The trace s0s1s0s2s0 has the same effect on a specified property and contains no difference states: it is an emulating trace.

This principle can also be applied to traces of infinite length. For example, consider the difference trace s0s1s3s2s0 where these states cycle infinitely often. Assume that s1s0s2 is an emulating sub-trace for the difference sub-trace s1s3s2. On each occasion the difference trace enters the difference sub-trace s1s3s2 we can substitute the emulating trace s1s0s2; this provides an emulation of the infinite trace.

Our approach is therefore to identify every possible difference sub-trace and then to find an *emulating sub-trace* to emulate its behaviour using only non-difference states.

This method can be thought of as an exhaustive search, as would be performed by a Spin verification, of the difference behaviour for a system. As with Spin, we are able to cope with traces that are potentially infinite by capturing the finite number of effects on a property these traces may cause. With a finite number of effects, and a finite number of states which could begin and end the difference sub-traces, there are also a finite number of difference sub-traces. Some exceptional cases will be discussed in section 6.6.

Note that an emulating sub-trace need not contain the same number of states as its difference sub-trace, provided it starts and ends in the same place and emulates the behaviour with respect to a specified property. How we determine this behaviour is discussed in the next section.

6.4 Trace Effect

In Spin, a property is represented as a Büchi automaton. Transitions between states are labelled by propositional logic formulae where the propositions are boolean expressions from the Promela model, for example, the propositions p and q from the examples in section 5.1. To verify a property, the automaton is treated as a process and run concurrently (synchronously) with the model, with property transitions traversed as the labelling conditions become true. It is the sequence of property states which dictates whether the property will be true or false. For more on the theoretical background to model checking temporal properties, see [4].

To determine the effect of a trace, such as a difference trace, we determine what variables will have values assigned by the statements associated with a trace. From this, we can identify what possible sequences of property states will occur. To ensure enough information for this analysis, we must carry in each trace the possible property states along with the relevant variable values as they are altered. By emulating difference sub-traces we make no assumption about the property states or variable values at the start of the sub-trace and therefore must check all combinations.

Consider the example difference sub-trace s1s3s2 and the specified property **user0 receive**. The difference trace s1s3s2 corresponds to the message statements {user1:chans[1]!u1,u0; user1:chans[0]?eval(u0,u1)}. By examining the Promela representation in section 5 we can see that the first message event causes the assignment lastaction[myid] = send. This assignment alters the value of the variable lastaction[0] (for this particular process instance), which is referenced in the proposition p in the property **user0 receive**. In this case, the value is set to send, making the proposition false. This effect on the truth value of the proposition will cause some transitions in the property automaton: it is this effect that we must emulate. Note that other statements may only alter variables which have no effect upon the specified property: such statements can be safely ignored.

6.5 Emulation Checking

So far we have discussed only emulating an individual difference trace, but property preservation testing involves checking *all* difference sub-traces. We call this procedure *emulation checking*. Emulation checking works in two stages:

1. Identify all difference sub-traces and their effect on the specified property.
2. Attempt to find an emulating sub-trace for each difference sub-trace.

In the first step, to identify all difference sub-traces, we identify all the message states at which there is a transition to a difference state. From these states we find all the traces that contain a number of difference states concluding with a non-difference state. In the simple example, from the state s1 there are difference sub-traces s1s3s1 and s1s3s2. From the state s2 there are difference sub-traces s2s3s2 and s3s2s1. The effect of each difference sub-trace, on the specified properties, is then determined by the method described in section 6.4.

In the second step, we attempt to find emulating sub-traces for each difference sub-trace. This part of the procedure is uncertain because the identification of emulating sub-traces is based on heuristic search. There may be a variety of routes that match the start and end states of a difference sub-trace and pass through only non-difference states but we must find one which also matches the effect of the difference sub-trace. Details of how we carry out this search and various optimisations are detailed in [13].

6.6 Further Detail

Due to space constraints we have not described the full detail of the property preservation test here. This includes dealing with ambiguous message states, addressing infinite difference traces and reducing the effort of calculating trace effect. This detail can be found [13].

7 Soundness and Completeness

It is crucial that our method is sound. If it is not sound, a user could believe that a property was preserved when in fact it is not. This could lead to an unsafe change to a communication structure — obviously undesirable. If our method is not complete, some safe alteration will be rejected as not property preserving, but then the user is back to where they started. Although they cannot verify their system, at least they do not have an untrustworthy verification result. Our method aims to be sound, but not necessarily complete. (Note, an algorithmic method cannot be complete, if the initial representation is intractable.)

Throughout section 6 we have implicitly justified the soundness of our technique. In section 6.3 we described how even infinite difference traces must consist of parts which can be emulated directly and difference sub-traces, of which there are a finite number. In section 6.4 we described how to ensure we capture all the possible behaviours which may be due to a difference sub-trace. In fact, there are some cases in which our method is not sound, based on the use of *control variables* within the Promela representation. These are relatively specialised cases and can be easily identified; for more detail, see [13].

The method is not complete because, as discussed in section 6.5, identifying all emulating traces can be a difficult task which may not be achieved. In fact, even if the identification of emulating traces was perfect, our method would still not be complete because the method only determines whether any unique behaviour exists, not whether this behaviour necessarily alters the verification of a specified property.

8 Implementation

We have implemented a tool based on the techniques described in section 6. The tool is approximately 8000 lines written in the scripting language Python [8] and interfaces with Spin as well as providing output via the graph drawing package dot [7]. As input, the tool requires a system modelled in Promela using two different communication structures and a specified property in the file format used by Spin. As output, it returns whether the property is preserved between the two structures. If the property is not preserved, it provides a trace for one communication structure which cannot be emulated by the other. This tool was used to generate the results presented in section 10.

9 Communication Structure Selection

Whether we are constructing a new representation or applying communication structure alteration, we must be able to choose a communication structure that will result in a small(er) state space. To achieve this, we have devised a set of guidelines for choosing a communication structure.

In general, we cannot know which communication structure will result in the smallest state space without model checking all conceivable communication structures. So, for some unusual systems, these guidelines may not provide the best choice of state space. However, intuition and empirical observation suggest that they are effective in most cases.

Few Channels. As demonstrated by the example in section 2, a smaller number of channels results in fewer combinations of messages. This in turn should result in a smaller state space.

Short Channels. As with few channels, the shorter the channels the fewer combinations and therefore the smaller the state space.

Use Exclusive Send/Exclusive Receive. Spin includes constructs to improve the application of partial order reduction [10] to communication operations. As far as possible within the other guidelines, channel structures which make use of these constructs should be employed.

Avoid Too Many Shared Channels. If channels are shared between more than 2 groups of processes it is very common for deadlocks to be created. To maintain a model without deadlocks, too many shared channels should be avoided.

10 Case Study

To illustrate our method, we expand the simple example from section 2.

10.1 System Description

The system we have chosen to model is based on the simple system with three users, where each process is an Email relay server, transmitting Email messages to the other servers. Once the Email arrives at the server it is either read or discarded by its intended recipient. To introduce some extra complexity into the model, each relay is capable of sending either legitimate Email or unsolicited junk Email, also known as *spam*. If a particular relay sends more than a fixed number of spam Emails, it is placed on a *block list*, and no further Emails from this relay are read — they are received, but immediately discarded. The identification of a message as spam is assumed to be perfect.

10.2 Specified Properties

We consider three specified properties, which we describe informally by:

- **no spam** spam messages are received but are not read.
- **blocked** once a relay is blocked, its messages are never read again.
- **arrival** unless a relay is blocked, sent non-spam messages are eventually read.

10.3 Communication Structures

The communication structures are as shown in figure 2 with three users. The initial structure, i.e. case (a), uses dedicated channels, one for each connection. The candidate structure, i.e. case (b), uses shared channels, where one channel is used to connect each pair of processes. In both cases all channels are of size 1.

10.4 Property Preservation Test Results

Applying our property preservation testing method to the example shows that all three properties are preserved. For detailed statistics about each run, see [13]. In summary, in the worst case the message automata for the initial and candidate representations were of size 1745 and 112 respectively and this resulted in a total of 356 difference sub-traces comprising 11970 trace states. The longest run of the tool took around half an hour.

10.5 Spin Results

Table 2 shows the results of using Spin to verify the specified properties with both representations[1]. In this table, N/A indicates that Spin was unable to complete an exhaustive search with the available memory.

From this table we can see that in each case, the number of states in the candidate communication structure is less than that of the initial communication structure. In the case of the properties **blocked** and **arrival**, state spaces which were previously intractable — too large to fit into available memory — have been made tractable by altering the communication structure. In the **no spam** case, where both the initial and the candidate models are tractable, Spin confirms that the properties are preserved (and the candidate state space is smaller).

This case study shows that our method yields smaller state spaces, and is applicable to systems with (initially) intractable state spaces.

[1] These tests were run with a PC running Linux, with memory set to 1GB.

Table 2. Spin verification results.

	Result		States Stored	
	Initial	Candidate	Initial	Candidate
no spam	True	True	6.67341e+06	1.27515e+06
blocked	?	True	N/A	5.59176e+06
arrival	?	True	N/A	9.11464e+06

11 Related Work

The idea that alteration of a Promela representation to reduce the state space has also been presented in [12]. The results presented in [12] is more from the perspective of a Promela programmers guide, whereas we aim for a more rigorous semantic equivalence.

The construction of a communication automaton is similar to the use of *slicing* [14,9], since it slices away non-communication behaviour. Our method differs from slicing in that it attempts a more radical alteration of the Promela code. Instead of simply trimming away parts of the model which are unnecessary, we alter the underlying structure of the system.

In [3], the authors describe the abstraction of communication channels in Promela to explore an otherwise intractable search space. They advocate reducing the number of messages that are passed. Like our work, [3] recognises the significance of communication however, the authors apply abstraction to the *messages* on the channels, rather than by altering the communication structure.

12 Limitations and Further Work

The difficulty of identifying emulating traces makes our prototype tool fairly slow: some of the results presented took several hours to generate. A more efficient implementation would make the method more accessible.

The communication structure choice guidelines are a good starting point, but would be difficult to apply them automatically. An algorithmic method for choosing good communication structures would be preferable.

13 Conclusion

We have proposed a method for reducing the state space size for a model of a communicating system by altering the communication structure. We have provided guidelines for choosing an appropriate communication structure, and presented a method for determining property preservation between two models with different communication structures. We have also presented an example which shows these principles in action. The major contribution is a method which is applicable even when the initial representation is intractable. The method is sound (except for a few special cases), but not necessarily complete.

References

1. Muffy Calder and Alice Miller. Using SPIN for feature interaction analysis - a case study. In *Lecture Notes in Computer Science*, volume 2057 of *Proceedings of the 8th International SPIN Workshop (SPIN2001)*, pages 143–162, May 2001.
2. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
3. E. Fersman and B. Jonsson. Abstraction of communication channels in promela: A case study, 2000.
4. Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
5. Gerard Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
6. G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.
7. Eleftherios Koutsofios and Stephen North. Drawing graphs with *dot*. Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, USA, September 1991.
8. Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, Fall 1996.
9. L. Millett and T. Teitelbaum. Slicing promela and its applications to model checking, 1998.
10. D. Peled. Combining partial order reductions with on-the-fly model-checking. *Lecture Notes in Computer Science*, 818:377–390, 1994.
11. Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *Partial Orders Methods in Verification*, DIMACS, pages 233–257, Princeton, NJ, USA, 1996. American Mathematical Society.
12. Theo C. Ruys. Low-fat recipes for spin. In *Lecture Notes in Computer Science*, volume 1885 of *7th SPIN workshop*, pages 287–321. Springer Verlag, sep 2000.
13. Peter Saffrey. *Optimising Communication Structure for Model Checking*. PhD thesis, Department of Computing Science, University of Glasgow, July 2003.
14. Frank Tip. A survey of program slicing techniques. Technical Report CS-R9438, CWI - Centrum voor Wiskunde en Informatica, July 31, 1994.