Generalising Feature Interactions in Email

Muffy Calder and Alice Miller
Department of Computing Science
University of Glasgow
Glasgow, Scotland.
muffy,alice@dcs.gla.ac.uk

Abstract.

We report on a property-based approach to feature interaction analysis for a client-server email system. The model is based upon Hall's email model [12] presented at FIW'00 [3], but the implementation is at a lower level of abstraction, employing non-determinism and asynchronous communication; it is a challenge to avoid deadlock and race conditions. Our analysis differs in two ways: interaction analysis is fully automated, based on model-checking the entire state-space, and the results are scalable, that is they generalise to email systems consisting of any number of email clients.

Abstraction techniques are used to prove the general results. The key idea is to model-check a system consisting of a constant number (m) of client processes, in parallel with a mailer process and an "abstract" process which represents the product of any number of other (possibly featured) client processes. We give a lower bound for the value of m.

All of the models – for any specified set of client processes and selected features – are generated automatically using Perl scripts.

1 Introduction

We consider modelling features and analysing feature interactions in an *email* system. Our model is derived from Hall's email model [12] presented at FIW'00 [3], but our analysis differs in two significant ways:

- interaction analysis is *fully automated*, based on a model-checking approach,
- results *generalise* to email systems consisting of *any* number of email clients.

We adopt a *property-based* approach to interaction analysis [4], that is we develop an explict model of the basic service and features which is checked against a set of more abstract, temporal properties. Interactions are uncovered through the analysis of property violations. The (parameterised) model is developed in Promela [13], a high-level, state-based, language for modelling (asynchronously) communicating, concurrent processes. Spin is the bespoke model-checker for Promela. Individual models and model-checking runs are generated using Perl scripts.

Our first goal is faithful modelling of an email system as client-server with explicit concurreny and asynchronous communication; this is challenging for a property based approach because of the high degree of concurrency and consequent state-space explosion. Nevertheless feature interaction analysis is comprehensive.

Our second goal is generalisation of interaction results. Model-checking alone is limited to reasoning about a *given* number of processes. This aspect is often overlooked, and proof for a fixed number, say m, processes, is informally assumed to scale up to imply proof for an *arbitrary* number of processes, i.e. for n processes, for any n. In this paper we address the problem explicitly and show how to generalise results without resorting to explicit induction (which is difficult in this case). Our approach is based upon a combination of abstraction and model-checking.

The paper is divided into two parts, in the first part we consider feature interaction analysis for a fixed number of client processes, in the second part, we consider how to generalise these results to an arbitrary number of clients.

In section 2, we give a brief overview of Promela and Spin. In section 3 we give an overview of the basic email service and feature behaviour, the Promela implementation, the properties for the basic service and features and the corresponding LTL formulae. In section 4 we define feature validation and interaction analysis, and give corresponding results for systems of 3 or 4 client processes. We also discuss how we use Perl scripts and the model-checker Spin for analysis. In section 5 we outline the abstraction technique and give results. We conclude in section 6.

2 Reasoning in Spin

Promela is an imperative, C-like language with additional constructs for non determinism, asynchronous and synchronous communication, dynamic process creation, and mobile connections, i.e. communication channels can be passed along other communication channels. Spin is the bespoke model-checker for Promela and provides several reasoning mechanisms: assertion checking, acceptance and progress states and cycle detection, and satisfaction of temporal properties.

In order to perform verification on a model, Spin translates each process template into a finite automaton and then computes an asynchronous interleaving product of these automata to obtain the global behaviour of the concurrent system. This interleaving product is referred to as the *state-space*.

As well as enabling a search of the state-space to check for deadlock, assertion violations etc., Spin allows the checking of the satisfaction of an LTL formula over all execution paths. The mechanism for doing this is via *never claims* – processes which describe *undesirable* behaviour, and Büchi automata – automata that accept a system execution if and only if that execution forces it to pass through one or more of its accepting states infinitely often [13, 11]. Checking satisfaction of a formula involves the depth-first search of the synchronous product of the automaton corresponding to the concurrent system (model) and the Büchi automaton corresponding to the never-claim.

If the original LTL formula f does not hold, the depth-first search will "catch" at least one execution sequence for which $\neg f$ is true. If f has the form $[]p$, (that is f is a *safety* property), this sequence will contain an *acceptance state* at which $\neg p$ is true. Alternatively, if f has the form $\langle\rangle p$, (that is f is a *liveness* property), the sequence will contain a cycle which can be repeated infinitely often, throughout which $\neg p$ is true. In this case the never-claim is said to contain an *acceptance cycle*. In either case the never claim is said to be *matched*.

When using Spin's LTL converter (a feature of XSpin – Spin's graphical interface) it is possible to check whether a given property holds for *All Executions* or for *No Executions*. A universal quantifier is implicit in the beginning of all LTL formulas and so, to check an LTL property it is natural, therefore, to choose the *All Executions* option. However, we sometimes wish to check that a given property (p say) holds for *some state* along *some execution path*. This is not possible using LTL alone. However, Spin can be used to show that "p holds for *No Executions*" is **not** true (via a never-claim violation), which is equivalent. Therefore, when listing our properties (section 3.4), we use the shorthand $\exists p$, meaning *for some path* p, i.e. for No Executions p is not true.

2.1 Parameters and further options used in Spin verification

When performing verification with Spin, three numeric parameters must be set. These are *Physical Memory Available*, *Estimated State-Space Size* and *Maximum Search Depth*. The meaning of the first of these is clear, and the second controls the size of the state-storage hash table. The *Maximum Search Depth* parameter determines the size of the *search-stack*, where the states in the current search are stored. If comparisons are to be made with other model-checkers, then the value of the Maximum Search Depth should be taken into account.

Partial order reduction (POR) [17] is based on the observation that execution sequences can be divided into equivalence classes whose members are indistinguishable with respect to a property that is to be checked. We apply POR in most cases.

Compression (COM) is a method by which each individual state is encoded in a more efficient way. We apply compression in all cases.

3 Basic email service and features

The email system consists of a number of *clients* and one server, in this case the *mailer*. Each client has a unique mail address. Clients send mail messages, addressed to other clients (or themselves) to the mailer; the mailer delivers mail messages to clients. Communication between client and server is asynchronous. Therefore, mail messages are not necessarily received by clients in the (global) order in which they were sent, but local temporal ordering is maintained, i.e. if client 1 sends messages A and B to client 2, in that order, then client 2 will receive message A before message B. We assume (like Hall) that the system does not lose or corrupt messages, because our motivation is feature interaction analysis, not error detection and/or recovery.

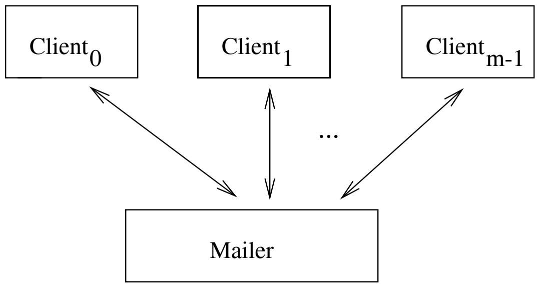

Figure 1: Email service with m clients

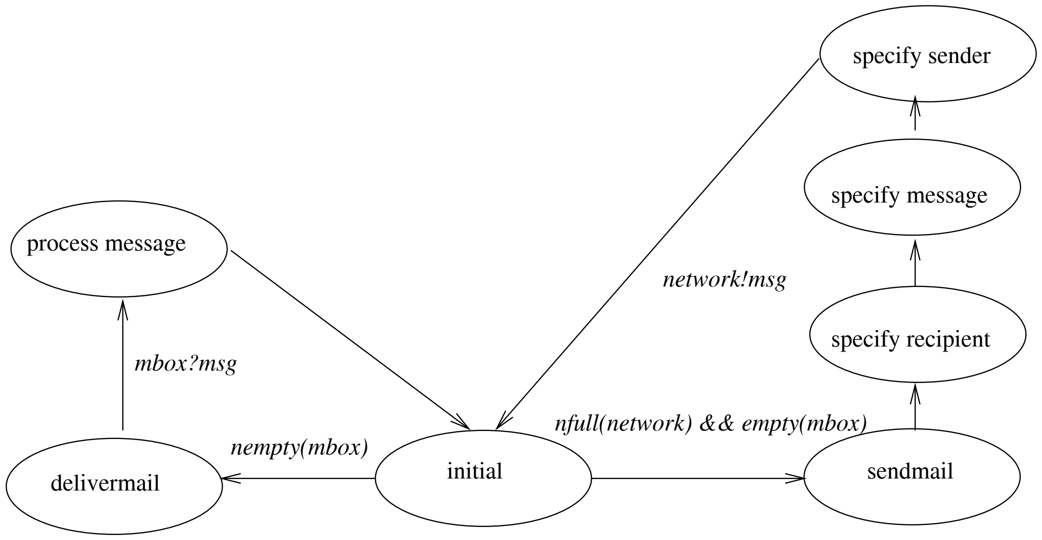

Figure 2: Client process with mailbox *mbox*

We assume (weak) fairness, i.e. an enabled process cannot be ignored infinitely often, when verifying liveness properties (e.g. 3 and 8, see section 3.4). In all other cases, it is not relevant (and just increases the state-space).

The overall system is illustrated in Figure 1. High level, abstract automata for the client and mailer processes are given in figures 2 and 3, respectively. Note that in these figures, transitions are labelled by conditions, e.g. in figure 2 a transition from *initial* to *sendmail* is only possible if the channel *mbox* is empty and the channel *network* is not full. Local and global variables are updated at various points; variable assignments omitted from the diagrams. We refer to states in these abstract automata as *abstract states*, these are not to be confused with states in the Promela model.

3.1 Basic email service in Promela

The model for m client processes consists of m instantiations of the parameterised proctype *Client*, all in parallel with one instance of the proctype *Mailer*.

A mail message consists of a *sender*, *receiver*, *message body*, and *key*. Mail messages may be sent from clients to the mailer, and delivered to clients from the mailer (via a *mailbox* belonging to the receiver). All communication between clients and the mailer is *asynchronous*. We chose to adopt this position because it is closer to actual system behaviour, however we have to take care that it does not result in state explosion. Delivery of mail takes precedence over sending, i.e. a client has to take delivery of any mail which has been delivered by the network, before sending mail. A client can otherwise send mail at any time, provided the channel $network$ has capacity.

The parameter associated with a client process is the identification (a byte) of that process. A client process can either send mail, or have mail delivered, the latter taking precedence over the former. The former can only occur if the network is not full. If neither is possible, the client is, in effect, in *busy waiting*.

Communication between clients and the mailer is via asynchronous, global channels, one for each client and one for the mailer. The sizes of the channels are set using the constants k for the client channels and N for the

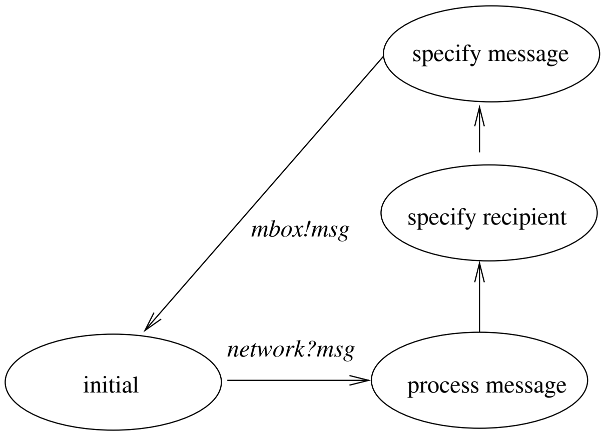

Figure 3: Mailer process

mailer. During most of this investigation $k = 2$ and $N = 4$. M is a constant denoting the number of clients in the given system and is often used to denote a default, or unassigned, value.

The client channels are, in effect, *mailboxes*. The role of the mailer is therefore to deliver mail messages to the appropriate mailboxes; clients take delivery of message by reading from the appropriate mailbox/channel. In figure 3, note that *mbox* is a free variable which must be correctly instantiated during the *specify recipient* abstract state.

Mail addresses are simple integers, used to index the client processes. Initially, we implemented a more sohpisticated addressing mechanism, with login names and hierarchical domains. However, this resulted in a very large state vector and state-space (due in part to Promela's poor handling of structured types). Since we found no additional analysis benefit to this approach (apart from the aesthetic one), we have implemented a simpler, more abstract addressing mechanism.

Mail messages themselves are of no consequence, save to observe whether or not they are encrypted. We denote encrypted text by the value 1 and plain text by the value 0. Keys are simple mail addresses, i.e. simple bytes.

An important issue for any distributed system is that of *atomicity*. This is especially important from a model-checking perspective as it provides a means of controlling state-space explosion and resolving race conditions. Promela provides a facility for grouping together statements as atomic, provided only the initial statement has the potential to block. Our model employs as much atomicity as possible within each consituent process. Specifically, in the $Client$ process, each iteration from the *initial* state, back to the *initial* state, is a single atomic step (i.e. figure 2 encapsulates a single atomic step) – with suitable guards which block if the process can neither read nor write. This is crucial in order to avoid deadlock – since all channels are of finite size. On the other hand, the $Mailer$ process consists of two atomic steps: one for reading a message and the other for sending the appropriate message. Any variable about which we may intend to reason should not be updated more than once within any atomic statement (so that each change to the variables is visible to the never-claim), other variables may of course be updated as required.

Another implementation issue is the size of the state vector, i.e. the number and nature of global and local variables. We must be careful not to introduce any extraneous variables (see also 3.3) nor introduce extraneous state values. To avoid the latter, we must be careful to reset variables when returning to so-called initial, abstract states, to ensure that we are indeed representing the same abstract state.

The interplay between atomicity, the number and nature of variables, and faithful modelling/levels of abstraction is very subtle and a challenge in this domain, particularly due to the asynchronous communication. It took considerable time and expertise to develop a tractable, deadlock free model. Fortunately, we were able to employ some lessons learned from modelling *POTS* [6].

3.2 The Features

We consider here a set of five features.

- **encrypt** a message, using a (private) key, the *intended* recipient.
- **decrypt** a message, using a (private) key, the *actual* recipient.
- **filter** all messages from a given mail address.

- **forward** all messages to another mail address.
- **autorespond** to incoming messages. The automatic response is only sent in response to the first message from a given client. Any subsequent message from that client is received, but no automatic response is issued as a result.

The features encrypt, decrypt and autorespond reside at the client side, the remaining reside at the server side. Note that only the features encrypt and decrypt alter the actual mail message, forwarding does not affect the message.

We have considered all the features proposed in [12]; but for brevity, we omit them here because they do not reveal any further "interesting" behaviour paradigms for our analysis. That is not to say that they do not reveal further interactions, and interesting aspects of email, but that they do not reveal any further aspects with respect to generalisation.

3.3 Features in Promela

Features are implemented within the Promela model via a number of *inline* functions (procedures with dynamic bindings). Most features are relatively straightforward to implement, simply involving additional transitions or steps during one or more of the abstract states of the client or mailer processes.

The exception is autorespond, because this feature involves both *reading* – a message from a client channel, and *writing* – a message to the network channel. Both events are potentially blocking, hence cannot take place within one atomic step. Therefore, to implement this feature we add an additional data structure to indicate whether or not a client requires to send an autoresponse. We enhance the *initial* state to include the possibility that an autoresponse message needs to be sent, and give priority to this over any other event. This means that it is possible for an autoresponse to *never* be sent, if the network channel is continuously full. We cannot avoid this situation[1].

In order to reason about feature behaviour (see section 3.4) we introduce a number of "observation" variables. These are not integral to the behaviour of the service and/or features, but exist solely for the purposes of analysis. For this reason, these variables are included/excluded on a per model/property basis.

Examples of "observation" (process indexed) variables include:

- $last\_del\_to_i\_to$ the intended receiver of the mail message last delivered to $Client[i]$
- $last\_del\_to_i\_from$ the intended sender of the mail message last delivered to $Client[i]$
- $last\_del\_to_i\_body$ the body of the mail message last delivered to $Client[i]$.
- $last\_sent\_from_i\_to$ the intended recipient of the mail message last sent from $Client[i]$.

A further variable required both for correct function and for reasoning is the array of bit vectors $autoarray$. The function

- $IS\_0(autoarray[i], j)$ indicates whether $Client[i]$ has already sent an autoresponse to $Client[j]$.

3.4 Feature Properties

We give a small number of illustrative properties for both the basic service and the features. The properties are linear temporal logic (LTL) formulae over propositions about states. Temporal operators include [] (always), $<>$(eventually) and X (next). Propositional connectives are $||$ (disjunction), $\&\&$ (conjunction), $\rightarrow$ (implication) and $\neg$(negation). The path quantifier is (implicitly) $\forall$, except when explicitly given as $\exists$. Feature properties are properties that are expected to hold when *one* such feature is present.

Property 1 – Basic Messages are delivered only to intented recipients.

If $Client[i]$ receives a message from $Mailer$, then the (intended) recipient of that message is $Client[i]$. Alternatively, Client[i] has not yet received any messages (in which case the $last\_del\_to_i\_to$ variable will remain set to the default value M).

[1] In any operational email system, buffers can become full. Although this is unlikely in reality, as model-checking involves the exploration of all *feasible* behaviours, this possibility must be considered.

LTL: $[](p||q)$
p = $(last\_del\_to_i\_to == M)$
q = $(last\_del\_to_i\_to == i)$

Property 2 – Basic Messages can be sent between any two clients.

It is possible for Client[i] to recieve a message such that the sender of that message is Client[j].

LTL: $\exists <> (p)$
p = $(last\_del\_to_i\_from == j)$

Property 3 – Basic Messages are eventually delivered correctly.

If $Client[i]$ *sends a message to* $Client[j]$*, then* $Client[j]$ *will eventually receive a message from* $Client[i]$.

LTL: $[](((\neg p)\&\&X(p)) \rightarrow X(<> q))$
p = $(last\_sent\_from_i\_to == j)$
q = $(last\_del\_to_j\_from == i)$

Property 4 – Encryption Messages are properly encrypted.

If $Client[i]$ *has encryption on, then if Client[j] receives a message whose sender is Client[i], then the message will be encrypted.*

LTL: $[](p \rightarrow q)$
p = $(last\_del\_to_j\_from == i)$
q = $(last\_del\_to_j\_body == 1)$

Property 5 – Decryption Messages are properly decrypted.

If $Client[i]$ *has decryption on, then all messages received by* $Client[i]$ *will have been decrypted.*

LTL: $[](p)$
p = $(last\_del\_to_i\_body == 0)$

Property 6 – Filtering Messages are discarded by a filter.

If Mailer filters messages from Client[i] to Client[j] then it is not possible for Client[j] to receive a message from Client[i].

LTL:$[](\neg p)$
$p == (last\_del\_to_i\_from == j)$

Property 7 – Forwarding Messages are forwarded.

If Client[i] forwards messages to Client [j], then it is possible for Client[j] to receive messages not addressed to Client[j] (or to the default value M*).*

LTL: $\exists <> (\neg(p||q))$
$p == last\_del\_to_j\_to == M)$
$q == (last\_del\_to_j\_to == j)$

Property 8 – Autorespond Single automatic response messages are sent out.

If $Client[i]$ has autorespond on, then if Client[j] sends a message to Client[i], and Client[j] hasn't already received an automatic response from Client [i], then Client[j] will eventually receive a reply from Client[i]. Alternatively, Client[i] eventually stops sending messages because network can't be accessed.

LTL: $[](p-> (<> q||(<> ([]\neg r))))$
p = $((last\_sent\_from_j\_to == i)\&\&((autoarray[i,j] == 0)))$
q = $(last\_del\_to_j\_from == i)$
r = $(network??[i,x,y,z])$
(The function $network??[i,x,y,z]$ determines whether there is a message *at any position* on the network channel in which the sender field is i.)

4 Analysis for a constant number of clients

The basic idea of feature interaction analysis is to detect when features behave as expected in isolation, but not in the presence of each other. So, interaction analysis involves feature *validation* (checking a feature in isolation) and then analysis of *tuples* of features (checking for violation of expected behaviour). Fortunately, we need only restrict our attention to pairwise analysis, as empirical evidence shows that it is extremely rare to have a 3-way interaction which is not detected as 2-way interaction [14]. In each case we consider a model consisting of either 3 or 4 client processes and 1 mailer. An example Promela model of a system of 3 Client processes and a Mailer process in which $Client[0]$ has encryption, $Client[1]$ filters messages from $Client[2]$ and property 4 is to be verified for $i = 0$, $j = 2$ can be found on our website at [5].

For all verification runs we used a PC with a 2.4GHZ Intel Xenon processor, 3GB of available main memory, running Linux (2.4.18).

An overview of the reasoning process is given in Figure 4.

4.1 Use of Perl Scripts

For each pair of features, set of feature parameters, associated property and set of property parameters, a relevant model needs to be individually constructed, to ensure that only relevant variables are included and set. We have developed two Perl scripts, *mailchange.pl* and *auto_mailchange.pl* for automatically configuring the model and for generating model-checking runs. These scripts greatly reduce the time to prepare each model and the scope for errors.

During initial investigations, *mailchange.pl* is used to generate a model for a given set of features, feature parameters, property and property parameters. The resulting model is then loaded into SPIN with an appropriate set of search parameters (MSD, POR, WF for example) and results interpreted manually. Once confidence has been gained in the model, suitable values assertained for the value of MSD and the applicability of POR and WF determined for successful verification in each case, *auto_mailchange.pl* is used to iteratively select pairs of features and parameters, set up model checking runs and interpret results. An overnight run is required to collect all results from all pairs of features and suitable parameter sets. It is important to note that a certain amount of simple symmetry reduction is incorporated within the Perl script to avoid repeating runs of configurations which are identical up to renaming of processes.

The interpretation stage of *auto_mailchange.pl* involves reading the output file from the SPIN verification run. Firstly it is checked that the maximum search depth (MSD) has not been reached and that the total memory available has not been exhausted. If neither of these is true, the second phase of the interpretation phase involves checking if there are any errors - and as such, whether the associated property is true. Finally, the interpretation stage involves determining whether a feature interaction has occurred and, if so, what type (SU or MU). All parameter sets and corresponding results are output to a results file.

4.2 Results - single property validation

Table 1 gives the results of verification of properties 1 – 8. In each case the feature (if any) associated with the property is given in the column labelled 'feature'. When there is no feature present, (during the verification of properties 1–3) the term 'basic' is written in this column. When a feature is present, the verification corresponds to checking the associated property for a model consisting of a Mailer process, two basic Client processes and a Client processes for which the given feature is 'turned on'. (When no feature is present the associated model simply consists of a $Mailer$ process and three $Client$ processes.) In all cases we give results for verification of the model in which $Client[0]$ has the given feature (in relation to $Client[1]$ if appropriate) and i (and j) is (are) assigned the value(s) 0 (and 1). The Prop column contains the property being checked and a $\surd$ or a

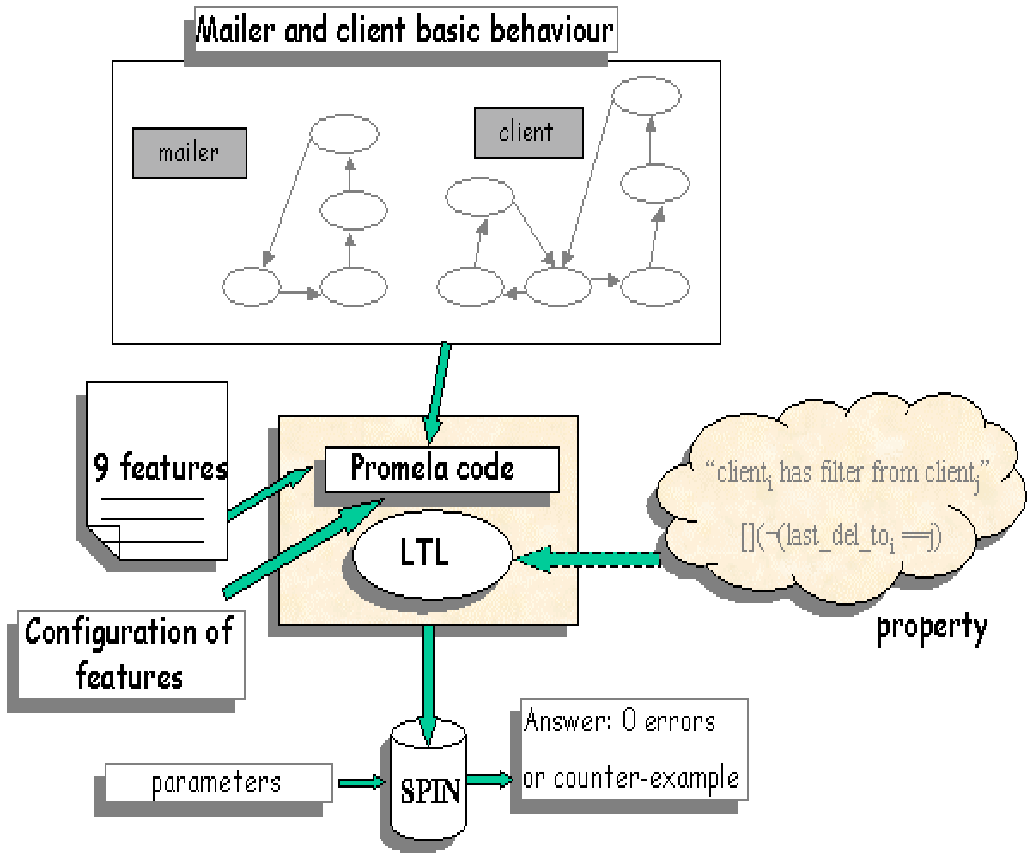
Mailer and client basic behaviour
mailer
client
9 features
Promela code
LTL
"client$_i$ has filter from client$_j$"
$[](\neg(last\_del\_to_i == j))$
property
Configuration of features
parameters
SPIN
Answer: 0 errors
or counter-example

Figure 4: The reasoning set up

Table 1: Results of verification of the properties

Feature	Prop	WF?	POR?	MSD ($\times 10^4$)	States ($\times 10^4$)	Depth (Mb)	Mem (s)	Time
basic	1	$\times$	$\surd$	6	8.3	52989	4.4	2
basic	2	$\times$	$\surd$	0.01	0.001	85	2.3	0.1
basic	3	$\surd$	$\times$	17	98.2	162286	38.2	44
encryption	4	$\times$	$\surd$	13	9.3	121792	4.8	3
decryption	5	$\times$	$\surd$	5	10.2	45365	4.2	3
filtering	6	$\times$	$\surd$	6	7.1	52069	3.7	2
forwarding	7	$\times$	$\surd$	0.01	0.001	66	2.3	0.1
autorespond	8	$\surd$	$\surd$	17	250	163176	111.5	334

$\times$ in the 'WF?' and 'POR?' columns indicate whether weak-fairness and partial order reduction are selected respectively. Note that property 3 is the only property for which POR is not applied. This is due to the presence of the next operator (X) in the property. Also, WF is only applied during the verification of *liveness* properties – properties that contain the *eventually* operator $<>$. The entries in the MSD column show the value to which the maximum search depth is set prior to verification. The remaining columns contain the number of stored states, the depth reached, the memory required for state storage (in Mbyte) and the time taken (in seconds) for each verification.

4.3 Results - feature interactions

Now we turn our attention to consideration of *pairs* of features. For each pair of features we generate a model for each distinct set of parameters (the union of the sets of parameters for each feature) and for each appropriate set of property parameters. This may mean that up to 5 client processes are required. For example, if $Client[i]$ has filtering from $Client[j]$ and $Client[k]$ has filtering from $Client[l]$, (i, j, k, l distinct), then 4 client processes are required. For each suitable pair of features, f_i, f_j, an interaction is said to occur if the feature property associated with f_i does not hold for the model in which features are f_i and f_j are present. Note that we do not consider the basic service in our analysis, as all other features interact with it in some way. This can be determined without the need for model-checking.

We enumerate the interactions found below. In each case, we indicate whether the interaction is single user (SU), i.e. both features reside at the same network component, or multiple user (MU), i.e. the features reside at different network components. We also give a *witness* for the interaction. We do not give details of timing or memory requirements etc. as these vary depending on the parameter set under consideration. (There are 111 feasible parameter sets after symmetry reduction. It would be impractical to give details of such requirements for each case.) In some cases more $Client$ processes are required to fully check for interaction. Clearly when more $Client$s are required, verification takes longer and more memory is required. In addition, when an error is reported during verification (in most cases, excluding the verification of property 2, indicating an interaction) a full search of the state-space is not required. This again results in far smaller time and memory requirements.

1. encryption and decryption (SU)
   witness i=j=0 – $Client[i]$ has encryption and decryption

2. encryption and decryption (MU)
   witness i=0, j=1 – $Client[i]$ has encryption, $Client[j]$ has decryption

3. filter and forward (MU)
   witness $i = 0, j = 1$ – $Client[i]$ has filter from j, $Client[j]$ has forwarding to i

4. forward and forward (MU)
   witness i=0,j=1,k=2 – $Client[i]$ has forwarding to j, $Client[j]$ has forwarding to i

5. autoresponse and filter (SU)
   witness $i = 0, j = 1$ –$Client[i]$ has autoresponse, $Client[i]$ has filter from j

6. autoresponse and filter (MU)
   witness i=0,j=1 – $Client[i]$ has autoresponse, $Client[j]$ has filter from i

7. autoresponse and forward (SU)
   witness $i = 0, j = 2$ – $Client[i]$ has autoresponse, $Client[i]$ has forwarding to j

8. autoresponse and forward (MU)
   witness $i = 0, j = 1$ – $Client[i]$ has autoresponse, $Client[j]$ has forwarding to i

Each of these pairs of features are listed in Hall's results [12] but in all cases, he only reports a MU example. While Hall explicitly states that his method is not complete, it is not clear if the SU interactions would be found in his approach, or he stopped after the MU interaction was found. Our method is combinatorially complete. We note that in the MU cases, our witnesses are identical to Hall's (modulo translation).

5 Generalisation

We have shown above that a property holds (or does not hold) for a fixed number of clients, i.e. for $Client[0]||Client[1]||\ldots Client[m]||Mailer$, where $||$ denotes parallel composition. But how can we deduce that (if at all) a property holds for $Client[0]||Client[1]||\ldots||Client[n]||Mailer$, for an arbitrary n? It is not possible to demonstrate this with straight-forward model-checking [1].

More formally, the generalisation problem is how to prove (disprove)

$$M(Client[0]||Client[1]||\ldots||Client[n]||Mailer) \models \phi[0, 1, \ldots, t]$$

where the left hand side is the finite-state model of the parallel composition of client and mailer processes (the former are instances of the parameterised process $Client$) and $\phi[0, 1, \ldots, t]$ is a temporal logic formula containing free variables indexed by $0, 1, \ldots, t$. The indices refer to instances of $Client$ (e.g. the variables i and j in section 3.4). In general, the $Client[i]$ are not isomorphic because they have different sets of features enabled.

We offer a solution based on abstraction and model-checking. The technique and theoretical justification are described in more detail in [8, 7], here we apply the results. Briefly, the technique involves choosing a fixed m, such that t is constrained by $0 \leq t \leq m - 1$. We refer to $Client[0]||\ldots||Client[m-1]$ as *concrete* processes and the $Client[m]||\ldots||Client[n-1]$ as *abstract* processes. We represent the behaviour of $Client[m]||\ldots||Client[n-1]$, by a new abstract process, *Abstractclient*. We do not assume that the (original) abstract processes, i.e. $Client[m]||\ldots||Client[n-1]$ are isomorphic: they might have different combinations of features enabled. However, we do assume that the features are all drawn from our given set and we know how they can communicate with each other and more importantly, how they communicate with the concrete processes.

A model of the m *concrete* processes, together with the abstract process, is generated automatically from a model of the concrete processes together with a single (parameterised) $Client$. This is summarised by Figure 5. The value of m depends upon the particular feature set considered. Here, because each feature involves at most two parameters, a worst case analysis suggests that 5 concrete $Client$s are required, however further detailed analysis shows that in this case, we require only a maximum of 4. For some combinations, 3 will suffice.

Our approach is based upon our result:

$$\begin{array}{l} M(Client[0]||Client[1]||\ldots Client[m]||Abstractclient||Mailer') \models \phi[0, 1, \ldots, t] \\ \quad\Rightarrow \\ M(Client[0]||Client[1]||\ldots Client[n]||Mailer) \models \phi[0, 1, \ldots, t]. \end{array}$$

The process $Mailer'$ is a slightly modified version of $Mailer$, modified to take into account communication with *Abstractclient* (instead of communication with the original abstract processes).

Thus to generalise interaction analysis results, we need only consider interaction analysis of the *finite* (model of) $Client[0]||Client[1]||\ldots Client[m]||Abstractclient||Mailer'$.

In the next section we outline the form of $Abstractclient$ and $Mailer'$ and give our interaction analysis results.

5.1 Abstractclient specification and analysis results

$Abstractclient$ is defined as follows. $Abstractclient$ can only affect the behaviour of the m concrete processes indirectly via $Mailer'$. Therefore, communication to/from a concrete process from/to $Mailer'$ takes place via a *virtual* channel. Rather than concrete processes reading/writing to this (virtual) channel and behaving accordingly, each possible read is replaced by a non-deterministic choice over the possible contents of the channel. In this way, all possible behaviours are explored (a write to the virtual channel is not relevant).

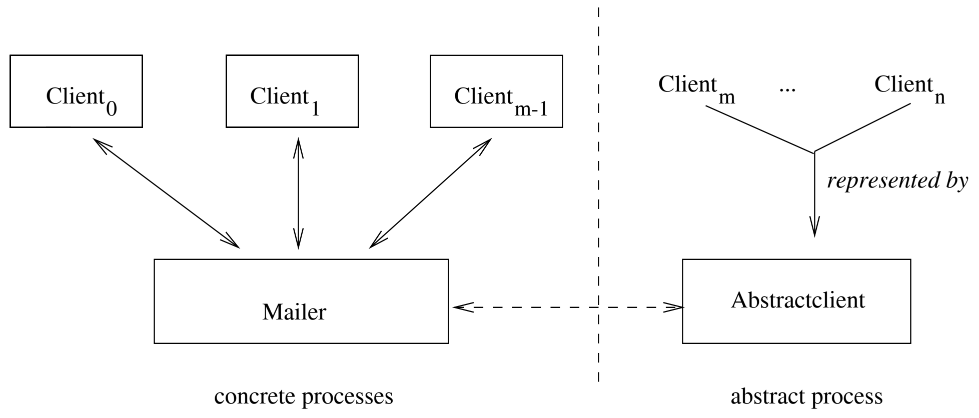

Figure 5: Generalised Email service

As an example, when $m = 3$ $Abstractclient$ is as follows:

```
proctype Abstractclient(byte id)
{Mail msg;
 atomic
 {
 msg.receiver=M;
 msg.sender=M;
 msg.key=M;
 msg.body=0};
 do
 ::blocked==1->blocked=0
 ::atomic{nfull(network)->
   if
   ::msg.receiver=0
   ::msg.receiver=1
   ::msg.receiver=2
   ::msg.receiver=3
   /*another client within Abs process */
   fi;
   msg.sender=id;
   network!msg;
   msg.receiver=M;
   msg.sender=M}
 od
 }
```

Note that the $Abstractclient$ process can send messages to $Mailer'$ (via the $network$ channel), but does not receive messages. *Abstractclient* is also able to set the *blocked* variable to 0. This simulates “unblocking” $Mailer'$ when it is unable to “send” a message to a particular process within the abstract process. Note that *Abstractclient* is always able to unblock $Mailer'$ but can only send messages when the $network$ channel is not full. This reflects the finite model. When $Mailer'$ wants to “deliver” a message to *Abstractclient*, it first checks whether the relevant channel is blocked (via non-deterministic choice). If so, $Mailer'$ waits until the channel becomes unblocked (when the *blocked* variable is reset to 0 by *Abstractclient*) before delivering the message. (In fact no message is actually sent, but $Mailer'$ continues as if it has been.) Here we give the new $Mailer$ proctype:

```
proctype Mailer'()
{
Mail msg;
chan deliverbox=null;
atomic{
bit myanswer=0;
msg.sender = M;
msg.receiver= M;
      msg.key=M;
      msg.body=0;
}
loop:
atomic{
       network?msg;
```

```
        filter_message(msg.receiver,msg.sender,myanswer);
        if
        :: myanswer -> /* throw away message from this sender*/
           myanswer=0;
           msg.sender = M;
           msg.receiver= M;
           msg.body = 0; msg.key = M;
           deliverbox = null;goto loop
        :: else -> skip
        fi;
        if
        ::msg.receiver==3->/* abstract process */
          if
          :: blocked=0
          :: blocked=1
          fi
        ::else->
          mailbox_lookup(msg.receiver,deliverbox)
        fi;
/* now pass on message */
}

atomic{

if
::((msg.receiver!=3)&&(nfull(deliverbox)))->deliverbox!msg
::((msg.receiver==3)&&(blocked==0))->skip /*delivered virtual message*/
fi;

/*reset variables to initial values*/
msg.sender = M; msg.receiver= M; msg.body = 0; msg.key = M;
deliverbox = null;
goto loop
      }}
```

The concrete $Client$ processes are declared in the usual way and communication between them and *Mailer* is unchanged.

It is important to note that this model is not, strictly, a conservative extension, because *Abstractclient* allows additional behaviour. Namely, an (abstract) client can send mail even when there is mail to be delivered (to that client). This is not possible in any concrete model. However, the constraint that mail delivery takes priority is in the concrete model only to prevent deadlock (when mail buffers are full), not for any reason of *functional* behaviour. Relaxing this constraint in the general model neither allows deadlock nor affects the observational behaviour of the system. Thus the constraint is safely relaxed and our approach is sound.

The interaction analysis results reveal no new interactions, nor new witnesses. We therefore do not give details, save to indicate that time and space complexity lie in between those for the system with m (concrete) $Clients$ and $m + 1$ (concrete) $Clients$. The value of m depends on the parameter set and the property to be verified. Again, all analysis was automated through the use of Perl scripts.

An example Promela model of a system of $m = 3$ $Client$ processes, a $Mailer$' process and an $Abstractclient$ process in which $Client[0]$ has encryption, $Client[1]$ filters messages from $Client[2]$ and property 4 is to be verified for $i = 0, j = 2$ can be found in the appendix below. It can also be found on our website at [5].

6 Conclusions

We have developed a property-based approach to feature interaction analysis for a client-server email system. The feature set described here is not as extensive as Hall's [12], but it is sufficient to reveal most of the interesting behaviour (from a modelling point of view) and to validate our approach. On the other hand, our analysis is complete and fully automated. We note that a difficult feature for our implementation was autoresponder, this is because unlike most other features, it initiated the sending of a completely new message. This was a difficulty because we chose to use asynchronous communication, with fixed size communication channels (thus leading to more interleavings and potentially, more interactions).

Additionally, our results are scalable, that is they generalise to email systems consisting of any number of $Client$ processes.

Abstraction techniques are used to prove the general results. The key idea is to model-check a system consisting of a constant number (m) of client processes, in parallel with an "abstract" process which represents the product of any number of other client processes. We give a lower bound for the value of m, for our given feature set.

While the general results did not reveal any new interactions (and we admit it is difficult to think of situations where they would, for *fixed* feature sets), it is nevertheless important to prove rigorously that results scale up. We have achieved this.

Our results demonstrate the feasibility of the abstraction technique for this application domain – the model-checking requirements are well within the capability of our machine. Also, the transformation to a general model is relatively straightforward: we need only consider the *communication* between the abstract process and the concrete process(es). (In this case, we need only consider communication with $Mailer$ process, as there is no peer-peer communication.) An alternative, an induction approach [9, 15], requires the construction of an inductive invariant. This involves incorporating the behaviour of the entire system within the invariant; moreover, it requires that both the concrete and abstract m $Client$s are isomorphic. Our abstraction approach offers a more suitable and tractable alternative. However, at some level they are similar, future work aims to establish this.

References

[1] Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986.

[2] M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems*, volume VI. IOS Press, Amsterdam, 2000.

[3] M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press (Amsterdam), 2000.

[4] M. Calder, E. Magill, and S. Kolberg, Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41/1:115 – 141, 2003.

[5] M. Calder and A. Miller. Veriscope publications website: `http://www.dcs.gla.ac.uk/research/veriscope/publications.html`.

[6] M. Calder and A. Miller. Using SPIN for feature interaction analysis - a case study. In *[10]*, pages 143–162, 2001.

[7] M. Calder and A. Miller. Feature validation for any number of processes. Technical Report TR2002-110, University of Glasgow, Department of Computing Science, 2002.

[8] Muffy Calder and Alice Miller. Automatic verification of any number of concurrent, communicating processes. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 227–230, Edinburgh, UK, September 2002. IEEE Computer Society Press.

[9] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In *[16]*, pages 395–407, 1995.

[10] M.B. Dwyer, editor. *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, Toronto, Canada, May 2001. Springer-Verlag.

[11] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th international Conference on Protocol Specification Testing and Verification (PSTV '95)*, pages 3–18. Chapman & Hall, Warsaw, Poland, 1995.

[12] R.J. Hall. Feature interactions in electronic mail. In *[3]*, pages 67–82, 2000.

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

[14] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Results of the second feature interaction contest. In *[2]*, pages 311–325, May 2000.

[15] R. P. Kurshan and K.L. McMillan. A structural induction theorem for processes. In *Proceedings of the eighth Annual ACM Symposium on Principles of Distrubuted Computing*, pages 239–247. ACM Press, 1989.

[16] Insup Lee and Scott A. Smolka, editors. *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR '95)*, volume 962 of *Lecture Notes in Computer Science*, Philadelphia, PA., August 1995. Springer-Verlag.

[17] Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *[18]*, pages 233–257, 1996.

[18] Doron A. Peled, Vaughan R. Pratt, and Gerard J. Holzmann, editors. *Proceedings of the DIMACS Workshop on Partial-Order Methods in Verification (POMIV '96)*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

Appendix: The email service with features

The following model is generated (by a Perl script from template) for features: encryption[0] and filter[1]=2 to verify property 4 with i=0 and j=2.

```
/*Email model with  Client processes */
/*plus Mailer process */
/*Generated from template*/
/* Assume that no more than 8 client processes */
/* Features: filter,  forward, encrypt, decrypt,autorespond */
/* Muffy Calder and Alice Miller */

/* Message and Address Format */
/* a mail message consists of sender, recipient, message body, and key */
/* message bodies are either text or encrypted  */
/* a mail address consists of byte */

/*With property 4 relating to Client[0] and Client[2]*/
/*If Client[0] has encryption, and Client[0] sends
a message to Client[2], the message will be encrypted on receipt.*/

typedef Mail {byte sender;
              byte receiver;
              byte key;
              bit body};

#define BITV_8 byte
#define ALL_1s = 256
#define SET_0(bv,i) bv=bv&(~(1<<i))
#define SET_1(bv,i) bv=bv|(1<<i)
#define SET_ALL_0(bv) bv=0
#define SET_ALL_1(bv,n) bv=ALL_1s
#define IS_0(bv,i) (!(bv&(1<<i)))
#define IS_1(bv,i) (bv&(1<<i))

#define k 1 /*size of Mailbox */
#define N 2  /* size of network channel */

/*#define no_clients 3 */

#define M 4 /* default value of variables. */
/*One more than any id*/
/* equal to no_Clients + 2*/

bit blocked=0;

chan null = [k] of {Mail}; chan zero = [k] of {Mail};
chan one = [k] of {Mail};  chan two = [k] of {Mail};
chan network = [N] of {Mail};

BITV_8 Encrypt=0;
byte Filter[M]=M;
byte last_del_to2_from=M; bit last_del_to2_body=0;

inline mailbox_lookup(login,box)
{
if
:: (login==0) -> box = zero
:: (login==1) -> box = one
:: (login==2) -> box = two
fi}

inline encrypt_message(login,answer)
{if
::(IS_1(Encrypt,login))->answer=1
::else->answer=0
```

```
fi
/*if encryption is on, answer=1, otherwise answer=0*/
}

inline filter_message(to,from,answer)
{if
  ::(Filter[to]==from)->answer=1
  ::else->answer=0
fi
/*if appropriate filter is on, answer=1, otherwise answer=0*/
}

inline reset_vars(i)
{if
::i==2->last_del_to2_body=0;last_del_to2_from=M
::else->skip
fi
}

inline set_deliv_vars(i,from,to)
{if

::i==2->last_del_to2_from=from
::else->skip
fi
}

inline set_body(i,text)
{if
::i==2->last_del_to2_body=text
::else->skip
fi
}


proctype Client(byte id)
{chan mybox=null;
Mail msg;

atomic{
msg.sender=M; msg.receiver=M; msg.key=M; msg.body=0;
bool myanswer=0;

/*get appropriate mailbox*/
mailbox_lookup(id,mybox)};

initial:
atomic
{
 (nempty(mybox) ||nfull(network));
 /* wait here if cannot send or deliver */
 reset_vars(id);
 /* deliver mail, if mail present */
 /* otherwise may send mail  */
  if
  :: nempty(mybox) -> goto delivermail
  :: empty(mybox) && nfull(network) ->  goto sendmail
fi;

 delivermail:
 mybox?msg;
 set_body(id,msg.body);
 set_deliv_vars(id,msg.sender,msg.receiver);
 goto endClient;

 sendmail:
 /*specify recipient */
 if
 :: msg.receiver= 0
 :: msg.receiver= 1
 :: msg.receiver= 2
 fi;
 /* specify message */
 encrypt_message(id,myanswer);
 if
 :: myanswer -> /* encryption is on */
    myanswer=0;
    /* use  reciever id as  key */
    msg.body = 1;
    msg.key = msg.receiver
 :: else -> msg.body = 0;  /*no encryption */
 fi;
 /* specify sender */
 msg.sender = id;
 network!msg;
 /* send mail */

 endClient:
 /* reset other variables */
 msg.sender = M; msg.receiver = M;
 msg.body = 0; msg.key = M;
 goto initial
}
}/*end Client proctype*/

proctype Network_Mailer()
{
 Mail msg;
 chan deliverbox=null;
 atomic{
 bit myanswer=0; msg.sender = M;
 msg.receiver= M; msg.key=M; msg.body=0;}
 loop:
 atomic{
 network?msg;
 filter_message(msg.receiver,msg.sender,myanswer);
 if
 :: myanswer -> /* throw away message from this sender*/
    myanswer=0;
    msg.sender = M;
    msg.receiver= M;
    msg.body = 0; msg.key = M;
    deliverbox = null;goto loop
 :: else -> skip
 fi;
 if
 ::msg.receiver==3->
   if
   :: blocked=0
   :: blocked=1
   fi
 ::else-> mailbox_lookup(msg.receiver,deliverbox)
 fi;
 /* now pass on message */
 }
 atomic{
 if
 ::((msg.receiver!=3)&&(nfull(deliverbox)))->deliverbox!msg
 ::((msg.receiver==3)&&(blocked==0))->skip /*delivered virtual message*/
 fi;
 /*reset variables to initial values*/
 msg.sender = M; msg.receiver= M; msg.body = 0; msg.key = M;
 deliverbox = null;
 goto loop}};
/*end Network_Mailer proctype*/

proctype Abstractclient(byte id)
{Mail msg;
 atomic
 {msg.receiver=M;
 msg.sender=M; msg.key=M; msg.body=0};
 do
 ::blocked==1->blocked=0
 ::atomic{nfull(network)->
   if
   ::msg.receiver=0
   ::msg.receiver=1
   ::msg.receiver=2
   ::msg.receiver=3
     /*another client within Abs process */
   fi;
   msg.sender=id; network!msg; msg.receiver=M; msg.sender=M}
 od}
/*end Abstractclient proctype*/

#define t1 (last_del_to2_from== 0)
#define t2 (last_del_to2_body== 1)

/* []( t1 -> t2 )*/

init
{atomic{
SET_1(Encrypt,0); Filter[1]=2;
run Abstract(3);
run Network_Mailer();
run Client(0);
run Client(1);
run Client(2);
}}

/*
  * Formula As Typed: []  (t1 -> t2)
  * The Never Claim Below Corresponds
  * To The Negated Formula !([]  (t1 -> t2))
  * (formalizing violations of the original)
  */  never {    /* !([]  (t1 -> t2)) */
T0_init:
      if
      :: (! ((t2)) && (t1)) -> goto accept_all
      :: (1) -> goto T0_init
      fi;
accept_all:skip }
```