Verifying parameterised, featured networks by abstraction

Muffy Calder and Alice Miller

Department of Computing Science
University of Glasgow
Glasgow, Scotland.
muffy,alice@dcs.gla.ac.uk

Abstract. Model-checking is a very effective, formal technique for reasoning about systems. A common limitation with model-checking is how to relate an individual results to the general case, i.e. does a result for a system of a given size scale to a system of any size? This question cannot be answered by model-checking alone, because in general, it is undecidable. However, we have developed techniques to solve the problem for some cases. In this paper we consider the application domain of reasoning about peer to peer, featured networks by model-checking. We have already shown [5] that under certain circumstances, we can reason about the general case using a combination of abstraction and model-checking. The method relies on dividing the set of components into two distinct subsets: observable concrete components, which may have features, and abstract components, which have no features. In this paper we extend the method to allow featured abstract components. This means that the abstract components are no longer isomorphic and so the new technique is considerably more expressive than the original. We specify criteria which the features must fulfill and show that our new approach is sound.

1 Introduction

We consider the problem of reasoning about the behaviour of peer to peer, featured networks. Examples of such networks include email and telephony services. In such networks, each component may have a number of features: a structuring mechanism for additional functionality. Features may fundamentally affect basic behaviour (in different ways) and so components with features are not, in general, isomorphic. In many cases, the features associated with one component can affect the behaviour of other (possibly featured) components. In order to detect this situation (known as feature interaction [2]), we require to verify temporal properties of the networks (known as property-based feature interaction detection). Since our networks consist of concurrent, communicating components, model-checking is the most suitable reasoning technique. It is rigorous, automated and widely adopted in industry and academe.

A common limitation when reasoning about networks by model-checking is how to relate an individual result, i.e. a result about a network of fixed size
and configuration, to the general case. Namely, does a result for a given system scale to a system of any size – can we leverage a general result from a specific one? This question cannot be answered by model-checking alone because it is an example of the parameterised model checking problem (PMCP) which is, in general, undecidable [1]. But in some subclasses the PMCP may be decidable. In this paper we propose such a subclass and present an abstraction technique which allows us to infer results about families of peer to peer networks from results obtained by model-checking a network of a given size. The novelty is that we consider networks consisting of any number of featured components, that is, we abstract over non-isomorphic components. Thus the new technique is considerably more expressive than the original.

Our results build upon our earlier work [5] where we prove, by abstraction and model checking, that certain classes of properties about featured components can hold for networks of arbitrary size. Our method relies on being able to divide the set of components into two distinct subsets: concrete components, which are observable and may have features, and abstract components, which have no features. We developed three distinct classes of model, for a given network size $N$ and $m$ concrete components. $M_N$ is a given concrete model, for a network of size $N$. $M_N^{m}$ is the data abstract model derived from $M_N$ by abstracting data values into equivalence classes; the abstraction depends on $m$. $M_N^{abs}$ is the abstract model derived from $M_N$ in which observable (abstract) component behaviour is abstracted. There is a simulation relation between the three: $M_N \succeq M_N^{m} \succeq M_N^{abs}$. Hence, when a property holds for the abstract model, we can infer that it holds for the appropriate concrete model. Namely, for an appropriate formula $\phi$, $M_N^{abs} \models \phi \Rightarrow M_N \models \phi$.

The key contribution of this paper is to extend our method to allow featured abstract components, i.e. the components in the abstract set are not isomorphic. We present, in some detail, how this affects the construction of $M_N^{m}$, and we specify criteria which the features must fulfill.

Our example network is a telecoms network with peer to peer, asynchronous, communication and features such as call forwarding, ring back when free, etc. The network specification is low level, i.e. close to operational code, and is specified in the language Promela [12]. This language is very similar to a concurrent programming language, and includes variable assignments, dynamic channels, and asynchronous communication. We explain our results both in terms of the network specifications using Promela, and the underlying (mathematical) models. Thus this paper demonstrates the application of a theoretical result to a non-trivial example.

In the next section we define model checking in our context and our notation for parameterised, featured networks. Following that, we define the GC form, a guarded command form for Promela. In section 4 we outline our technique of verification by data and behavioural abstraction. In section 5 we give an overview of the example system, the features, and a sample property. In section 6 we describe the abstract model $M_N^{m}$ for the example system, outlining the
changes to the feature handler function in some detail. In section 7 we discuss
the results and related work. We conclude in section 8.

2 Model-checking parameterised, featured networks

Model checking is a technique for verifying finite state systems. Systems are
specified using a modelling language and the model – or Kripke structure [6]–
associated with this specification is checked to verify given temporal properties.
We employ the modelling language Promela and its bespoke model-checker SPIN
[12] to check LTL (linear temporal logic) properties. LTL has temporal operators
$\square$ (always), $\diamond$ (eventually), $X$ (next) and $U$ (weak until), the only path operator
is (implicit) universal quantification.

SPIN translates each (concurrent) component defined in the Promela speci-
fication into a finite automaton and then computes the asynchronous interleaving
product of these automata to obtain the global behaviour of the concurrent sys-
ystem. This interleaving product is essentially a Kripke structure describing the
behaviour of the system. It is this Kripke structure to which we refer when we
talk about the model of our system.

Definition 1. Let $AP$ be a set of atomic propositions. A Kripke structure over
$AP$ is a tuple $M = (S, S_0, R, L)$ where $S \subseteq S$ is a finite set of states, $S_0$ is the
set of initial states, $R \subseteq S \times S$ is a transition relation and $L : S \to 2^AP$ is a
function that labels each state with the set of atomic propositions true in that
state.

For a given model $M$, and temporal property $\phi$, model checking allows us
to verify whether or not $M \models \phi$. We wish to verify families of (finite) parameter-
ised, featured networks. The networks are parameterised both in their number
of components and the nature of each component. That is, each component has
the same basic behaviour, with a number (possibly zero) of additional features.
For example, in the telecommns system, a component may have features which
include “forward when busy”, or “forward at all times”, or “block” calls from an-
other component. Featured components are not isomorphic, but they do exhibit
isomorphic, basic behaviour.

Not only does each component in a network have a set of associated features,
but the features themselves are parameterised by components. For example if
feature $f$ is “Component[0] forwards calls to Component[3]”, then $f$ is said to
be indexed by 0 and 3. We introduce the following notation.

Definition 2. For any feature $f$, we say that $f$ is indexed by $I_f = \{i_0, \ldots, i_{r-1}\}$
if the feature relates to Component[$i_0$], …, Component[$i_{r-1}$]. Similarly we say
that a property $\phi$ is indexed by a the set $I_\phi$ where $I_\phi$ is the set of component
identifiers, or indices, associated with $\phi$. For a (possibly empty) set of features
$F = \{f_0, \ldots, f_{s-1}\}$ and property $\phi$, we define the complete index set $I$ of $\{\phi\} \cup F$,
to be $I_{f_0} \cup \ldots \cup I_{f_{s-1}} \cup I_\phi$. 
Our problem is to show that, for a given feature set $F$, indexed property $\phi$, and models $M_N = M(p_0[p_1][p_2] \cdots [p_{N-1}])$ of a system of $N$ concurrent instantiations of a parameterised component $p$, $M_N \models \phi$ for all $N \geq 1$. Note that the $p_0, p_1, \ldots, p_{N-1}$ are concurrent and communicating; they are not, in general, isomorphic. We assume asynchronous communication and all components can update shared variables.

3 Features

A feature in our context is additional functionality added to the main body of the code. In order to quantify the ways in which features can affect the main body (usually called basic behaviour) we introduce the guarded command form.

3.1 GC form

It is convenient to assume that our models are expressed using a guarded command language, and written as a list of statements of the form $g \rightarrow c$, where $g$ is a guard, and $c$ a command. Each step of the program (after variable declaration) involves non-deterministically executing a statement for which the corresponding guard is true. In some model checking tools (e.g. Mur $\varphi$ [9] and SMV [14]), models are specified directly in this form. Let us refer to this form of specification as GC form. Promela specifications (programs) consist of processes (proctype) representing individual parameterised components. However, it is simple to convert a Promela program into GC form by incorporating the control state (or program counter, $p\omega$) of each process within the guards, as illustrated by the following example. Consider a simple Promela specification describing two Client processes, passing a token between themselves:

```plaintext
mtype = {null,token};

proctype Client(chan chanin,chanout)
{mtype message=null;
 start:chanin?message; assert(message==token);
 chanout!message; message=null; goto start}

init{chan1!token;run Client(chan1,chan2);run Client(chan2,chan1)}
```

During the initialisation process the token is placed on channel chan1. Then two instantiations of the Client component are instantiated. In the first, the values of the channel names chanin and chanout (local variables) are chan1 and chan2 respectively, and in the second instantiation, the values of these channel names are reversed. If we let chanin1 and chanin2 denote the local chanin variables associated with the first and second instantiation of the Client process respectively (similarly for the chanout variables), the corresponding specification in GC form is:
chan chan1, chan2 = [1] of {mtype};
mtype = {null, token};
chan chanin0=chan1; chan chanout0=chan2;
chan chanin1=chan2; chan chanout1=chan1;
mtype message0, message1 = null;
short p_c0, p_c1 = -1; short p_c2 = 0; /* program counter of init proc */
do
  ::(p_c2 == 0) -> chan1!token; p_c2 = 1
  ::(p_c2 == 1) -> p_c0 = 0; p_c2 = 2
  ::(p_c2 == 2) -> p_c1 = 0; p_c2 = 3
  ::((p_c0 == 0) && (nempty(chanin0))) -> chanin0!message0; p_c0 = 1
  ::((p_c1 == 0) && (nempty(chanin1))) -> chanin1!message1; p_c1 = 1
  ::(p_c0 == 1) -> assert (mtype == token); p_c0 = 2
  ::(p_c1 == 1) -> assert (message1 == token); p_c1 = 2
  ::((p_c0 == 2) && (nfull(chanout0))) -> chanout0!message; p_c0 = 3
  ::((p_c1 == 2) && (nfull(chanout1))) -> chanout1!message; p_c1 = 3
  ::(p_c0 == 3) -> message0 = null; p_c0 = 0
  ::(p_c1 == 3) -> message1 = null; p_c1 = 0
od

Each statement in the body of an (original) process corresponds to one or more cases in the corresponding GC do loop. Each of these cases consists of the appropriate combinations of program counter variables and predicates over channel(s). The program counter variables associated with the processes have form \( p_{ci} \), where \( i \) denotes processes 0 to \( N-1 \) and \( N \) denotes the init process. The propositions \( nempty(chan) \) and \( nfull(chan) \) assert that the channel \( chan \) is not empty, and not full respectively. To avoid deadlock, the (list of) commands associated with a guard \( g \) must be executable providing \( g \) is true. Thus, for example, no command can involve a read/write from (to) a channel, unless the associated guard \( g \) includes the appropriate \( nempty \) or \( nfull \) proposition.

Under the assumption that any Promela model consists of a sequence of guarded, or unguarded atomic statements in which only the first command can block (via a read or write statement), we can convert any Promela program to GC form. Note that an unguarded atomic statement \( e \) in process \( i \), with associated program counter \( n \) will simply have the guard \( p_{ci} == n \). For reasoning purposes, we will assume that a model is expressed in GC form, although our Promela models are not actually written in this way.

### 3.2 Adding features to the telecomms example

In previous work [3] we have shown how features can be added to a Promela specification representing a telecommunications network using arrays to record which components subscribe to which features. At various points in a call, components make reference to a \texttt{feature lookup} function to check whether any features will affect their current behaviour. In [4] a similar approach was used to add features to an email system. In this paper we assume that our Promela model is expressed
in GC form, and describe the implementation of features in this context. When component $i$ reaches a state at which the `feature_lookup` function would have been called, a local variable, `feature_check` is set to `on`. Behaviour is modified (in a given state) only if `feature_check == on` and a relevant feature is active. All other guards must now include the proposition `feature_check! = on`.

To illustrate this process, we give below the relevant statements in the GC form of the Promela specification of a basic telecommunications model, for a call forwarding unconditional (CFU) feature – in which every incoming call to a number is forwarded to another number – and an outward dial screening feature (ODS) – in which a user is not permitted to dial a specified number. We give here generic forms of the statements (in terms of arrays of variables indexed by a process $i$). If there are $N$ components, then the appropriate instantiations of these statements would be added for all $0 \leq i \leq N - 1$. The variables $i$, $st[i]$, `partnerid[i]` and `partner[i]` denote the process index, the local call state (eg. diall, calling), the index of the current partner, and the channel associated with the current partner, of component $i$ at this point. For a feature such as CFU which, if selected for a given component $i$, is not just switched on but set to another component index (the component to which calls are forwarded to), the value $D$ denotes some default value (usually set to $N$) indicating that no feature of this type is valid.

The two cases associated with CFU, derived directly from the non-GC form, are

$$
:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\n
We also need to switch `feature_check[i]` to off if `(feature_check[i] == on)` is true but all of the guards are false. We do this as follows: Suppose that $(g_i,0|g_i,1|\ldots|g_i, k-1)$ are subguards such that the guards containing the proposition `(feature_check[i] == on)` all have the form `(feature_check[i] == on)&&g_{i,j}` for some $0 \leq j \leq k - 1$. Define the propositional formula `not_active` to be \((g_i,0|g_i,1|\ldots|g_i, k-1)\) and add one further statement:

$$
:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\n
4 Verification by abstraction

As mentioned in section 1, in general, it is not possible to prove $\mathcal{M}_N \models \phi$, for all $N \geq 1$, by model-checking. In our abstraction approach [5], to verify a property $\phi$ in the presence of a set of features $F$, we divide the components into two distinct sets: those whose identifiers belong to the complete index set of $\{\phi\} \cup F$,
(concrete components), and those whose do not (abstract components). When there are $m$ concrete components we can infer properties about $\mathcal{M}_N$, for any $N$, from properties of a model which does not depend on $N$, i.e. from properties of the model $\mathcal{M}^{m}_N$. We prove the latter properties by model-checking. Since these (latter) properties pertain to $\mathcal{M}^{m}_N$ they cannot refer to any abstract component explicitly.

The abstraction approach for a peer to peer network, with $N = 5$ and $m = 3$, is illustrated by Figure 1. In the specification of $\mathcal{M}^{m}_N$, i.e. in the Promela specification of the concurrent components, the two abstract components $c_3$ and $c_4$ are replaced by one new component $Abstract_3$. All components in both network specifications communicate peer to peer. Properties could refer to aspects of the behaviour of the components $c_0$, $c_1$, and $c_2$, but not $c_3$ and $c_4$, as these are abstract components.

![Figure 1. Abstraction technique when $N = 5$ and $m = 3$](image)

The abstraction approach is sound if we can demonstrate a simulation relation between the two models, and the formula $\phi$ is indexed appropriately (i.e. indices are less than $m$).

We rely on the standard result that simulation preserves LTL properties [6], as given below.

**Definition 3.** Given two structures $\mathcal{M}$ and $\mathcal{M}'$ with $AP \supseteq AP'$, a relation $H \subseteq S \times S'$ is a simulation relation between $\mathcal{M}$ and $\mathcal{M}'$ if and only if for all $s$ and $s'$, if $H(s, s')$ then

1. $L(s) \cap AP' = L'(s')$
2. For every state $s_1$ such that $R(s, s_1)$, there is a state $s'_1$ with the property that $R'(s'_1, s'_1)$ and $H(s_1, s'_1)$.

If a simulation relation exists between structures $\mathcal{M}$ and $\mathcal{M}'$ we say that $\mathcal{M}'$ simulates $\mathcal{M}$ and denote this by $\mathcal{M} \preceq \mathcal{M}'$.

**Lemma 1.** Suppose that $\mathcal{M} \preceq \mathcal{M}'$. Then for every LTL formula $\phi$ with atomic propositions in $AP'$, $M' \models \phi$ implies $M \models \phi$. 
8

In our approach, to simplify the reasoning process, we introduce a third, intermediate model \( M_N^m \) such that \( M_N \preceq M_N^m \preceq M_{abs}^m \). In the next section we describe the general requirements of the models \( M_N, M_N^m \) and \( M_{abs}^m \) and indicate where our approach must be extended to allow featured abstract components.

4.1 \( M_N, M_N^m \) and \( M_{abs}^m \)

To show that a logical property \( \phi \) holds in the presence of a feature set \( F \) the network models are constructed in three steps.

1. Suppose that the concrete model is \( M_N = \mathcal{M}(p_0||p_1||\ldots||p_{m-1}) \).

Without loss of generality, let us assume that the complete index set of \( \{\phi\} \cup F \subseteq \{0,1,\ldots,m-1\} \). We call components \( p_0||p_1||\ldots||p_{m-1} \) concrete components and the remaining \( N - m \) components, abstract components.

(\text{Note that this implies that no feature belonging to a concrete component is indexed by an abstract component.})

2. Define a set of data abstraction functions [8]. These are surjections, based on \( m \), which map variables to reduced domains. For example, for any local variable associated with an abstract component, the surjection is the trivial surjection \( \overline{h_0} \) which maps every element of the associated domain to itself (identity). For any global variable, or local variable associated with a concrete component whose domain \( D \) consists of the set of component indices the surjection is \( \overline{h_1} \) where

\[
h_1(x) = \begin{cases} 
  x & \text{if } x < m, \\
  m & \text{otherwise}
\end{cases}
\]

for any \( x \in D \).

This approach is extended to the other variable domains in the obvious way, as described in [5]. Using the function \( \overline{h} \) to denote reducing all data in a component, the result is the data abstract model

\[
M_N^m = \mathcal{M}(\overline{h}(p_0)||\overline{h}(p_{m-1})||\overline{h}(p_m)||\ldots||\overline{h}(p_{N-1})).
\]

3. Define an Abstract\(_m\) component which encapsulates the (observable) behaviour of the \( (N - m) \) abstract components, and modify the concrete components to reflect the fact that Abstract\(_m\) replaces the abstract components.

This means that communication between concrete and abstract components must be encapsulated appropriately. Communication amongst concrete components is as for \( M_N \). We use the notation \( \overline{p}_i \) to denote a concrete component which has been data reduced (as described above, in step 2) and then altered so as to communicate with Abstract\(_m\) instead of actual abstract components.

In [5] the definition of \( \overline{p}_i \) was fairly straightforward. However, we must now amend the \( \overline{p}_i \) to allow the possibility of there being features present at the partner component, when a \( \overline{p}_i \) is involved in a call with an abstract component. The definition of Abstract\(_m\) allows a call to be initiated from an
abstract component to a concrete component at any point at which the concrete component is not already engaged in a call. We will discuss the \( p_i \) in detail in section 6.

The final result of this step is the abstract model

\[
M_{ab}^m = M(p_1 || \ldots || p_{m-1} || Abstract_m).
\]

In the next section, we outline the relevant aspects of the example system, and how they are specified in Promela. In section 6 we outline how \( M_{ab}^m \) is constructed for the example system.

## 5 Basic components and features

Call control in a telecoms network essentially involves call setup: establish a connection between two components, and call teardown: tear down a connection between two components. We assume two party calls: connections are between two components. One is the originating party and the other the terminating party. We assume asymmetric call control, thus the originator controls call teardown. Basic call control is achieved by two components engaging in the appropriate protocols. Essentially, each component moves through a set of states, in response to events which may be internal or external (asynchronous) communications from the other component. External events may be user-initiated, such as handset on or off hook, or control signals to devices and lines such as ringing tone or line engaged. We do not enumerate the states or give details of the protocols here (see [1]) but indicate that communication is crucial to call control and coordination of components. Communication between components takes place via channels (asynchronously), and there is a channel associated with each component. Channels are in effect (global) shared variables, and they contain references to other channels. Thus our communication topology is dynamic.

### 5.1 The Features

The feature set includes the following. To make the descriptions more intuitive, we refer to components as Users.

- **CFU – call forward unconditional** Assume that User\( [j] \) forwards to User\( [k] \). If User\( [i] \) rings User\( [j] \) then a connection between User\( [i] \) and User\( [k] \) will be attempted before User\( [k] \) hangs up.
- **CFB – call forward when busy** Assume that User\( [j] \) forwards to User\( [k] \). If User\( [i] \) calls User\( [j] \) when User\( [j] \) is busy then a connection between User\( [i] \) and User\( [k] \) will be attempted before User\( [k] \) hangs up.
- **OCS – originating call screening** Assume that User\( [i] \) has User\( [j] \) on its screening list, \( i \neq j \). No connection from User\( [i] \) to User\( [j] \) is possible.
- **ODS – originating dial screening** Assume that User\( [i] \) has User\( [j] \) on its screening list, \( i \neq j \). User\( [i] \) may not dial User\( [j] \).
TCS – terminating call screening Assume that User[i] has User[j] on its screening list, i ≠ j. No connection from User[j] to User[i] is possible.

RBWF – ring back when free Assume that User[i] has RBWF. If User[i] has requested a ringback to User[j], i ≠ j, (and not subsequently requested a ringback to another user) and subsequently User[i] is idle when User[i] and User[j] are both free (and they are still free when User[i] is no longer idle) then User[i] will receive the ringback tone.

OCO – originating calls only Assume that User[j] has OCO. No connection from User[i] to User[j] is possible.

TCO – terminating calls only Assume that User[j] has TCO. No connection from User[j] to User[i] is possible.

RBWF is known as a camp-on feature. In our specification this require setting a flag rybknmu[i] to indicate that a return call is required. Note that RBWF applies when the originator has the feature, and is initiated when the two parties both become free. The initiator of the return call, and hence call control (and usually charging) is the party which has the feature.

An example property to be checked is:

**TCO property** Assume that user j has TCO. No connection from user j to user i is possible. The LTL formula is \( (\neg \text{connect}[j].to[i] == 1) \)

Note that, for example, if there is a forwarding feature, then the TCO property would not hold, for some i and j.

6 The abstract model

The third stage of the abstraction approach outlined in section 4 consists of two steps: define Abstractm and modify the concrete components, including replacing communication between concrete and abstract components, by appropriate (explicit or implicit) communication between the concrete components and Abstractm. In this section we describe this process in terms of the GC specification. The process is illustrated in figure 2.

The definition of Abstractm is straightforward and follows the definition given previously [5]. One important aspect to note is the introduction of absid: the index associated with Abstractm (usually absid == m). We concentrate here on the modifications to the concrete components. We consider modification to communication first, and then other behaviour.

6.1 Modifications to generic statements involving communication

Communication between concrete components and Abstractm is implicit, i.e. there is no explicit communication channel associated with Abstractm. Thus each modified concrete component \( p_i \) no longer writes to (the associated channels of) any of the abstract components \( p_m, p_{m+1}, \ldots, p_N \), but there is a non-deterministic choice whenever such a write would have occurred as to whether
the associated channel is empty or full (thus, whether the write is enabled or not). Thus, in the GC form of the specification, we replace the (generic) statement, with associated program counter \( n \), of the form

\[
::(\text{feature_check}[i] == \text{off}) \&\& (\text{p_ci} == n) \&\& (\text{empty} (\text{partner}[i])))
\]

-> partner[i]!message; p_ci++

with the statements

\[
::(\text{feature_check}[i] == \text{off}) \&\& (\text{p_ci} == n) \&\& (\text{partnerid}[i] < \text{absid})
\]

&\& (empty (partner[i]))

-> partner[i]!message; p_ci++

::(\text{feature_check}[i] == \text{off}) \&\& (\text{p_ci} == n) \&\& (\text{partnerid}[i] == \text{absid})

-> skip; p_ci++

::(\text{feature_check}[i] == \text{off}) \&\& (\text{p_ci} == n) \&\& (\text{partnerid}[i] == \text{absid})

-> skip

Note that the second statement corresponds to a message having been sent, and the third statement to the write being blocked. There are \( m \) instantiations of these generic statements, for \( 0 \leq i \leq m - 1 \). Similarly, each \( i'_j \) no longer reads from (the associated channels of) any of the abstract components. Instead, they make a non-deterministic choice over the set of possible messages (if any) that could be present on such a channel.

6.2 Modifications to generic statements involving features

Let us consider generic statements of the form
Each subguard consists of a proposition regarding the value of the local variable \( st[i] \) (let us call this stateprop), a proposition concerning the value of a feature array element, which may be indexed by \( i \) and/or \( partnerid[i] \) (let us call this featureprop) and possibly a proposition concerning the value of some local or global variables. The list of commands command involves resetting the value of \( st[i] \) and possibly other local variable (associated with component \( i \)) and global variables indexed by \( i \) and/or \( partnerid[i] \).

Suppose that subguard has the form stateprop&&featureprop and that the proposition featureprop depends only on \( i \). This is true, for example in the ODS feature, for which subguard = ((\( st[i] == \) st_dial)&&(ODS[i] == on)). In this case, if no global variables indexed by \( partnerid[i] \) are to be updated, the entire generic statement remains unchanged. Otherwise, the generic statement is replaced by two statements. In the first, the proposition (partnerid[i] < absid) is added to subguard and command is unchanged. In the second, the proposition (partnerid[i] == absid) is added to subguard, and any update to global variables indexed by \( partnerid[i] \) is removed from command.

Consider now when subguard has the same form as in the previous case, but featureprop depends only on partnerid[i]. This case was not considered in [4] because it was assumed that no abstract component had a feature. Now abstract components may be featured. A feature statement that has this form is CFU:

\[
((\text{feature} \_\text{check}[i] == \text{on}) && (\text{st}[i] == \text{st_dial}) && (\text{CFU}[\text{partnerid}[i]] == D))
\]

Again we split the generic statement into two statements, depending on whether \( partnerid[i] \) is not equal to absid (in which case, command is left unchanged), or otherwise. If \( partnerid[i] \) is equal to absid, we must choose non-deterministically whether the feature is present in the abstract partner or not. This is achieved by setting a local variable, (forwarding_feature[i]) to on or off after the statement preceding the feature check. It is switched off once the feature has been instantiated. The feature is only instantiated if the variable is set to on. A non-deterministic choice is made as to the new value of any global variables indexed by \( i \) updated within command. Setting the value of the local variable to off immediately after the feature has been instantiated prevents the statement guard from remaining true and the identical feature being instantiated repeatedly. For example, observationally, a call that is continually forwarded within the abstract components is equivalent to one that is not forwarded at all within the abstract components. We provide the associated generic statements for the CFU feature. The original statements (for the unabstracted model) were given in section 3.2.

::((\text{feature} \_\text{check}[i] == \text{on}) && (\text{st}[i] == \text{st_dial})
 && (partnerid[i] == absid) && (\text{CFU}[\text{partnerid}[i]] == D)) ->
 partnerid[i] = \text{CFU}[\text{partnerid}[i]]; \text{partner} = \text{chan} \_\text{name}[\text{partnerid}[i]]
::((\text{feature} \_\text{check}[i] == \text{on}) && (\text{st}[i] == \text{st_dial})

&& (partnerid[i]==absid) && (forwarding_feature==on) ->
partnerid[i]=0; partneri=chan_name[partnerid[i]];
forwarding_feature=off
:: (feature_check[i]==on) && (st[i]==st_dial1) &&
     (partnerid[i]==absid) && (forwarding_feature==on) ->
partnerid[i]=1; partneri=chan_name[partnerid[i]];
forwarding_feature=off
...
:: (feature_check[i]==on) && (st[i]==st_dial1) &&
     (partnerid[i]==absid) && (forwarding_feature==on) ->
forwarding_feature=off

Note that the number of concrete components (i.e. m) determines the actual
number of statements to be added.

Suppose that subguard has the same form as in the previous case, but that
featureprop depends on partnerid[i] and i. Here featureprop will have the form
feature[i] == partnerid[i] or feature[partnerid[i]] == i. Because of our assump-
tion that no feature belonging to a concrete component can index an abstract
component, feature[i] == partnerid[i] can only be true if partnerid[i] != absid.
Thus we add the proposition partnerid[i] = absid to the start of subguard (to
avoid our program unnecessarily trying to access an array element with index
absid).

If featureprop has the form feature[partnerid[i]] == i we first split the state-
ment into two cases, when partnerid[i] != absid (in which case the statement
is otherwise unchanged, as before), and when partnerid[i] == absid. Note that
in our telecommunications example, the only feature that has this form is TCS.
Here we use a new local variable as before to determine whether the abstract
partner subscribes to this feature or not, and statements are split as in the CFU
case as described above.

The last cases to consider are when subguard contains a proposition involving
the value of global variables indexed by i and/or var, where var is some index
not equal to i. If the proposition only involves global variables indexed by var,
then this case is treated almost the same as the case above except that, if var ==
absid, a new statement is created for every possible value of the global variable
indexed by var. Often this global variable is actually a check on the status of
a channel, and so only the values “full” and “not full” need be considered, for
example. Our CFU feature comes into this class of features. In CFU, var is
actually equal to partnerid[i]. In the RBWF feature, the global variable is the
channel associated with a global variable (rgbknum[i]) which has been reset by
compact 1.

The only remaining class of feature is that for which the subguard contains
a proposition variables_prop, containing a global variable indexed by i. Being a
global variable, it could have been reset by an abstract process. This situation
does not arise with any of our selected features, and is excluded from our main
theorem (below).
6.3 Correctness of abstraction

We can now state the main theorem which demonstrates that our data and behavioural abstraction approach is sound for features which fulfill certain criteria.

**Theorem 1.** Let \( F \) be a set of features described by GC generic statements of the form:

\[
:: \text{(feature\_search[i])=on)\&\& (subguard)} \rightarrow \text{command}
\]

such that \text{subguard} consists of a proposition containing at most

- the value of \text{st[i]},
- the value of a feature array element, which may be indexed by \( i \) and/or \text{partnerid}[i]
- and a proposition concerning the value of some local or global variables which are not indexed by \( i \),

Let \( M_N = M(p_0||p_1||...||p_{N-1}) \) be the model of a system of \( N \) concurrent instantiations of a parameterised component \( p \), with feature set \( F \).

Then, for appropriate \( m \) and \( \phi \), if none of the components \( p_i \), for \( 0 \leq i \leq m - 1 \) have features that refer to any component \( p_j \), for \( m \leq j \leq N - 1 \), \( M_N \preceq M^{m}_{\text{as}} \preceq M^{m}_{\text{as}} \), and thus for all \( N \geq 1 \), \( M^{m}_{\text{as}} \Rightarrow M_N \models \phi \).

7 Discussion and Related Work

The abstraction approach is based on simulation: the concrete and abstract models are clearly not bisimilar. For example, the abstract model offers more non-determinism than any concrete model (e.g., a write to an abstract component in the latter is matched by a choice in the former). But, they do behave the same with respect to properties that refer only to concrete components, under the constraints of weak fairness. This holds regardless of whether or not the abstract components have features. An example illustrates the point. Assume that an abstract component forwards a call to another abstract component, or that an abstract component calls another abstract component which has RBWF. In both cases, the behaviour is not incorporated in \( M^{m}_{\text{as}} \). But, any property which allows us to make a judgement about the final connection, or the number of “legs” in a call, for example, involves references to abstract indices. Thus, we cannot actually observe the differences between the two models, and so the class of properties we can prove remains the same.

All the features presented here fulfill the criteria for Theorem 1, thus our method is effective for reasoning about them. One feature which does not fulfill the criteria is RWF — return when free. This feature is similar to RBWF but it resides at the terminal party. We have (manually) shown that abstraction is also sound for this case, but we have not yet identified how our criteria can be relaxed, in general. This is future work.

Abstraction techniques have been employed for reasoning about parameterised networks of identical processes, or systems consisting of a set of identical
processes plus a controller process, [10, 13, 7, 8]. We are not aware of results pertaining to non-isomorphic processes. We note that our feature “lookup” is not dissimilar to Plath and Ryan’s feature construct [15], and also Ryan’s related update operator [11].

8 Conclusions

We have developed a solution to a well known limitation of model-checking: how to infer results about the general case from a specific result.

Namely, we have extended our abstraction method [5] for reasoning about families of concurrent, communicating systems of featured components. The extension permits features to be associated with both concrete and abstract components. The abstraction approach is based on three distinct models: \( M_N \), \( M_N^m \) and \( M_{abs} \) and criteria which the features must fulfill. The final result is a finite model \( M_{abs}^m \) which incorporates all observable behaviour of the concrete model \( M_N \), for any \( N \) and given \( m \). We use model-checking to prove properties of this model, and then infer that the properties also hold for \( M_N \). So, from a user point of view, the only additional step required in order to infer a general result from a specific result (for a given \( N \)) is the construction of the model \( M_{abs}^m \), and we give a procedure for that.

We have illustrated our approach with a network specified in the language Promela, with a feature set which fulfills the criteria. Promela is very similar to a concurrent programming language, and includes variable assignments and dynamic channels. Thus this paper demonstrates the application of a theoretical result to a non-trivial example.

References


