A generic approach for the automatic verification of featured, parameterised systems

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

Abstract. A general technique is presented that allows property based feature analysis of systems consisting of an *arbitrary* number of components. Each component may have an *arbitrary* set of *safe* features. The components are defined in a *guarded command* form and the technique combines model checking and abstraction. Features must fulfill certain criteria in order to be *safe*, the criteria express constraints on the variables which occur in feature guards. The main result is a generalisation theorem which we apply to a well known example: the ubiquitous, featured telephone system.

1 Introduction

Model-checking is a popular technique for property-based feature interaction analysis of featured systems. But, there are limitations – we can only reason about systems of fixed, tractable size. This paper addresses the limitations by introducing a technique that allows the inference of properties of a system with arbitrary size and arbitrary features, from analysis of a system of fixed size and features. The constraint is that the features fulfill certain criteria which we call *safe*.

Property-based feature interaction analysis is based upon checking that temporal properties which characterise a given feature are preserved (or not) in the presence of another feature(s). The usual notation for this is the following, assuming S is a system updated with features f_1 and f_2: does $S + f_1 \models \phi$ imply $S + f_1 + f_2 \models \phi$?

We take a distributed, user oriented view of systems and therefore model a system as a set of concurrent, communicating components, each of which may subscribe to a number of features. We assume the paradigm of components communicating with each other (pairwise), either directly (e.g. telephony) or indirectly (e.g. email). Assuming p is a component with no features, p_{f_1} and p_{f_1} are components with features f_1 and f_2 resp., and $||$ is parallel composition, then an example of interaction analysis in this context is:

i) does $(p_{f_1}||p) \models \phi$ imply $(p_{f_1}||p_{f_2}) \models \phi$?

Of course a component can subscribe to more than one feature. For example, assuming that $p_{f_{1,f2}}$ is a component with features f_1 and f_2, then another example of feature interaction analysis is:

ii) does $(p_{f_1}) \models \phi$ imply $(p_{f_{1,f2}}) \models \phi$?

Now suppose that have proved i) or ii), what can we infer about the validity of ϕ when there are other components in the system? For example, what can we infer about

iii) $(p_{f_1}||p_{f_2}||p) \models \phi$, or

iv) $(p_{f_1}||p_{f_2}||p_{f_3}||p||p) \models \phi$?

More generally, when $p_2 \dots p_N$ and $p_3 \dots p_N$ are components each of which subscribe to arbitrary sets of features, what can we infer about

v) $\forall N.\ (p_{f_1}||p_{f_2}||p_3|| \dots ||p_N) \models \phi$, or

vi) $\forall N.\ (p_{f_{1,f2}}||p_2|| \dots ||p_N) \models \phi$?

v) and vi) are examples of the *parameterised model checking problem*, *PMCP*, which is of course undecidable [2]. However, for certain subclasses of systems (with a regular topology), a solution is possible using an invariant-based approach [17, 4, 14]. We propose such an approach and combine model checking with abstraction to allow inference of a general property like v) and vi) from a property of a fixed system such as the antecedents of i) and ii).

The approach relies on partitioning components into two distinct subsets: *concrete* components and *abstract* components. The former are the components involved in the fixed system analysis, i.e. the components in examples i) and ii). The abstract components are the remaining components in the systems of arbitrary size/features. For example, $p_3, \dots, p_N$ are the abstract components in v). It is important to note that both concrete and abstract components can have features, provided that the features are *safe*. The main contribution of this paper is to define *safe* features and prove that the parameterised model checking problem is solvable for the class of safe features. Previous work [8, 9] has only allowed abstract components to subscribe to a small set of specific features, here we define general criteria.

2 Background

Systems are specified using a modelling language and the model – or *Kripke structure* [11] – associated with this specification is checked to verify given temporal properties.

Definition 1 *Let AP be a set of atomic propositions. A Kripke structure over AP is a tuple $\mathcal{M} = (S, S_0, R, L)$ 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 $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.*

From here on we assume that $|S_0| = 1$, i.e. it is a single fixed state, S_0 say. We employ the modelling language Promela and its bespoke model-checker SPIN [16] to check LTL (linear temporal logic) properties. Simulations between Kripke structures preserve LTL properties.

Definition 2 *Given two Kripke 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 $R'(s', s'_1)$ and $H(s_1, s'_1)$.*

If $H(s_0, s'_0)$, we say that $\mathcal{M}'$ *simulates* $\mathcal{M}$ and write $\mathcal{M} \preceq \mathcal{M}'$.

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

Definition 3 *Let p be a component parameterised by a set of features. For a given N, let $\mathcal{M}_N$ be the model associated with N components, namely*

$$\mathcal{M}_N = \mathcal{M}_N(p_0||p_1||\ldots||p_{N-1}),$$

or

$$\mathcal{M}_N = \mathcal{M}_N(C||p_1||p_2||\ldots||p_{N-1}),$$

where the p_i denote instantiations of p and C denotes a distinguished context component. Note that the p_i are not, in general, isomorphic.

Definition 4 *Given $\mathcal{M}_N$ and $0 \leq m \leq N-1$, we refer to the components p_0, …, p_{m-1} as concrete components, and the remaining components as abstract components. All of the components may subscribe to different sets of features, but the features associated with the concrete components may not involve any abstract component.*

Definition 5 *When a component subscribes to a feature, we refer to that component as the feature host. When a component has the goal of communicating with another component, we refer to the latter as the former's partner.*

2.1 Solving PMCP by abstraction

Our approach to solving PMCP, in some cases, has been introduced in [9]. We give an overview here.

For a given system of size N with m concrete components, we derive $\mathcal{M}_N^m$, the *data abstract* model derived from $\mathcal{M}_N$ by abstracting data values into equivalence classes. The abstraction depends on m; essentially all values greater than m are made equivalent. $\mathcal{M}_{abs}^m$ is the (behaviour) *abstract* model derived from $\mathcal{M}_N^m$. $\mathcal{M}_{abs}^m$ consists of the concrete components, and one single new component, called abs which encapsulates (is an abstraction of) the observable behaviour of all the abstract components i.e. $\mathcal{M}_{abs}^m = \mathcal{M}(p_0||p_1||\ldots||p_{m-1}||abs)$. The communication to/from concrete components in $\mathcal{M}_{abs}^m$ may be slightly modified to take account of substitution of abs for the abstract components. The key step is to define abs such that there is a simulation relation between the three classes of models: $\mathcal{M}_N \preceq \mathcal{M}_N^m \preceq \mathcal{M}_{abs}^m$. Because simulation preserves LTL properties, we can conclude, for appropriate ϕ, that $\mathcal{M}(p_0||p_1||\ldots||p_{m-1}||abs) \models \phi$ implies $\forall N.\ \mathcal{M}(p_0||\ldots||p_{m-1}||p_m||\ldots||p_{N-1}) \models \phi$.

In this paper we extend the approach to include abstract components which subscribe to *safe* features. The safety criteria express constraints on the variables which occur in feature guards (e.g. local or global) and the form of feature actions (e.g. read or update). The criteria therefore impose a taxonomy on features and we use this taxonomy to classify features as safe or unsafe with respect to our approach to solving PMCP.

The main result is a theorem which states that if we only consider *safe* features, then our technique for feature interaction detection is applicable. This result is useful because it does not depend on the intention of a feature, nor on the application domain, it depends only the checkable form of a feature. It is important to note that our feature classification is orthogonal to any classification of feature interactions, our classification indicates whether or not we can apply our general interaction detection technique.

2.2 Guarded Command form and open symmetric features

While we specify a system in a distributed way, i.e. user components with features, we require to assume a different form for reasoning. Namely, we assume the *guarded command*, GC, form which consists of one, global loop over a choice of statements of the form $guard \rightarrow command$. Guards will be over-lapping when the system behaviour is non-deterministic. The precise definition of the form depends upon the specification language; we have defined it for Promela. We note that in some model checking tools (e.g. Murϕ [13] and SMV [18]), models are specified directly in this form. The conversion of a distributed specification into GC form is relatively simple, it is essentially the form one obtains after applying an expansion theorem (parallelism is converted to choice), modulo some additional "housekeeping" variables such as local process counters. We therefore omit the details here but illustrate the form by way of an extended Promela example in section 8.

We restrict our attention to features whose behaviour does not depend upon the behaviour of any particular component, that is, the features are generic with respect to components. Nearly all features in practical applications fall into this category.

Definition 6 *Let $\mathcal{M}_N$ be a model associated with N components, expressed in GC form. Features which result in statements that conform to the following constraints are called open symmetric. Let i be a variable over indices of the components. Then guards may not contain propositions of the form $var_i == val$, for a fixed value val, and for any assignment of a component index j say to i, there must exist equivalent statements (with identical guards) in which i can be set to any component index. Similarly for channel indices.*

For example, let $var1_i$ and $var2_i$ be variables associated with a component. A statement of the form $(var1_i == i) \rightarrow var2_i = 1$ is only allowed if there are also statements: $(var1_i == i) \rightarrow var2_i = 0$, $(var1_i == i) \rightarrow var2_i = 2$, $(var1_i == i) \rightarrow var2i = 3$, ... $(var1_i == i) \rightarrow var2_i = N-1$.

In the remainder of this paper we assume that all features are open symmetric.

3 Safe features

In our abstraction approach to PMCP, we define a safe feature as one which can be subscribed to by an abstract component without affecting the validity of any property ϕ which is suitably indexed (by concrete components).

Definition 7 *Given a parameterised component p, $\mathcal{M}_N$=$\mathcal{M}(p_0||p_1||\ldots p_m||\ldots||p_{N-1})$ such that $0 \leq m \leq N-1$, and formula ϕ indexed by elements of $\{0,1,\ldots,m-1\}$. A set of features F is safe iff*

$$\mathcal{M}(p_0||p_1||\ldots p_{m-1}||abs) \models \phi$$

implies

$$\forall N.\ \ (p_0||\ldots||p_{m-1}||p_m||\ldots||p_{N-1}) \models \phi$$

where $p_0,\ldots,p_{m-1}$ subscribe to arbitrary features and $p_m,\ldots,p_{N-1}$ subscribe only to features in F.

We can easily extend this definition to systems which include a context component.

4 Classifying features

Informally, many features can be divided into three broad categories according to whether they are managed by the feature host, the partner of the feature host, or by a third party. Examples of these categories are Ring Back When Free (host) and Call Forward Busy (partner) from telephony, and mail filtering (third party) from email. The main result of this paper, given in section 7, is that these features *are all safe*. A feature which does not fall into one of these categories is called multi-owned and it is not safe. An example feature in this category is Return When Free. There is a simple syntactic test for (lack) of safety, assuming a common form for feature specifications. We will prove this result in section 7. In order to do so, we first give some more detail on classifying features.

Guards refer to a number of propositions, each of which refers to a wide variety of variables. Variables are either *local* (to a component), *global*, or *feature related*. The last category includes variables which denote whether or not a feature is subscribed to. Local variables include the local program counters (not to be confused with Promela built-in local program counters).

Assuming guards are in conjunctive normal form, a guard that can trigger a feature has the form

$$(feature\_prop)\&\&(localprop)\&\&(varprop)$$

where $(feature\_prop)$ is a proposition which checks whether or not a feature is subscribed to by a component, $localprop$ checks properties of local variables, and $varprop$ checks properties of global variables.

5 GC form

In this section we describe additional assumptions made about our specifications in *GC* form, and describe how features are implemented in this form.

5.1 Additional assumptions

Local variables associated with each component are of three types: p-*variables*, the values of which are component indices drawn from the set $V = \{D, 0, 1, \dots, m\}$ (or $\{D, 1, \dots, m\}$ if there is an environment process); c-*variables*, the values of which are channel indices; and *standard variables* (variables which are not p-variables or c-variables) of finite type. The value D is a default value which is chosen to take the value of the smallest positive value not equal to any component index. (In the unabstracted case this is N.)

We assume that each component has, amongst its local variables, the variable p_c, denoting its program counter. In addition all processes (except possibly the context process, when one exists) have p-variables $selfid$ denoting the component index, and $partnerid$ denoting its current partner. No operations on $selfid$ are permitted; the value of $partnerid$ is either set by reading from a channel, or via non-deterministic choice. No other operations on the $partnerid$ variables, apart from resetting to D, are permitted.

The context process has index $context\_id$ and may have p-variables. For example, in the email model, the $Network\_Mailer$ process has p-variables $receiver$ and $sender$ which take values according to the destination and source of the current message being distributed. The values of these variables are set by reading messages from the $Network$ channel, and the only operation permissible on these variables is a reset to the default value D.

We assume that the only global variables present in our system are channels and those pertaining to arrays indexed by elements of V. Thus all global variables are channels or have the form $glob\_var[i]$, for some $i \in V$. For any global variable $glob\_var[i]$ we assume that there exist global variables $glob\_var[j]$ for all $j \in V$. We also assume no component with index i can carry out any operation on a global variable $glob\_var[j]$, for any $j \in V$, unless $glob\_var$ is a *feature-flag array* (see section 5.2 below) and the operation occurs within a feature statement. We assume that all channels are finite buffers, and there is one channel associated with each component.

5.2 Features in GC form

Feature statements have the form

$$(feature\_prop)\&\&(localprop)\&\&(varprop) \rightarrow command$$

Depending on the form of $feature\_prop$ it is possible develop a feature categorisation. We will subsequently use our categorisation together with an analysis of the form of var_prop to determine which features can be considered *safe* with respect to our abstraction technique.

Let us first consider $feature\_prop$. Now for any feature statement, $feature\_prop$ either has the form $feature\_name[myvar_1] == myvar_2$ or has the form $feature\_name[myvar_1]! = D$, where $feature\_name$ is a feature array, $myvar_1$ and $myvar_2$ are p-variables, and either:

1. $myvar_1$ is one of the p-variables $selfid$ or $partnerid$, and $myvar_2$ is $partnerid$ if $myvar_1$ is $selfid$, and $selfid$ if $myvar_1$ is $partnerid$, or

2. neither $myvar_1$ or $myvar_2$ belong to $\{selfid, partnerid\}$.

Note that $feature\_name[myvar_1]! = D$ is to be read as "$feature\_name[myvar_1]$ does not equal the default value D". This is true if the component with id $selfid$ subscribes to the feature, regardless of the particular value of $feature\_name[myvar_1]$.

Many features can be divided into three broad categories according to whether they are managed by the feature host, the partner of the feature host, or by a third party. They are therefore described as: *host owned*, *partner owned* or *third party owned*. These classes directly correspond to whether, in all feature statements, within all $feature\_prop$ guards, $myvar1$ is $selfid$, $partnerid$, or some other p-variable. Examples of the first category are ODS (outgoing calls only, telephone) and encryption (email). An example of *partner owned* feature is CFU (forward all calls unconditionally, telephone). In our email model, many of the features are handled by the $Network\_Mailer$ process, and so none of our email features are *partner owned*. Examples of *third party owned* features include (email) filtering and forwarding which are owned by a $Client$ process, but managed by the $Network\_Mailer$ process.

Note that the only one of our example features that can not be described in these terms is RWF (return when free). This feature sometimes triggers a change in behaviour because the host component has the feature (if the component has the feature and another component has requested a ringback by setting a *feature-flag array* element associated with the host component), and sometimes because the partner component has the feature (when a request is made by the host component for a ringback by the partner component by setting a *feature-flag array* element associated with the partner element). As such, we describe RWF as *multi-owned*.

Features can be further classified as to whether they are singly or doubly indexed. Singly indexed features are those for whom all associated $feature\_prop$ guards have the form

$$feature\_name[myvar1]! = D,$$

and doubly indexed features are those for whom all associated $feature\_prop$ guards have the form

$$feature\_name[myvar1] == myvar2.$$

For example OCO (outgoing calls only) is indexed by $selfid$ only, and so is said to be *host owned, single index*. (Clearly, in some contexts, features may be indexed by more than two components. However, as none of our example features have more than two indices, we will limit ourself to at most double index.) Accordingly we classify all of the telephone and email features below. The remaining acronyms are TCO (terminating calls only), TCS (terminating call screening), CFB (call forward on busy) and RBWF (ring back when free). Full descriptions of the features may be found in [6] and [8] respectively.

host owned, single index (HS)	TCO, RBWF
host owned, double index (HD)	OCS, ODS
partner owned, single index (PS)	CFU, CFB, OCO
partner owned, double index (PD)	TCS
third party owned, single index (TS)	
third party owned, double index (TD)	
multi-owned, single index (MS)	RWF

Table 1: Classification of features in the telephone example

host owned, single index (HS)	encryption, decryption, autoresponse
host owned, double index (HD)	
partner owned, single index (PS)	
partner owned, double index (PD)	
third party owned, single index (TS)	forwarding, mailhost, remail
third party owned, double index (TD)	filtering

Table 2: Classification of features in the email example

Recall that statements implementing features have the form

$$(feature\_prop)\&\&(localprop)\&\&(varprop) \rightarrow command.$$

As discussed earlier, $feature\_prop$ is a proposition determining whether the feature is currently applicable and $localprop$ and $varprop$ are conjunctions of propositions concerning local variables of a component, and global (indexed) variables respectively.

The substatements $varprop$ may be empty, refer to global variables that are indexed only by j say, where $j \neq selfid$, or contain propositions that refer to global variables indexed by $selfid$. As such we refer to the type of $varprop$ as either *empty*, *external* or *internal*.

6 Constructing the abstract model $\mathcal{M}^m_{abs}$

In this section we describe changes that are made to an original model representing a finite number of components, $\mathcal{M}_N$, to construct an abstracted model $\mathcal{M}^m_{abs}$. Every modification is made with the specific aim of ensuring that every transition in $\mathcal{M}_N$ which does not involve any internal transitions of abstract components, is reflected in the abstract model. Recall, we must ensure a simulation relation between $\mathcal{M}^m_{abs}$ and $\mathcal{M}_N$. There are three important aspects that must be considered: communication between concrete and abstract components, the form of the *abstract* component abs, which represents all abstract components, and the implementation of features. The first two of these have been discussed fully in previous work [7, 8] and are illustrated in the extended example given in section 8. Thus we only consider here how feature statements in concrete components must be modified.

6.1 Constructing features in the abstracted model

Let us first note that there are now $m+1$ processes (m concrete and 1 abstract process). The *Abstract* process abs is assigned index m, also referred to as $Absid$. Note that the default value D of all p-variables is now $m+1$. (Additional values $m+2$, $m+3$ etc. are sometimes used to represent a "callback_number" (telephone) or a pseudonym (email) within feature implementations, but this detail is not relevant here.)

In this section we show how statements involving features in the unabstracted model are replaced with a set of equivalent statements in the abstracted model. The nature of these equivalent statements is determined by the class of feature, and the particular type of $varprop$, namely empty, external or internal, see section 5.2 above. We consider the possible combinations of feature class and $varprop$ type separately. Note that we provide no examples here, but illustrate the methods in the full, extended example given in section 8.

(a) HS/empty
These statements have the form

$$(feature\_name[selfid]! = D)\&\&(localprop) \rightarrow command$$

In this case, if no global variables indexed by $partnerid$ (i.e. flag-array elements) are to be updated, the entire statement remains unchanged. Otherwise, the statement is replaced by two statements. In the first, the proposition $(partnerid! = Absid)$ is added to $subguard$ and $command$ is unchanged. In the second, the proposition $(partnerid == Absid)$ is added to $subguard$, and any update to global variables indexed by $partnerid$ is removed from $command$. The reason for this is that updates to global variables indexed by abstract components do not result in transitions in the data abstract model. Therefore they are not necessary in the abstract model.

(b) PS/empty
These statements have the form

$$(feature\_name[partnerid]! = D)\&\&(localprop) \rightarrow command$$

We note that we did not consider any examples of this case in our previous work since it was assumed that no abstract component had features. Now abstract components may be featured.

Again we split the generic statement into two statements, depending on whether $partnerid$ is not equal to $Absid$ (in which case, $command$ is left unchanged), or otherwise. If $partnerid$ is equal to $Absid$, we must choose non-deterministically whether the feature is present in the abstract partner. This is achieved by non-deterministically setting a local variable to on or off in the statement preceding the feature statement. For example, if the feature is a forwarding feature, let the local variable be "$forwarding\_on$". The feature is only instantiated if the local variable is set to on. Resetting the value of the local variable to off immediately after the feature has been applied prevents the statement guard from remaining true and the identical feature being *instantiated* repeatedly. A non-deterministic choice is made as to the new value of any global variables indexed by $selfid$ updated within $command$. . If the feature is a forwarding feature for example, the value of $partnerid$ will be set to either a concrete component id or another abstract process (i.e. fixed at $Absid$) via individual choices.

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

HD/empty, HD/external and HD/internal

Here $featureprop$ has the form $feature\_name[selfid] == partnerid$. Because of our assumption that no feature belonging to a concrete component can index an abstract component (see section 2), $feature\_name[selfid] == partnerid$ can only be true if $partnerid! = Absid$. Thus we add the proposition $partnerid! = Absid$ to the start of $subguard$ (to avoid our program unnecessarily trying to access an array element with index $Absid$).

PD/empty

These statements have the form

$$(feature\_name[partnerid] == selfid)\&\&(localprop) \rightarrow command$$

We first split the statement into two cases, when $partnerid! = Absid$ (in which case the statement is otherwise unchanged, as before), and when $partnerid == Absid$. Here we again use a local variable to determine whether the abstract partner subscribes to this feature or not, and statements are split as described above.

TS/empty and TD/empty

These statements have the form

$$(feature\_name[otherid_1]! = D)\&\&(localprop) \rightarrow command$$

or

$$(feature\_name[otherid_1] == otherid_2)\&\&(localprop) \rightarrow command$$

where $otherid_1$ and $otherid_2$ are not $selfid$ or $partnerid$.

In either case feature statements should again be split into two cases, this time according to whether $otherid_1! = Absid$ (in which case the statement is otherwise unchanged) or $otherid_1 == Absid$. In the latter case, a non-deterministic choice is made as to whether the feature is active or not, and statements are split as for the case above.

The last cases to consider are when $varprop$ contains propositions involving the value of global variables indexed by i, or contains propositions involving other global variables (not indexed by i).

Any class/external If $varprop$ only contains propositions involving global variables which are not indexed by $selfid$, then the variables either refer to channels or to global variables indexed by j say, where $j \neq selfid$. This case is treated almost the same as the case above. Suppose that a proposition refers to a channel $chan$, or a global variable indexed by j, $glob\_var[j]$. If $chan$ refers to an abstract channel (i.e. is the channel associated with an abstract process), or $j == Absid$, a new statement is created for every possible value of the status/contents of $chan$, or value of $global\_var[j]$. If the global variable is a channel, the proposition may simply involve a check on the status of the channel, and so only the values "full" and "not full" need be considered.

Any class/internal
The last class of feature is that for which the $varprop$ contains a proposition $variables\_prop$, containing a global variable indexed by $selfid$, $glob\_var[selfid]$ say. We may assume that this variable may be changed by components with ids not equal to $selfid$ (otherwise the variable would have been declared as a local variable) and so $glob\_var$ represents a feature-flag array. As it could have been reset by an abstract process, we can not simulate the possibility of an abstract process resetting this variable *at any time*. Note that we can not simply use non-deterministic choice to decide whether the value of $glob\_var$ has been changed by the abstract process (presumably to $Absid$) because to do so would assume that at some point an existing, non-default value of $glob\_var[selfid]$ may have been overridden. This would imply an earlier transition which would not have been reflected in our simulated model. Note that we do allow non-deterministic choice over channel contents (when component with id $selfid$ is communicating with an abstract process), but only when communication has been established and such *overriding* is not possible. (Initial communication set up from abstract components is simulated using real messages sent to the appropriate concrete channels.) For this reason, in order to ensure that our abstraction is sound, we do not include any feature for which $varprop$ is internal.

7 Theoretical results

In this section we give our main result relating feature classification and soundness of our abstraction. First we give a lemma in which we use the notation of section 6.1.

Lemma 2 *Any feature that has at least one associated feature statement in which $varprop$ is internal, is a multi-owned feature.*

proof Let us call the feature f. A component i will only set the associated feature-flag array (FFA say) element with index j if component j subscribes to f. Thus, for any component, among the feature statements associated with f will be a statement for which $feature\_prop$ has the form $f[partnerid]! = D$. In order for the feature to be implemented, the owner of the feature must check the FFA element (associated with its index) to see if it has been requested. Therefore, for any component i, among the feature statements associated with f will be a statement for which $feature\_prop$ has the form $f[selfid]! = D$ (and varprop contains the proposition $FFA[selfid]! = D$. Thus f is multi-owned.

Lemma 2 leads to the following theorem:

Theorem 1 *A set of features is safe if it does not contain any multi-owned features.*

Sketch of proof A set of features F is safe if allowing abstract components to subscribe to features in F preserves the solution of PMCP by abstraction (see definition 7). The solution of PMCP by abstraction depends on establishing a simulation between the unabstracted and the abstracted model. When all the features from F do not have an internal *varprop*, then the simulation follows from the way we have shown, in section 6, how statements involving features in the unabstracted model are replaced by equivalent statements in the abstracted model. The only case where a replacement cannot be made is when *varprop* is internal. Thus by Lemma 2, a multi-owned feature is not safe. For all the other feature categories, there is a simulation and thus the features are safe.

In the next section we give an example of a system of feature-parameterised components taken from a user-oriented, distributed view of telephony. We discuss, in some detail, how two features are implemented in GC form and how the abstracted model is constructed, assuming $m = 2$ (three concrete components). Although we omit the bulk of the specification (which is available from our website), the discussion is necessarily quite technical. However, its inclusion allows our previously discussed methods to be illustrated.

8 An Example from telephony

We illustrate our abstraction technique via four example Promela programs, $GC\_form.p$, $Abs\_GC.p$, $featureAbsGC.p$ and $original.p$, all of which are available on our website [5]. We do not provide the entire programs here, but use extracts from them to illustrate our methods. In each case we consider a system of User processes,

$$User[0], User[1], User[2], \ldots User[n-1]$$

and check for interaction between features $CFU[0] = 1$ ($User[[0]$ forwards all incoming calls to $User[1]$) and $ODS[1] = 0$ ($User[1]$ is prohibited from dialling $User[0]$). The associated property is ϕ where ϕ is "if $User[2]$ calls $User[0]$ then a call will be attempted from $User[2]$ to $User[1]$".

The program $GC\_form.p$ is an unabstracted model representing four $User$ processes (so $n = 3$ in this case). We use a parameterised proctype definition to define the $User$ processes, and each is instantiated via a run statement passing the value of the process id ($selfid$) and the channel name associated with that process ($self$). As such our program could be described as being in "modular GC form". Actual GC form would be achieved by pasting four copies of the User proctype sequentially and annotating each local variable with the process id. However, we believe that it makes our explanations simpler to leave it in modular *GC form*. Note that this example differs from similar examples provided for previous work, in that the model is expressed in GC form. In fact, Promela models are much easier to read when not expressed in this form (GC form prevents the use of $goto$ statements and labels for example), but our abstraction approach is easier to implement if they are (and easier to formalise under the *assumption* that they are). The program $original.p$ allows for the same behaviour as $GC\_form.p$ and is provided for the interested reader, as reference. Note that $original.p$ is annotated with comments indicating the value of p_c (see below) at the corresponding statement in $GC\_form.p$. Similarly, $GC\_form.p$ is annotated with comments indicating the *label* (representing the call state, *idle*, *diall*, *calling* etc.) associated with the corresponding statement in $original.p$.

In $GC\_form.p$, propositions that are to be used within guards in statements implementing the CFU and ODS features are defined as:

```
#define feature_property1
((partnerid!=callback_number)&&(state==st_diall)&&
(CFU[partnerid]!=default1))
```

and

```
#define feature_property2
((state==st_diall)&&(ODS[selfid]==partnerid))
```

respectively.

Feature statements associated with these guards are:

```
::atomic{((position_prop)&&(feature_property1))->
  partnerid=CFU[partnerid];
  partner[selfid]=chan_name[partnerid]}
```

and

```
::atomic{((position_prop)&&(feature_property2))->
  state=st_unobt}
```

respectively. Note $chan\_name$ is an array whose ith entry contains the channel name associated with $User[i]$ and $position\_prop$ is a disjunction of atomic statements concerning the value of p_c, a local variable labelling the position of the $User$ process within the proctype definition. The value of $position\_prop$ is true at specific points in the code at which features may be triggered. If $position\_prop$ is true but none of the guards within the feature statements are true, the value of p_c is increased and the $User$ process progresses to its next statement. (Note that feature propositions and statements are provided for a suite of other features, but are commented out as they are not relevant for this verification.) In the notation used throughout the paper, feature statements have the form

$$(feature\_prop)\&\&(localprop)\&\&(varprop) \rightarrow command$$

In our CFU statement, $(feature\_prop)$ is $(CFU[partnerid]! = default1))$, $localprop$ is $((partnerid! = callback\_number)\&\&(state == st\_diall)\&\&(position\_prop))$, and $varprop$ is empty.

For example, consider the following fragment taken from $GC\_form.p$, associated with states in which the User process is in the $diall$ state, and selecting a partner to call:

```
::/*diall*/atomic{(p_c==4)->
  partner[selfid]=zero;partnerid=0;state=st_diall;
  dialled[selfid]=0;state=st_diall;p_c++}
::/*diall*/atomic{(p_c==4)->
  partner[selfid]=one;partnerid=1;state=st_diall;
  dialled[selfid]=1;p_c++}
```

```
::/*diall*/atomic{(p_c==4)->
  partner[selfid]=two;partnerid=2;state=st_diall;
  dialled[selfid]=2;p_c++}
::/*diall*/atomic{(p_c==4)->
 partner[selfid]=three;partnerid=3;state=st_diall;
 dialled[selfid]=3;p_c++}
::/*diall*/atomic{(p_c==4)->partnerid=callback_number;
  state=st_diall;p_c++}
::/*diall*/atomic{(p_c==4)->
    dev[selfid]=on;self?messchan,messbit;
    messchan=null;messbit=0;
    MYSTATE[selfid]=preidle; p_c=24}
::/*diall*/atomic{((p_c==6)&&(state==st_unobt))->
   state=on;partner[selfid]=null;partnerid=default1;
   dialled[selfid]=default1;
   MYSTATE[selfid]=unobtainable;p_c=16}
```

Note that the actual code provided on our website involves more choices. This is for state-space minimisation reasons, we only want to change the value of the $dialled$ and $MYSTATE$ variables for process 2, as these variables are only used for verification purposes and do not affect behaviour. This does not contravene open symmetry.

When the value of p_c is 4 either a partner is selected, or the handset is replaced and the call attempt abandoned. In the first case, p_c is incremented. The atomic proposition $(p\_c == 5)$ is a disjunct of $position\_prop$, and so the feature statements are checked to see whether any of the features should be triggered at this point. All relevant features (if any) are then implemented until none of the feature guards are true. At this point p_c is incremented, so that the relevant statement (not feature-related) when $p\_c == 6$ can be executed.

In $Abs\_GC.p$ we give a model representing 3 concrete processes and an abstract process $Abstract$. In this case, none of the abstract processes are assumed to have any features. As such, $Abs\_GC.p$ is derived from $GC\_form.p$ in two ways:

1. The feature propositions are modified to include the proposition $(partnerid! = Absid)$, where $partnerid$ is a variable containing the value of the id of current partner, and $Absid$ is the id of the $Abstract$ process. For example, the feature proposition associated with the call forwarding feature described above, is modified to

   ```
   #define feature_property1
   ((partnerid!=callback_number)&&(partnerid!=Absid)&&
    (state==st_diall)&&(CFU[partnerid]!=default1))
   ```

   This is because we have assumed that no feature at a concrete component involves an abstract component.

2. Any statement in the $User$ proctype that involves communication must be modified. If the current partner is the abstract process then a write to the partner's channel is replaced by a $skip$ (or empty action, no communication actually takes place). For example, consider the following fragment of code taken from $GC\_form.p$ associated with the *calling* state.

   ```
   ::/*calling*/atomic{((p_c==9)&&(state==st_call2)&&
   ```

```
 (!(partner[selfid] == self))&&(len(partner[selfid])==0))->
 state=on;partner[selfid]!self,0;self?messchan,messbit;
 self!partner[selfid],0;messchan=null;messbit=0;
 MYSTATE[selfid]=oalert;p_c=17}
```

This is replaced by the following statements in $Abs\_GC.p$ (note that *out_channel* is the name of the channel associated with the abstract component).

```
::/*calling*/atomic{((p_c==9)&&(partner!=out_channel)
  &&(state==st_call2)&&(!(partner[selfid] == self))
  &&(len(partner[selfid])==0))->
 state=on;partner[selfid]!self,0;self?messchan,messbit;
 self!partner[selfid],0;messchan=null;messbit=0;
 MYSTATE[selfid]=oalert;p_c=17}
::/*calling*//*abstract choice*/ atomic{((p_c==9)
  &&(partner==out_channel)&&
  (state==st_call2))->
  state=on;
  self?messchan,messbit;
  self!partner[selfid],0;messchan=null;messbit=0;
  MYSTATE[selfid]=oalert;p_c=17}
```

When the partner is abstract, a read from the partner's channel (*out_channel*) is replaced by a non-deterministic choice over the possible contents (if any) of the partner's channel. For example, consider the following two statement in $GC\_form.p$ associated with the *talert* state. The first statement indicates the action to be taken when the partner's channel contains the name of the User's own channel (self), and the second when it does not.

```
::/*talert*/atomic{((p_c==21)&&(len(partner[selfid])==1)&&
  (partner[selfid]?[self,messbit]))->
   messchan=null;messbit=0;
   MYSTATE[selfid]=tpickup;p_c=22}
::/*talert*/atomic{((p_c==21)&&(len(partner[selfid])==1)&&
  (!(partner[selfid]?[self,messbit])))->
   self?messchan,messbit;
   partner[selfid]=null;partnerid=default1;
   messchan=null;messbit=0;
   MYSTATE[selfid]=preidle;p_c=24}
```

These are replaced by the following statements in $Abs\_GC.p$

```
::/*talert*/atomic{((p_c==21)&&(partner!=out_channel)&&
  (len(partner[selfid])==1)&&
  (partner[selfid]?[self,messbit]))->
   messchan=null;messbit=0;
   MYSTATE[selfid]=tpickup;p_c=22}
::/*talert*/atomic{((p_c==21)&&(partner!=out_channel)&&
  (len(partner[selfid])==1)&&
  (!(partner[selfid]?[self,messbit])))->
```

```
      self?messchan,messbit;
      partner[selfid]=null;partnerid=default1;
      messchan=null;messbit=0;
      MYSTATE[selfid]=preidle;p_c=24}
   ::/*talert*//*abstract choice*/
     atomic{((p_c==21)&&(partner==out_channel))->
      messchan=null;messbit=0;
      MYSTATE[selfid]=tpickup;p_c=22}
   ::/*talert*//*abstract choice*/
     atomic{((p_c==21)&&(partner==out_channel))->
      self?messchan,messbit;
      partner[selfid]=null;partnerid=default1;
      messchan=null;messbit=0;
      MYSTATE[selfid]=preidle;p_c=24}
```

Similarly, any poll of the User's own channel (or a read, followed by an action determined by the contents of the channel) which occurs after a connection has been established, is replaced by non-deterministic choice.

Finally, $featureAbsGC.p$ contains an abstract model in which abstract features are assumed to contain any of our suite of features, apart from RWF (return when free). The statements (from $Abs\_GC.p$) that must be modified are feature propositions, feature statements, and statements which precede feature statements (i.e. statements which contain an atomic proposition $(p\_c == r)$ where $(p\_c == r + 1)$ is a conjunct of proposition $position\_prop$, described above).

Let us consider each feature separately. Note that any feature proposition that relates to any feature not present in our concrete components will be commented out in our model. However, for features that are partner owned (see section 5.2), these statements must be replaced by propositions which implement the feature when the partner is abstract and the feature applicable. For example consider the forwarding features CFU and CFB (call forwarding on busy). In $featureAbsGC.p$ the feature proposition associated with both of these features is

```
 /*CFU or CFB*/
#define feature_property17 ((partnerid==Absid)
 &&(state==st_diall)&&(forwarding_feature==on))
```

where $forwarding\_feature$ is a local variable which, if the current partner is abstract, is non-deterministically set to *on* or *off* in the preceding statement. The corresponding feature statements are:

```
/*forward to a new concrete partner*/
::atomic{((position_prop)&&(feature_property17))->
  partnerid=0;partner[selfid]=zero;forwarding_feature=off}
::atomic{((position_prop)&&(feature_property17))->
  partnerid=1;partner[selfid]=one;forwarding_feature=off}
::atomic{((position_prop)&&(feature_property17))->
  partnerid=2;partner[selfid]=two;forwarding_feature=off}
 /*or forward to another abstract process*/
 /*or do not forward because partner is not busy (CFB)*/
::atomic{((position_prop)&&(feature_property17))->
  forwarding_feature=off}
```

Note that a non-deterministic choice is made as to whether the call is forwarded to one of the concrete components, to another abstract component, or not forwarded at all. Partner-owned, screening features (TCS, terminating call screening, and OCO, outgoing calls only) are dealt with in a similar way. Modifications to all other features, which are not relevant to this example, may be seen (commented out) in the full program $featureAbsGC.p$, on our website.

All of the programs can be verified using SPIN. For the interested reader we provide verification statistics for each run in table 3 below. In each case we give the number of stored states ($\times 10^6$), the memory used for state storage (in Mb), the maximum search depth reached ($\times 10^5$), and the time taken for verification (user + system, in seconds). All verifications were performed on a PC with a 2.4GHz Intel Xenon processor, 3Gb of available main memory, running Linux (2.4.18). State compression was used throughout.

model	states	Memory	Depth	Time
$original.p$	1.6	53.4	7.5	50
$GC\_form.p$	15.9	562	57	980
$Abs_GC.p$	1	423	16.5	608
$featureAbsGC.p$	22	788.3	30	1628

Table 3: Verification results

9 Related Work

Our induction approach involves constructing a process $\mathcal{M}^m_{abs}$, which encapsulates the behaviour of any number of processes. As such, our approach is similar to other induction approaches which involve the construction of an *invariant* process. Kurshan et al [17] prove a structural induction theorem for processes using the simulation pre-order (see section 2) to generate an invariant when there is no context process. Similar results are achieved [4, 22] by establishing a bisimulation equivalence between global state graphs of systems of different sizes. Extensions to these early results, when a (non-trivial) context process is involved, include [15, 3, 17, 1]. In some cases [20, 12] network grammars are used to generate both suitable families and an invariant.

In [7] we introduced our generalisation technique for feature interaction analysis of a telephone system with any number of components. In [8, 9] we applied a similar approach to an email system, allowing limited sets of features in abstract components. In [10] we began to investigate a more systematic way to relax the constraint on features in abstract components and to formalise our approach. We introduced the GC (guarded command) form as a uniform way of expressing basic components and features.

10 Conclusions and future work

The key, novel contribution of this paper is a technique for proving general results about systems of featured components. The technique extends our previous work so that we can allow *arbitrary* sets of *safe* features in the abstract components. We define criteria on the features to ensure that they are safe. These criteria express constraints on the variables which occur in feature guards and the form of feature action; they impose a taxonomy on features.

The main result is a theorem which states that if a feature is safe, then our generalisation technique for feature interaction detection is applicable. This result is useful because it does not depend on the intention of a feature, nor on the application domain, but only the checkable form of a feature.

The classification depends on whether a feature is *host*, *partner*, *third party* or *multi*-owned, and *single* or *double* indexed. We applied the classification to two well known featured systems, *POTS* and email. We found that all the features considered are safe, except RWF. This is because it triggers behaviour when it is both host owned and partner owned.

Future work will involve applying the results to 3-way calls and other featured systems; we also intend to investigate how the GC form relates to other feature presentations such as Ryan et al's feature construct [19].

References

[1] Parosh Aziz Abdulla and Bengt Jonsson. On the existence of network invariants for verifying parameterized systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 180–197. Springer-Verlag, 1999.

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

[3] Paul C. Attie and E. Allen Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20(1):51–115, January 1998.

[4] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81:13–31, 1989.

[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 M.B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 143–162, Toronto, Canada, May 2001. Springer-Verlag.

[7] 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.

[8] Muffy Calder and Alice Miller. Generalising feature interactions in email. In D. Amyot and L. Logrippo, editors, *Feature Interactions in Telecommunications and Software Systems VII*, pages 187–205, Ottawa, Canada, June 2003. IOS Press (Amsterdam).

[9] Muffy Calder and Alice Miller. Detecting feature interactions: how many components do we need? In Mark Ryan, Dieter Ehrich, and John-Jules Meyer, editors, *Objects, agents and features*, Lecture Notes in Computing Science, pages 45–66. Springer-Verlag, 2004.

[10] Muffy Calder and Alice Miller. Verifiying parameterised, featured networks by abstraction. In T. Margaria, B. Steffan, A. Philippou, and M. Reitenspiess, editors, *Proceedings of the first International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, pages 227–234, October – November 2004.

[11] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Masachusetts, 1999.

[12] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In 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*, pages 395–407, Philadelphia, PA., August 1995. Springer-Verlag.

[13] D. L. Dill. The Murϕ verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, NJ, USA, July/August 1996. Springer-Verlag.

[14] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, July 1992.

[15] Mahesh Girkar and Robert Moll. New results on the analysis of concurrent systems with an indefinite number of processes. In Bengt Jonsson and Joachim Parrow, editors, *Proceedings of the 5th International Conference on Concurrency Theory (CONCUR '94)*, volume 836 of *Lecture Notes in Computer Science*, pages 65–80, Uppsala, Sweden, August 1994. Springer-Verlag.

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

[17] 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.

[18] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[19] Malte Plath and Mark Ryan. The feature construct for SMV: Semantics. In M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI*, pages 129–144. IOS Press (Amsterdam), 2000.

[20] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In Sifakis [21], pages 151–165.

[21] J. Sifakis, editor. *Proceedings of the International Workshop in Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, Grenoble, France, June 1989. Springer-Verlag.

[22] Pierre Wolper and Vinciane Lovinfosse. Properties of large sets of processes with network invariants (extended abstract). In Sifakis [21], pages 68–80.