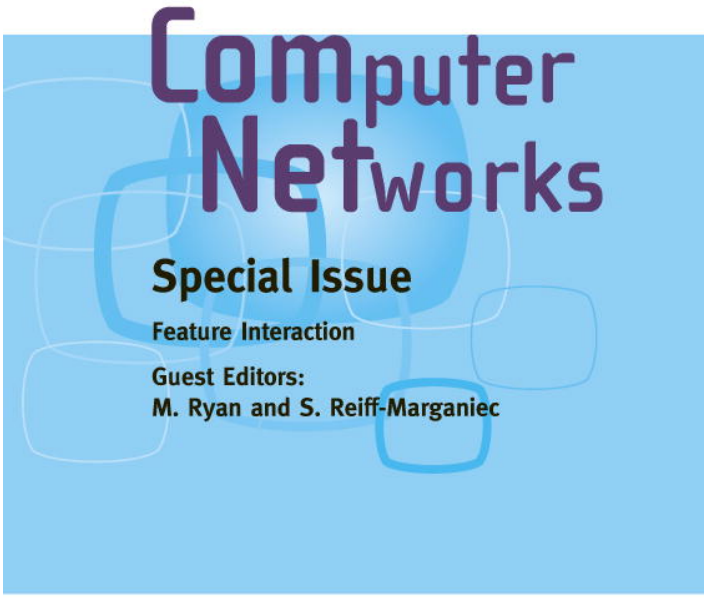
Volume 51 Issue 2 7 February 2007 ISSN 1389-1286



Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 51 (2007) 439–455
Computer Networks
www.elsevier.com/locate/comnet

A template-based approach for the generation of abstractable and reducible models of featured networks

A. Miller *, M. Calder, A.F. Donaldson

Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, United Kingdom

Available online 22 September 2006

Responsible Editor: H. Rudin

Abstract

We investigate the relationship between symmetry reduction and inductive reasoning when applied to model checking networks of featured components. Popular reduction techniques for combatting state space explosion in model checking, like abstraction and symmetry reduction, can only be applied effectively when the natural symmetry of a system is not destroyed during specification. We introduce a property which ensures this is preserved, *open symmetry*. We describe a template-based approach for the construction of open symmetric Promela specifications of featured systems. For certain systems (*safely featured parameterised systems*) our generated specifications are suitable for conversion to abstract specifications representing any size of network. This enables feature interaction analysis to be carried out, via model checking and induction, for systems of any number of featured components. In addition, we show how, for *any* balanced network of components, by using a graphical representation of the features and the process communication structure, a group of permutations of the underlying state space of the generated specification can be determined easily. Due to the open symmetry of our Promela specifications, this group of permutations can be used directly for symmetry reduced model checking.

The main contributions of this paper are an automatic method for developing open symmetric specifications which can be used for generic feature interaction analysis, and the novel application of symmetry detection and reduction in the context of model checking featured networks.

We apply our techniques to a well known example of a featured network – an email system.

Keywords: Model checking; Feature interaction; Induction; Abstraction; Symmetry reduction

1. Introduction

Model checking [14,36,38] is a popular automated approach for investigating the behaviour of computer networks. A system is specified using a modelling language, and a state space (or *model*) generated. The state space is explored to check properties that are expected to hold for the original system. In particular, model checking is a useful technique for carrying out *feature interaction* analysis on networks of featured components. However, model checking suffers from the well known state space explosion problem: the size of the state space grows exponentially with the number of components.

* Corresponding author.
E-mail address: alice@dcs.gla.ac.uk (A. Miller).

1389-1286/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2006.08.009

Approaches for combatting state space explosion often involve abstraction to replace sets of states with state representatives. One method, induction, is used to construct an (abstract) state space which encapsulates the behaviour of systems of *any* size. This method is useful for ensuring that properties which hold for small, finite systems, still hold when any number of new components are added to the system. However, if a property does not hold for the abstract state space, no general result can be inferred (and no meaningful counter-example generated).

For large, finite systems, symmetry reduction is an alternative reduction technique which can be used to reduce the size of the state space – sometimes dramatically. Symmetry reduction involves finding a group of permutations of the state space which preserve the property to be checked, and using it to build a (smaller) *quotient* state space. The property will hold for the quotient state space if and only if it holds for the original state space.

In previous work [9,37] we have used an abstraction/induction approach to model and analyse parameterised networks of featured components, for networks of any size. Our approach relies upon restricting the behaviour of the components to be *open symmetric*. Open symmetry requires that for any statement in the specification that refers to a literal component id, all symmetrically equivalent statements are present in the component specification.

We have also developed an approach, for balanced networks of unfeatured components, to detect the symmetry present in a system using a graphical representation of the process communication structure – the *static channel diagram* (*SCD*) [17,19]. The SCD is generated automatically from a Promela specification of the system, and a suitable automorphism group of the state space ($G \subseteq \mathrm{Aut}(\mathscr{M})$) is obtained from the automorphism group of the SCD ($\mathrm{Aut}(SCD)$). Although $\mathrm{Aut}(SCD)$ can be found easily and automatically, some elements of $\mathrm{Aut}(SCD)$ are not *valid*: they do not belong to $\mathrm{Aut}(\mathscr{M})$. Thus it is necessary to remove all invalid elements of $\mathrm{Aut}(SCD)$ to obtain a suitable automorphism group G. This involves checking the validity of the group generators against the Promela specification itself. If the model could be ensured to be open symmetric however, all elements of $\mathrm{Aut}(SCD)$ would be valid with respect to the model, and we would only need to check the validity of the generators against the property to be verified. This would be faster and in many cases would mean that we could use $G = \mathrm{Aut}(SCD)$ directly for symmetry reduced model checking.

While conducting our previous work on abstraction/induction and symmetry detection we have been struck by strong parallels between the approaches used. For example, in both cases the techniques are less effective (or do not apply at all) if components are not open symmetric. We have become increasingly aware that the tools we have developed for constructing models suitable for abstraction/induction, could be used to construct finite models of networks of featured components to which symmetry reduction can be applied.

In this paper we show how, for any *balanced* network of featured components we can use a template-based approach to generate Promela specifications which are, by construction, open symmetric. For *safely featured parameterised systems* the generated specifications are suitable for applying our induction approach.

In addition, we introduce a new graphical representation of the specification – a *feature configuration diagram* (*FCD*) and show that the automorphism group of the FCD induces an automorphism group of the underlying state space, for any balanced system. This allows for immediate application of symmetry reduction methods, without the need to check for symmetry validity. We present a tool – the *featured specification generator* (*FSG*) to implement our approach, and we present experimental results for an email system. This extension of our earlier work is the first time we have applied symmetry detection methods to networks of featured components.

Our methods are illustrated via two example networks: a telephone network, in which all components are of the same type, and an email network in which there are two types of component – client components and a mailer component. However, it is important to point out that our techniques are applicable to networks consisting of multiple component types.

2. Background

2.1. Systems and specifications

Consider a system of communicating components. A specification of the system consists of a set of processes (each describing a component) together with a set of channels. Processes can be separated into different types according to the type of component they represent (e.g., *client* component or *server* component).

Definition 1. A specification with k process types, for some $k > 0$, consists of the parallel execution of processes thus

$$p_{1,1}|| \ldots ||p_{1,n_1}||p_{2,1}|| \ldots ||p_{2,n_2}|| \ldots ||p_{k,1}|| \ldots ||p_{k,n_k},$$

where, for $1 \leqslant i \leqslant k$, n_i is the number of components of type i.

Channels are also classified by type according to their length, and the type of messages they can contain.

Definition 2. We say that a system is *balanced* if:

1. Every component has a single dedicated channel for incoming messages.
2. If components i and j have the same type and have dedicated channels i' and j' respectively, then i' and j' have the same type.
3. If component i has type t_i and can send messages to component j, which has type t_j, then component i can send messages to all components of type t_j.

The specification of a balanced system is called a *balanced specification*.

An example of a balanced system is a simple peer-to-peer system consisting of n components each of which send and receive the same type of messages, and can communicate with every other component. Similarly a system consisting of $n-1$ *client* components (say) together with a *hub* component which is responsible for relaying messages between the components is also balanced. However, consider a tiered system consisting of a number of *client* components together with a number of *server* components, in which *client* components send messages to a *specified server* component (but not to other *server* components). This system is not balanced: to be balanced, each *client* component should be able to send messages to either all or none of the *server* components. Features can be used to add this type of selective behaviour to an otherwise balanced system.

2.2. Features

System components with the same type may have different functionality. The mechanism for structuring functionality additional to a basic behaviour is commonly called a *feature*. The concept originated in telephony where features such as call forwarding, ring back when free, etc., are added to a basic call behaviour. Features fundamentally affect basic behaviour in different ways, and so components with features are not, in general, isomorphic. Moreover, the features associated with one component can affect the behaviour of other (possibly featured) components.

A component is said to *subscribe* to a feature f (belonging to a given set of features), and a network is *featured* when (at least one of) the associated components subscribes to at least one feature. The component that subscribes to a feature is known as the feature *host*. We assume that, when features are implemented within a specification of a system, they are implemented via an array.

An *instantiation* of a feature f is an application of f to a given component or set of components. A *configuration* of feature f is a a set of instantiations of f for a given system. For example, in a telephone system, setting component 2 to unconditionally forward messages to component 3 is an instantiation of the *call forward unconditional* (CFU) feature.

A *unary* feature is a feature which is instantiated with respect to a single component and a *binary* feature is a feature which is instantiated with respect to two components. For example, *originating calls only* (OCO) is a unary feature and *terminating call screening* (TCS) is a binary feature. We assume that unary features are instantiated using a statement of the form $F[i] = 1$ with the default instantiation $F[i] = 0$. Similarly binary features are instantiated using a statement of the form $F[i][j] = 1$, where i is the feature host. In this case the default instantiation is $F[i][j] = 0$.

A binary feature f induces a relation $R(f)$ on the set $\{1, 2, \ldots, n\}$ of component identifiers:

$$(i, j) \in R(f) \quad \text{if } f[i][j] = 1.$$

We use $\chi_{R(f)}$ to denote the characteristic function of $R(f)$, which maps (i,j) to 1 if $(i,j) \in R(f)$, and maps (i,j) to 0 otherwise ($1 \leqslant i,j \leqslant n$).

2.2.1. Feature interaction

Property-based feature interaction analysis involves 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 $S + f_1 + f_2 \models \phi$?

Assuming c is a component with no features, c_{f_1} and c_{f_2} 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 $(c_{f_1}||c) \models \phi$ imply $(c_{f_1}||c_{f_2}) \models \phi$?
Of course a component can subscribe to more than one feature. For example, assuming that c_{f_1,f_2} is a component with features f_1 and f_2, then another example of feature interaction analysis is:
(ii) does $(c_{f_1}) \models \phi$ imply $(c_{f_1,f_2}) \models \phi$?

2.3. Model checking

Model checking involves checking *Kripke structures* [14] to verify given temporal properties.

Definition 3. Let AP be a set of atomic propositions. A Kripke structure over AP is a tuple $\mathscr{M} = (S, S_0, R, L)$ where S is a finite set of states, $S_0 \subseteq S$ 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.

From here on we will assume that all models have a single initial state s_0.

The logic LTL is defined as a set of formulas of the form $A\phi$ where the quantifier A is used to denote *for all paths* and ϕ is a path formula in which the only state subformulas are atomic propositions. The set of LTL path formulas are defined inductively below where X, U, $\langle\rangle$ and $[]$ represent the standard *nexttime*, *strong until*, *eventually* and *always* operators (where $\langle\rangle\phi = trueU\phi$ and $[]\phi = \neg\langle\rangle\neg\phi$ respectively). Let AP be a finite set of propositions. Then

- for all $p \in AP$, p is a path formula;
- if ϕ and ψ are path formulas, then so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $X\phi$, $\phi U\psi$, $\langle\rangle\phi$ and $[]\phi$.

When referring to an LTL formula, one generally omits the A operator and instead interprets the formula ϕ as "for all paths ϕ". For a model $\mathscr{M}$, if the formula ϕ holds for all paths in $\mathscr{M}$ we write $\mathscr{M} \models \phi$.

Definition 4. Let $\mathscr{M}$ and $\mathscr{M}'$ be Kripke structures with associated sets of atomic propositions AP and AP' respectively, where $AP \subseteq AP'$. A relation $H \subseteq S \times S'$ is a *simulation relation* between $\mathscr{M}$ and $\mathscr{M}'$ 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 $\mathscr{M}'$ *simulates* $\mathscr{M}$ and write $\mathscr{M} \preceq \mathscr{M}'$.

The following is derived from a well known result [14].

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

2.3.1. Promela and Spin

Promela is an imperative specification language with constructs for concurrency, nondeterminism, asynchronous and synchronous communication, dynamic process creation, parameterised processes, and mobile connections, i.e., communication channels can be passed along other communication channels. Spin is the bespoken model checker for Promela and provides several reasoning mechanisms: deadlock and assertion checking, acceptance and progress states and cycle detection, and satisfaction of LTL properties.

In order to perform verification on a Promela specification, Spin translates a process template for each process type into a finite automaton and then computes an asynchronous interleaving product of these automata to obtain the global behaviour of concurrent specification. This interleaving product is referred to as the *state space*. We can infer properties of a concurrent system by checking properties of the state space associated with a Promela specification of the system. Given a Promela specification with n processes, the associated model or Kripke structure (which we identify with the state space), is denoted by $\mathscr{M}_n$.

2.3.2. The parameterised model checking problem (PMCP)

We consider specifications (see Definition 1) in which either all processes have the same type, or there is a distinguished context process, and all other processes have the same type. That is, our specifications have the form

$$p_1||p_2||\ldots||p_N,$$

where the p_i, for $2 \leqslant i \leqslant N$ have the same type and p_1 may or may not have the same type as p_i for $2 \leqslant i \leqslant N$. For each specification we can generate

a family of specifications by successively increasing N. As such, each specification

$$p_1||p_2||\ldots||p_N$$

is called a *parameterised specification* and the associated system is called a *parameterised system*.

For a fixed N, provided N is small enough, we can check that a property ϕ holds for a parameterised specification by building a model $\mathscr{M}_N$ (using SPIN) and checking that $\mathscr{M}_N \models \phi$. However, what can we infer about such a system in general?

Note that parameterised systems are not limited to having one or two types of component. Indeed, the parameterised model checking problem can be extended to systems with multiple component types where the number of components of one of the types is unlimited. We consider only the simpler cases here for ease of argument.

Definition 5. For a parameterised specification as described above, the *parameterised model checking problem* (*PMCP*) is thus: for an LTL property ϕ, can we show that $\mathscr{M}_N \models \phi$ for *any* N?

It is not possible to solve PMCP using model checking alone [2]. However, one approach that has proved successful for verifying some parameterised systems involves the construction of a *network invariant* [4,13,34]. The network invariant I represents an arbitrary member of the family $\mathscr{F} = \{\mathscr{M}_n : n \geqslant n_0\}$ and proof of a given property ϕ for I can be shown to imply that any member of the family $\mathscr{F}$ satisfies ϕ.

Some other techniques that have been used to verify parameterised systems include those based on theorem proving [15], on abstraction [28], or on a combination of the two [31]. A further method is to use explicit inductive techniques combined with model checking [20,24,35,42].

In Section 4 we describe an invariant-based approach which combines abstraction and induction to verify parameterised systems in which individual components may be distinguished by way of features. Our *invariant* process is constructed by modifying a Promela specification for a network of fixed size, and using SPIN to construct the corresponding Kripke structure. Our approach is an example of how an invariant processes can be constructed in practice, to extend results proved for small, fixed sized models, to results which hold for models of any size. One of the major assumptions we make is that our models satisfy a property known as *open symmetry*.

Like all network invariant approaches, this approach is limited to systems with a regular topology, which grow in a regular way as the number of components increases.

2.4. Open symmetry

In this section we summarise some definitions from group theory which we will use to define the conditions for our abstraction approach in Section 4 and to describe symmetry reduction techniques in Section 5.

Definition 6. Let G be a non-empty set, and let $\circ : G \times G \to G$ be a binary operation. We say that $(G, \circ)$ is a *group* if G is closed under $\circ$; $\circ$ is associative; G has an identity element 1_G; and for each element $\alpha \in G$ there is an inverse element $\alpha^{-1} \in G$ such that $\alpha \circ \alpha^{-1} = \alpha^{-1} \circ \alpha = 1_G$.

We call the operation $\circ$ *multiplication* in G. When it is clear what the binary operation is, we simply refer to a group as G rather than $(G, \circ)$, and use concatenation to denote multiplication.

For any group G be containing elements $\alpha_1, \alpha_2, \ldots, \alpha_n$, the smallest subgroup of G containing the elements $\alpha_1, \ldots, \alpha_n$ is denoted $\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$, and is called the subgroup *generated* by $\alpha_1, \alpha_2, \ldots, \alpha_n$. The elements α_i $(1 \leqslant i \leqslant n)$ are called *generators* for this subgroup.

The set of all permutations of a set X forms a group under composition of mappings $Sym(X)$. For any Promela specification P say, we can apply a permutation (of component ids) α to the statements of P by replacing every occurrence i of a literal component id with $\alpha(i)$.

Definition 7. Let P be a Promela specification and V the associated set of component ids. We say that P is *open symmetric* if, for any process p of P, for any statement ψ of p, $\alpha(\psi)$ is a statement of p for all $\alpha \in Sym(V)$.

An automorphism of a Kripke structure $\mathscr{M} = (S, R, L, s_0)$ is an edge-preserving permutation of S which fixes s_0.

3. Feature interaction analysis using model checking

We assume that our systems are balanced (see Definition 2). As examples of this type of system, we have modelled a featured telephone system consisting of N instantiations of a *user* component and a featured email system consisting of a *mailer*

component and *N client* components. In the Promela specifications for these systems, each process is an instantiation of a parameterised *proctype* process, where the only parameter is the dedicated channel name for the process.

3.1. Promela specifications

The basic (unfeatured) Promela process proctypes are based on high-level abstract automata (see [8] for full details) for the system components. In the automata, transitions are made between *call-states* (like *idle* and *calling* in the telephone example, and *sendmail* and *delivermail* in the email example). These call states are represented in the specification using labels.

Features are included via global *feature arrays* and implemented via feature statements of the form

```
atomic{(feature_prop)&& (localprop)
&& (varprop)->command_guard}
```

where *feature_prop* is a proposition which checks whether a feature is subscribed to, and *localprop* is a proposition about local variables. The proposition *varprop*, which may be empty, is a proposition about global variables. These global variables may include elements of global *feature_flag* arrays which indicate whether a feature has been instigated during a previous call. For example, the variable may indicate whether or not a ringback been requested.

Our specifications of featured networks are *safely featured*, as defined below:

Definition 8. A specification of a balanced featured network is said to be *safely featured* if the only global variables are channels or elements of global arrays indexed by process ids. These global arrays are either feature arrays (which are fixed), or arrays for which the element with index i is only set or checked by component with pid i.

Note that the latter type of global arrays are generally only included in a specification for verification purposes (otherwise a local variable would suffice). We are not able to include the *return when free* (RWF) feature in our telephone specification because it requires the use of a *feature_flag* array in which element with index i is set by components other than that with pid i.

Feature configurations (see Section 2.2) are declared within the *init* process. For full example code for both the email and telephone models see [9].

3.2. Feature interaction analysis

For every feature f, an LTL property, or set of LTL properties is constructed describing the functionality of the feature. The features are hereafter assumed to be defined by the properties describing them. For example, in the telephone example the *call forwarding unconditionally* feature (CFU) (from *user i* to *user j*) property is: If *user k* dials *i* then a call attempt will be made from *user k* to *user j* before *user k*'s handset is replaced. When $i = 0$, $j = 1$ and $k = 2$ this is described in LTL as $[](p \rightarrow (\neg((\neg r)Uq)))$, where $p = ((dialled[2] == 0) \wedge (user[p2]@calling))$, $q = (dev[2] == on)$ and $r = ((partner[2] == one) \wedge ((user[p2]@oalert) \vee (user[p2]@busy)))$.

To validate a feature for a given set of parameters (i, j and k in this example) a Promela specification is constructed for a small number of processes, in which the feature is initiated (by setting the associated feature array). The associated LTL property (or a set of properties in some cases) is then checked using SPIN. To determine whether a pair of features interact, a specification in which both features are initiated is checked separately against the LTL properties for the two features.

Note that, for any feature (or pair of features) there are many cases to be checked, depending on the values taken by the feature parameters. For example, to validate CFU for a specification of 6 processes there are 216 possible combinations of i, j and k. However, this can be reduced to a much smaller number of cases (3 in this example) by exploiting the symmetry between the processes [10].

Full feature interaction results for both the telephone and email examples are given in [8].

4. Abstraction/induction for model checking featured networks

As we saw in Section 3.2, we can use model checking for feature interaction analysis of a fixed number of components. However, suppose that, for the telephone example, we have shown that features f_1 and f_2 do not interact when considered within a specification of a system of 5 *user* components, can we be sure that they do not interact within a specification of a system containing additional, possibly featured components? That is, can we prove generic properties of specifications of our

telephone system? Similarly, can we prove generic properties for the email system?

When unfeatured, our two example systems are parameterised systems, as described in Section 2.3.2. In order to prove generic properties of our systems in the presence of features, we will have to extend the PMCP (see Definition 5) to *featured parameterised systems*. Using our approach we can:

1. validate features for specifications consisting of any number of processes (and for which only one process has any features, namely the feature being validated), and
2. identify all pairs of features (f_1 and f_2 say) that do not interact, regardless of the numbers of components.

We outline our approach in Section 4.1 below. For a more detailed description and proofs of results, see [9,37].

4.1. The abstraction/induction approach

We consider parameterised systems (Definition 2.3.2) with associated parameterised specifications of the form

$$p_1||p_2||\ldots||p_N$$

where p_1 may, or may not, be a distinguished context process, and may have associated features. Otherwise, all of the processes are instantiations of the same parameterised process and may, in addition, have features. We refer to such specifications as *featured parameterised specifications*, and the corresponding systems as *featured parameterised systems*.

For any feature f, we say that f is *indexed* by $I_f = \{i_1, \ldots, i_r\}$ if the feature relates to components $i_1 \ldots i_r$. For example, in the telephone example, if f is "*user*[0] forwards calls to *user*[3]", then f is said to be indexed by 0 and 3. Similarly we say that a property ϕ is indexed by a the set I_ϕ where I_ϕ is the set of component ids associated with ϕ. For a (possibly empty) set of features $F = \{f_1, \ldots, f_s\}$ and property ϕ, we define the *complete index set* I of $\{\phi\} \cup F$, to be $I_{f_1} \cup \cdots \cup I_{f_s} \cup I_\phi$.

Suppose that for features f_1 and f_2 and property ϕ, the complete index set (or complete index set together with 1 if there is a context process in the specification) is $1 \ldots m$. For every $N > m$, and every set of features subscribed to by components $p_{m+1}, \ldots, p_N$, we can generate a specification $p_1||p_2||\ldots||p_N$. This gives rise to an infinite family, $A = A_{f_1,f_2,\phi}$, of specifications. We call these *concrete specifications*.

For each f_1, f_2 and ϕ we generate a new, finite model, *abs*(m) which represents *any* member of A, and show that, provided our specifications (and features) satisfy certain conditions, $\mathcal{M}(abs(m)) \models \phi$ implies that $\mathcal{M}_N \models \phi$ for all $\mathcal{M}_N \in \mathcal{M}(A)$, where $\mathcal{M}(A)$ is the set of models associated with F. Note that *abs*(m) is an invariant process for the system (see Section 2.3.2).

In *abs*(m), all processes $p_{m+1}, \ldots, p_N$ (which we call *abstracted processes*) are represented by a single process *Abstract*(m), and all processes $p_1, \ldots, p_m$ (concrete processes) are represented by *modified concrete processes* $p'_1, \ldots, p'_m$. The abstraction is illustrated in Fig. 1, with the original specification on the left hand side and the specification *abs*(m) appearing on the right hand side. Our main theorem

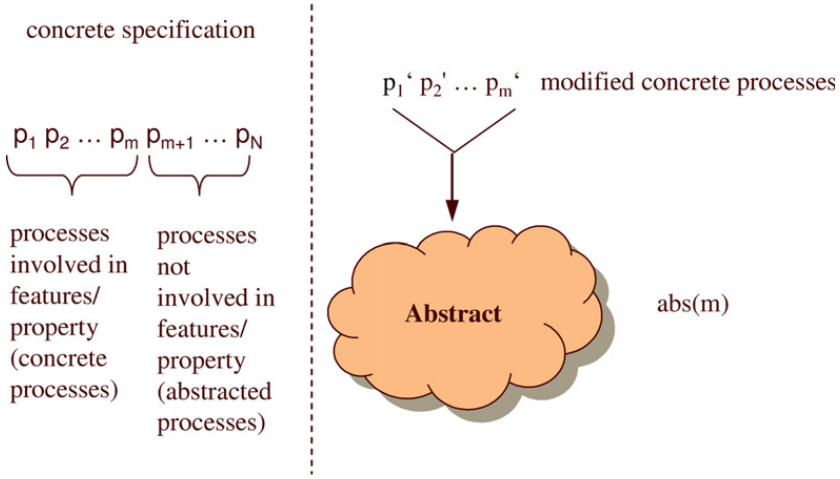


Fig. 1. Abstraction technique.

holds because there is a simulation relation (see Definition 4) between the two specifications.

Theorem 1. *Let* $\mathscr{M}_N = \mathscr{M}(p_1||p_2||\ldots||p_N)$ *be a model of any featured parameterised specification in which features F are present, and* ϕ *a property. If the total index set of* $F \cup \{\phi\}$ *is* $\{1,\ldots,m-1\}$ *then, provided the specification is both open symmetric and safely featured,* $\mathscr{M}_{abs(m)} \models \phi$ *implies that* $\mathscr{M}_N \models \phi$.

The full proof of Theorem 1, together with a details of how the modified processes are derived from the original concrete processes, and a description of *Abstract*(*m*) is given in [9,37].

5. Symmetry reduced model checking

In a model of a concurrent system with many replicated processes, Kripke structure automorphisms (see Section 2.4) usually involve the permutation of process identifiers of identical processes throughout all states of a model. The set of all automorphisms of the Kripke structure $\mathscr{M}$ forms a group under composition of mappings. This group is denoted $\mathrm{Aut}(\mathscr{M})$. A subgroup G of $\mathrm{Aut}(\mathscr{M})$ induces an equivalence relation $\equiv_G$ on the states of $\mathscr{M}$ thus: $s \equiv_G t \iff s = \alpha(t)$ for some $\alpha \in G$. The equivalence class under $\equiv_G$ of a state $s \in S$, denoted $[s]$, is called the *orbit* of s under the action of G. The orbits can be used to construct a *quotient* Kripke structure $\mathscr{M}_G$ as follows:

Definition 9. The quotient Kripke structure $\mathscr{M}_G$ of $\mathscr{M}$ with respect to G is a tuple $\mathscr{M}_G = (S_G, R_G, L_G, [s_0])$ where

- $S_G = \{[s] : s \in S\}$ (the set of orbits of S under the action of G),
- $R_G = \{([s],[t]) : (s,t) \in R\}$,
- $L_G([s]) = L(rep([s]))$ (where $rep([s])$ is a unique representative of $[s]$),
- $[s_0] \in S_G$ (the orbit of the initial state $s_0 \in S$).

In general $\mathscr{M}_G$ is a smaller structure than $\mathscr{M}$, but $\mathscr{M}_G$ and $\mathscr{M}$ are equivalent in the sense that they satisfy the same set of logic properties which are *invariant* under the group G (that is, properties which are "symmetric" with respect to G). For a proof of the following theorem see [14].

Theorem 2. *Let* $\mathscr{M}$ *be a Kripke structure,* G *a subgroup of* $Aut(\mathscr{M})$ *and* ϕ *an LTL formula. If* ϕ *is invariant under the group G then*

$$\mathscr{M}, s \models \phi \iff \mathscr{M}_G, [s] \models \phi$$

Thus by choosing a suitable symmetry group G, model checking can be performed over $\mathscr{M}_G$ instead of $\mathscr{M}$, often resulting in considerable savings in memory and verification time [3,12].

If automorphisms of a Kripke structure can be identified in advance, then a quotient structure can be incrementally constructed using an algorithm given in [29]. This means that it may be possible to construct the quotient structure even if the original structure is intractable.

In [19] we show that symmetries of the Kripke structure associated with a Promela program can be detected by analysing a graph derived from the associated Promela specification, namely the *static channel diagram* of the specification. We summarise this approach below. In Section 7 we show that, if Promela specifications are generated using our template-based approach (see Section 6), we can extend our symmetry detection techniques to effectively handle featured networks of components.

5.1. Detecting symmetry in Promela specifications of unfeatured networks

Given a Promela specification $\mathscr{P}$, the static channel diagram [17,18] of $\mathscr{P}$, SCD is a graphical representation of the communication structure associated with $\mathscr{P}$. The nodes of the graph represent the processes and *static channels* (channels which are declared globally within the specification, out of the scope of any proctype definition). Nodes are coloured, according to component or channel type. An edge exists between nodes associated with component i and channel j if and only if component i can send messages on channel j. Static channel diagrams extend the notion of *channel diagrams* introduced in [43].

Generators for a group of *candidate* automorphisms for the model $\mathscr{M}$ derived from $\mathscr{P}$ are found by analysing the SCD. These generators are checked individually against the specification to see if they induce valid automorphisms of $\mathscr{M}$ and the largest possible subgroup of valid candidate automorphisms computed. Unlike previous approaches to specifying symmetry using *scalarsets* [3,29], the static channel diagram method can detect *arbitrary* structural symmetries arising from the communication structure of a model.

All of our symmetry groups are computed automatically using a tool: SymmExtractor, which

makes use of the computational group theory package GAP [23]. Although this approach could handle featured networks, the symmetry detection process could be inefficient due to asymmetry induced by features, which is not captured by the SCD. In Section 7 we define a new kind of diagram, the feature configuration diagram, which allows us to directly obtain structural symmetries of a featured model.

6. The template-based approach

In this section we describe how we use a template-based approach to generate a Promela specification of a featured system, which is open symmetric and thus amenable to both abstraction (when the specification is parameterised and safely featured) and symmetry reduction.

6.1. Overview of approach

The basic idea is described in Fig. 2. We have developed a tool *Featured Specification Generator* (FSG) (a Java application) to generate an open symmetric specification from three meta files: the *proctype definitions* file, the *global definitions* file and the *feature configuration* file, together with a series of *proctype templates*, one for each proctype specified in the process definitions file.

The proctype definitions file records, for each proctype in the model, the name and number of instantiations of the proctype, and the type of the input channel for instantiations of the proctype. The global definitions file includes user-defined record types, as well as global variables (which must not be channels). The feature configuration file provides the name and configuration of each (unary or binary) feature for the model. For each proctype declared in the proctype definitions file there must be exactly one proctype template. The template for a given proctype essentially consists of the body of the proctype, but the Promela code is restricted to ensure that the generated model is open symmetric. The template body can include *template options*—parameterised statements which are expanded during model generation to allow non-deterministic choice over all component ids for a given proctype.

We describe the format of the proctype definitions file, the feature configuration file and the process templates and show how FSG uses them to generate an open symmetric Promela specification which is amenable to state space reduction by both abstraction and symmetry. We illustrate the approach using a featured email system adapted from [7]. Note that the global definitions file requires no translation by FSG (and is empty for this example).

6.2. Proctype definitions

The proctype definitions file consists of a series of definitions, each representing a single proctype. A

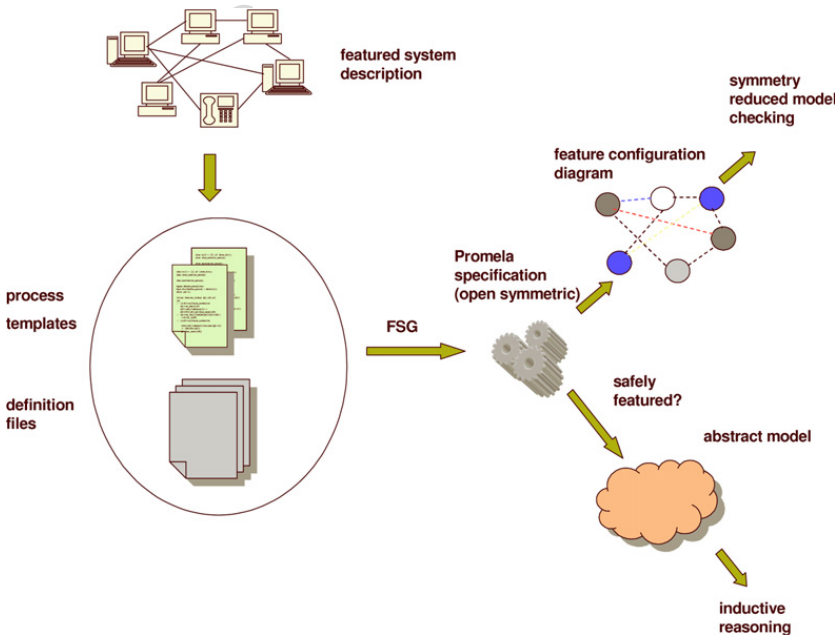


Fig. 2. The template-based approach.

definition has the form *n*: *name* → [*k*] of {*list of types*}, where $n \geqslant 1$ specifies how many copies of the proctype *name* should be instantiated, $k \geqslant 0$ is the length of the input channel for an instantiation of this proctype, and *list of types* is a tuple of Promela types or user-defined types which specifies the format which messages on this channel should conform to.

The following example gives the proctype definitions for an email system consisting of five *client* components, and a *mailer* component. The *client* input channels are one place buffers which accept messages of the form {*sen*,*rec*} where *sen* and *rec* are process ids. The *mailer* input channel is a five place buffer, which accepts messages of the same form

```
5 : client <- [1] of {pid,pid}
1 : mailer <- [5] of {pid,pid}
```

From these definitions, FSG generates an initialised channel for each instantiated process, a *lookup* procedure, headers for each proctype and the *init* process. The *lookup* procedure returns the channel name associated with a given id and the *init* process consists of a series of `run` statements instantiating the specified number of components for each proctype. Note that channel `link`*i* is the channel associated with the *i*th instantiated process and, to ensure that the resulting specification is balanced, each proctype has a single channel parameter, *in*.

FSG would generate the following from the proctype definitions file above:

```
chan link1 = [1] of pid,pid;
chan link2 = [1] of pid,pid;
...;
chan link6 = [5] of pid,pid;
inline lookup(id,link) {
  if
    :: id==1 -> link = link1
    :: id==2 -> link = link2
    ...
    :: id==6 -> link = link6
  fi
}
proctype client(chan in) {
  /* Body generated from template */
}
proctype mailer(chan in) {
  /* Body generated from template */
}
init {
  atomic {
    run client(link1); run client
    (link2);...; run client(link5);
    run mailer(link6);
    /* Feature configuration */
  }
}
```

6.3. Feature configuration

The feature configuration file specifies the name, arity and components associated with each feature for the model. A unary feature definition has the form *name*[*list of component ids*], indicating that the feature *name* is switched on for each of the components listed. The form of a binary feature is similar, except that a list of pairs of ids is specified.

Continuing the email example, the following configuration specifies a unary feature `AUTORESP` (autorespond), which is on for *client* components 1 and 2, and a binary feature `FILTER`, such that messages from *client* 5 intended for *client* components 3 or 4 should be filtered.

```
AUTORESP[1,2]
FILTER[(3,5),(4,5)]
```

FSG generates feature initiation within the *init* process and an array of size $n + 1$ for each unary feature, and a 2-dimensional array of size $(n + 1) \times (n + 1)$ for each binary feature, where n is the number of components in the system. Thus:

```
typedef array {
  bit to[7]
};
hidden bit AUTORESP[7];
hidden array FILTER[7];
...
init {
  atomic {
    ...
    AUTORESP[1] = 1; AUTORESP[2] = 1;
    FILTER[3].to[5] = 1;
    FILTER[4].to[5] = 1
  }
}
```

Note that indices of the arrays go up to $n+1$ because process identifiers are assigned from 1

upwards, but arrays in Promela are always indexed from 0. As 2-dimensional arrays are not supported directly by Promela they must be specified using a new array type, as shown above.

6.4. Proctype templates

For each proctype specified in the proctype definitions file there must be exactly one template file, named corresponding to the proctype. Recall that each proctype has a single parameter, *in*, which is the input channel for an instantiation of the proctype. The template for a proctype consists of Promela statements which must obey certain restrictions, and can include *template options*.

To ensure a fixed number of components and channels, new components may not be instantiated dynamically using a *run* statement, and channel variables may be declared, but not initialised as new channels (they may be assigned to names of existing channels). To ensure that the generated specification is open symmetric and balanced, literal component id values and the global channel names `link1`, `link2`, ... ,`link`*n* may not be referred to explicitly: literal id values may be used via a template option (see below), and global channel names may be accessed using the *lookup* inline. Finally, component identifiers may not be assigned to or from variables of other types, used as operands to arithmetic operators, or to the boolean operators $<$, $\leqslant$, $>$ and $\geqslant$.

We now define the syntax for template options. A template option may be used as a guard for an `if..fi` or `do..od` statement, to allow nondeterministic choice over all components of a specific proctype. The syntax for a template option is: `for` *name* `in` *proctype_name* {*Promela statement*}, where *name* is a legal Promela variable name which is not already used in the scope of the statement, *proctype_name* is the name of some proctype in the model, and *Promela statement* is a (simple or compound) Promela statement which may refer to *name* as if it were a literal value of type *pid*, and may itself contain template options. FSG expands choice options to include a concrete option for every associated component identifier. In the email example, a *client* component may choose to send a message, via the *mailer*, to any component of type *client*. In the *client* template, such a message is initialised using the following template option:

```
if
  :: for i in client {msg.receiver = i}
fi
```

which FSG expands to the following non-deterministic choice:

```
if
  :: msg.receiver = 1
  :: msg.receiver = 2
  :: msg.receiver = 3
  :: msg.receiver = 4
  :: msg.receiver = 5
fi
```

All of the code for the email example can be found as an appendix to this paper on our website [6]. This includes all of the definitions files, the proctype template files and the full Promela specification generated by FSG.

6.5. Applying abstraction/induction or symmetry reduction to generated specifications

By construction, the specifications generated by FSG are open symmetric. In order to apply our abstraction/induction approach of Section 4.1 to a parameterised specification, it is necessary to check that it is *safely featured*. We can do this easily using FSG. The email example above is safely featured, as is the telephone example discussed in Section 3.

In order to apply symmetry detection techniques (for symmetry reduction) no additional check is necessary. (Indeed, specifications do not even need to be parameterised (see Section 2.3.2)). In Section 7 we show that we can generate a graphical representation of the communication structure of the system from the files used to generate the specification. Symmetries of this graphical representation can be used to generate automorphisms of the underlying model.

7. Symmetry detection for featured networks

In this section, we show that if a Promela specification has been generated using the template-based approach, then a symmetry group of the model underlying the specification can be derived from a directed graph called the *feature configuration diagram* for the specification. The feature configuration diagram itself can be obtained efficiently from

the feature configuration and proctype definitions files used to generate the specification.

7.1. Feature configuration diagrams

Recall from Section 2.2 that a binary feature B naturally induces a relation $R(B)$ on the set of component identifiers of a featured specification. The *feature configuration diagram* for a Promela specification is a directed graph whose vertices are the component identifiers, coloured according to their process type and the unary features to which they subscribe, and whose edges are the elements of $R(B)$ for each binary feature with which the specification is configured. These edges are also coloured, according to the exact sets $R(B)$ to which they belong.

Formally, let $F = \mathscr{U} U \mathscr{B}$ be a set of features, where $\mathscr{U} = \{U_1, U_2, \ldots, U_x\}$ are unary, and $\mathscr{B} = \{B_1, B_2, \ldots, B_y\}$ are binary (for some x, $y \geqslant 0$).

Let $\mathscr{T}$ be a finite set of component types and, for a set of n components, let $type : \{1, 2, \ldots, n\} \to \mathscr{T}$ be a mapping which associates each component with a type.

Definition 10. Let $\mathscr{P}$ be a Promela specification configured with features $F = \mathscr{U} \cup \mathscr{B}$. The *feature configuration diagram* $FCD(\mathscr{P})$ is a directed graph with coloured vertices and coloured edges: $FCD(\mathscr{P}) = (V, E, C_V, C_E)$ where

- $V = \{1, 2, \ldots, n\}$
- $E = \bigcup_{i=1}^{y} R(B_i)$
- $C_V : V \to \{0,1\}^x \times \mathscr{T}$ is defined by $C_V(i) = (U_1[i], U_2[i], \ldots, U_x[i], type(i))$
- $C_E : E \to \{0,1\}^y$ is defined by

$$C_E((i,j)) = (\chi_{R(B_1)}((i,j)), \chi_{R(B_2)}((i,j)), \ldots, \chi_{R(B_y)}((i,j)))$$

Consider a configuration of the email model with 5 *client* components, featured as follows:

```
AUTORESP[1]
FILTER[(2,1)]
```

Let $\mathscr{P}$ be the Promela specification of the email system with five *client* components, and features as above. Then $FCD(\mathscr{P}) = (V, E, C_V, C_E)$ where $V = \{1,2,3,4,5,6\}$; $E = \{(2,1)\}$; $C_V(1) = (1, \text{client})$ (indicating that *client* component 1 has the *autorespond* feature), $C_V(i) = (0, \text{client})$ for $2 \leqslant i \leqslant 5$ (*client* components 2 to 5 have no unary features), $C_V(6) = (0, \text{mailer})$ (the *mailer* component is unfeatured); and $C_E((2,1)) = 1$ (component 2 subscribes to the *filter* feature with respect to *client* 1).

7.2. Automorphisms of feature configuration diagrams

For a Promela specification $\mathscr{P}$ which has been generated using the template-based approach of Section 6, we define an automorphism of the feature configuration diagram for $\mathscr{P}$ thus:

Definition 11. An automorphism of $FCD(\mathscr{P})$ is a bijection α of $\{1,2,\ldots,n\}$ such that

- $(i,j) \in E \iff (\alpha(i), \alpha(j)) \in E$.
- $C_V(i) = C_V(\alpha(i)) \forall i \in \{1,2,\ldots,n\}$.
- $C_E((i,j)) = C_E((\alpha(i), \alpha(j))) \ \forall i,j \in \{1,2,\ldots,n\}$.

Consider the FCD for the email example, described in Section 7.1. It is easy to check that the permutation (3 4) is an automorphism of this FCD, whereas (1 2) is not as components 1 and 2 are coloured differently. In fact if $\mathscr{P}$ is a Promela specification of the email system with this feature configuration diagram then $\text{Aut}(FCD(\mathscr{P})) = \langle (3\ 4), (4\ 5) \rangle$.

7.3. Action of $\text{Aut}(FCD(\mathscr{P}))$ on $\mathscr{M}$

Let $\mathscr{P}$ be a Promela program. In order to show how an element of $\text{Aut}(FCD(\mathscr{P}))$ acts on states of the Kripke structure associated with $\mathscr{P}$, we must define the set AP of atomic propositions for a Promela program. Let *Loc* be the set of local variables, *Glob* the set of global variables, and *Chan* the set of channels of $\mathscr{P}$. Let D be the set of data values for the program. To denote a local variable of a process with process id i we write x_i where x is the name of the variable. If x_i is a local variable of process i, and if processes i and j have the same process type, then x_j is the corresponding local variable of process j.

Let $AP_{\text{local}} = \{(x_i = val) : x_i \in Loc, val \in D\}$, the set of propositions relating to local variables, and define AP_{global} and AP_{channel}, the set of propositions relating to global variables and channels respectively, similarly. Then $AP = AP_{\text{local}} \cup AP_{\text{global}} \cup AP_{\text{channel}}$. The underlying Kripke structure $\mathscr{M}$ over AP for the program $\mathscr{P}$ is generated by exploring all possible behaviours of $\mathscr{P}$. States of $\mathscr{M}$ are uniquely identified by a labelling of atomic propositions. Note that each process in $\mathscr{P}$ has its own

program counter variable which indicates the statements which may be executed in the next transition. Thus two states, for which all other variables are assigned identical values, may be distinguished due to assignments of the associated program counters.

For an element $\alpha \in \mathrm{Aut}(FCD(\mathscr{P}))$ we define a corresponding mapping α^* which is a permutation of the Kripke structure $\mathscr{M}$ underlying $\mathscr{P}$. If $val \in D$ has type *chan*, i.e., $val = link_i$ for some $1 \leqslant i \leqslant n$, then $\alpha(val) = link_{\alpha(i)}$. For any $s \in S$, let $L(\alpha^*(s)) = \{\alpha(p) : p \in L(s)\}$. For a proposition $p \in AP$, the proposition $\alpha(p)$ is defined as follows:

If $p = (x_i == val) \in AP_{\mathrm{local}}$ for some $x_i \in Loc$, where the type of x_i is *pid* or *chan*, then $\alpha(p) = (x_{\alpha(i)} == \alpha(val))$, otherwise $\alpha(p) = (x_{\alpha(i)} == val)$. If $p = (x == val) \in AP_{\mathrm{global}}$ for some $x \in Glob$, where the type of x is *pid* or *chan*, then $\alpha(p) = (x == \alpha(val))$, otherwise $\alpha(p) = p$. If $p = (link_i[j] == msg) \in AP_{\mathrm{channel}}$, i.e., *msg* is at position j on channel $link_i$, then $\alpha(p) = (link_{\alpha_i}[j] == \alpha(msg))$. Here α acts on *msg* by permuting the value of each field of *msg* which has type *pid* or *chan*, and leaving all other fields unchanged.

7.4. Correspondence between $\mathrm{Aut}(FCD(\mathscr{P}))$ and $\mathrm{Aut}(\mathscr{M})$

The following theorem shows that if $\mathscr{P}$ is a Promela specification of a featured network, generated by our template-based approach, we can derive a symmetry group for the Kripke structure $\mathscr{M}$ underlying $\mathscr{P}$ from a symmetry group for the feature configuration diagram, $FCD(\mathscr{P})$. We sketch the proof, which is analogous to a similar result for automatic symmetry detection by static channel diagram analysis [17].

Theorem 3. *Let* $\alpha \in Aut(FCD(\mathscr{P}))$. *Then* $\alpha^* \in Aut(\mathscr{M})$.

Proof. Without loss of generality, assume that all statements of $\mathscr{P}$ have the form

guard $\rightarrow$ *update*

Let $(s,t) \in R$, and suppose this transition is fired by statement $g \rightarrow u$ in $\mathscr{P}$. Let $\alpha(g)$ be the guard obtained from g by replacing each occurrence of a literal *pid* value i or channel reference link $link_i$, with the value $\alpha(i)$ or $link_{\alpha(i)}$ respectively. Define the update $\alpha(u)$ similarly. Clearly, applying the update $\alpha(u)$ to the state $\alpha^*(s)$ leads to the state $\alpha^*(t)$. Similarly $\alpha(g)$ is executable in state $\alpha^*(s)$, since it is executable in state s and α belongs to $\mathrm{Aut}(FCD(\mathscr{P}))$ and so preserves the truth of boolean expressions over feature arrays. The statement $\alpha(g) \rightarrow \alpha(u)$ is in $\mathscr{P}$ due to the open symmetry of $\mathscr{P}$, which is guaranteed by the template-based approach. Thus the statement $\alpha(g) \rightarrow \alpha(u)$ is enabled in state $\alpha^*(s)$, and fires the transition $(\alpha^*(s), \alpha^*(t))$. It follows that $(\alpha^*(s), \alpha^*(t)) \in R$, and hence $\alpha^* \in \mathrm{Aut}(\mathscr{M})$. □

8. Implementation

For symmetry reduction, FSG uses the *saucy* program [16] and the GAP system [23] to compute $\mathrm{Aut}(FCD(\mathscr{P}))$. We could not find a graph automorphism computation package to work directly with graphs that have coloured edges, so in practice $\mathrm{Aut}(FCD(\mathscr{P}))$ is computed (using GAP) as the intersection over the set of binary features of groups $\mathrm{Aut}(R(B), C_V)$. Here $\mathrm{Aut}(R(B), C_V)$ is the group which preserves the relation $R(B)$ as well as the vertex colouring C_V.

We have also developed a prototype symmetry reduction package for the SPIN model checker. The package is based on an existing symmetry reduction package, SymmSpin [3], but whereas SymmSpin requires symmetries to be specified using *scalarsets* (a purpose-built data type for symmetry reduction), our system supports automatic symmetry detection by static channel diagram analysis [19], or, in this case, by feature configuration diagram analysis as described above. During search, orbit representatives are computed in a standard way by sorting the vector associated with a state. This representative computation technique has been used in a variety of approaches to symmetry reduction [3,11,21].

9. Experimental results for symmetry reduction

In this section we demonstrate the effectiveness of symmetry reduction using FSD for the email example. We consider email specifications with a varying number of *client* components. For each specification size, we consider the case where components are unfeatured, and the case where components 1 and 2 are featured as in Section 7.1. In Table 1 we show, for each specification size, the sizes of the state spaces associated with both unfeatured and featured networks, together with the sizes of the corresponding reduced state spaces when symmetry reduction is applied. We give the time taken for verification

Table 1
Experimental results

#Clients	Unfeatured				
	#States (orig.)	Time (orig.)	$\|G\|$	#States (red.)	Time (red.)
3	23,256	0.1	6	3,908	0.2
4	852,641	9	24	38,560	2
5	$3.04 \times 10^{7\dagger}$	3576	120	315,323	40
6	1.5×10^{9} (E)	–	720	2.3×10^{6}	576
7	6.9×10^{10} (E)	–	5040	1.53×10^{7}	6573
	Featured				
3	46,151	0.2	1	n/a	n/a
4	2.3×10^{6}	33	2	1.2×10^{6}	21
5	9.5×10^{7} (E)	–	6	1.75×10^{7}	1160

in seconds. In each case, the size of the symmetry group computed by FSG is given. For configurations for which verification proved intractable an estimate (denoted "(*E*)") is given for the number of number of states and the time omitted (indicated by "-"). Entries marked "*n/a*" indicate that symmetry reduction is not applicable, due to a trivial symmetry group.

By default, no compression was used during search. For large models, the *collapse* compression technique provided by SPIN [25] was used (indicated by †), which results in slower verification. All experiments were performed on a PC with a 2.4 GHz Intel Xeon processor, 3Gb of available main memory, running Red Hat Linux, with SPIN version 4.2.3.

Notice that in the unfeatured case with three client processes, adding symmetry reduction results in fewer states but longer verification time. Exploiting symmetry carries a time overhead due to the conversion of states to their representatives. When the unreduced state space is small, the overhead can result in an increase in search time when symmetry reduction is applied.

If $|M|$ and $|M_G|$ are the number of states of the unreduced and reduced models respectively, then if $|G|$ is the size of the associated symmetry group, $|M| \leqslant |M_G| \cdot |G|$ (as each state of M_G represents at most $|G|$ states of M). In all of the experiments where it was possible to find $|M|$ (and where there was a non-trivial symmetry group), it was found to be between eighty and ninety six percent of this upper bound. Therefore, in cases where it is impossible to generate M, an estimate is given at approximately the middle of this range (ninety percent of the upper bound).

As expected, adding features to a specification considerably reduces the size of the symmetry group associated with the underlying model. Additionally, the *autorespond* feature increased the size of the state space in all cases quite dramatically. Nevertheless, applying symmetry reduction to featured specifications where there are several identically featured (in this case, unfeatured) components leads to large savings in both memory requirements and verification time. For large examples it proved possible to generate the reduced state space when the original state space was intractably large.

If two features do not interact, for example message filtering is not affected by the *autorespond* feature [7], then an exhaustive search of the state space associated with a specification is required. These experimental results show that state space reduction by symmetry can be extremely useful in such cases.

10. Scalability of our approach

Our abstraction/induction approach is not limited to systems with one or two component types. We can extend the approach to any system in which there are a small number of components types (but potentially large numbers of components of each type). For example we believe our approach is to be eminently applicable to the verification of SIP networks [33], which consist of *end user devices* and different type of *server* components, and to Web Services [46]. The investigation of the applicability of our approach in these cases (e.g., the determination as to when such systems are *balanced*), is the subject of future work.

Symmetry reduction techniques, by their nature, allow one to apply model checking techniques to some systems which, in unreduced form, are not verifiable. However, even in systems which clearly

contain inherent symmetry, the application of reduction methods is usually ad hoc and time consuming. Our template-based approach allows us to simply and automatically apply symmetry reduction to models of systems as they are developed.

As new featured domains emerge, our techniques will allow us to rapidly and systematically produce Promela models to which both abstraction/induction and symmetry reduction can be applied for efficient feature interaction analysis using model checking.

11. Related work

Model checking for feature interaction analysis has been investigated by others, notable approaches are those using COSPAN [22], Caesar [45], SMV [40], SPIN (the FeaVer project) [26,27,44] and a bespoke tool [30]. None of these studies generalise results to more than three or four users.

As we have discussed in Section 2.3.2, the invariant approach to parameterised model checking problem has been applied in many contexts [4,13,34]. However, none of these address the feature interaction problem.

Symmetry reduction in model checking is a common technique. However, in most cases symmetries of a model are either known *a priori* [12], or are coded into the model through the use of special keywords [3,29]. Both approaches require the modeller to provide information on the presence of symmetry in a model. Our automatic symmetry detection method allows us to infer symmetries of the state space underlying a model *without* explicitly constructing the state space.

Symmetry reduced model checking for feature interaction detection is considered in [39], where permutation symmetry is used to construct a reduced *symmetric reachability graph*, which is similar to a quotient Kripke structure. Their approach is only applicable to cases where *all* of the users subscribe to *all* of the features currently being analysed, and as a result their unreduced models grow even faster than ours and the automorphism group is the group of all permutations of the user ids. For this reason, no symmetry detection is required. We believe that our approach is more realistic and adaptable.

As far as we are aware, we are the first to develop a systematic technique for the construction of specifications which are amenable to both inductive analysis and symmetry reduction methods.

12. Conclusions

Model checking is a popular automated technique for reasoning about networks of components; it is often applied to the problem of detecting interactions between featured components. But, it suffers from the well known problem of state space explosion.

Abstraction is key to reducing the state space. Two common abstraction approaches are *induction by invariant* (to encapsulate the behaviour of a system of any size) and *symmetry reduction* (to encapsulate the behaviour of a group of permutations by a representative). While these two approaches are generally considered to be orthogonal, we have found that they are related. In particular, they are both applicable under similar circumstances. We encapsulate these circumstances by a property of the system specification: *open symmetry*.

Essentially, this property constrains the way components refer to other components in the system. The constraints are reasonably intuitive and not overly restrictive.

Our main result is a template for producing components that are open symmetric, and a new graphical representation for the entire system, called the *feature configuration diagram*. Any (safely featured parameterised) specification thus produced is immediately amenable to state reduction by induction (using the invariant method), and *any* generated specification is applicable for symmetry reduction (using an automorphism group derived from the feature configuration diagram). We believe that this represents a novel application of symmetry detection (and thus reduction) for featured networks. The template is defined for the specification language Promela, but the approach is applicable to other specification formalisms. Throughout, the techniques are illustrated by application to an example featured network: *email*.

Acknowledgements

The first and third authors would like to thank the Glasgow University John Robertson Bequest and the Carnegie Trust for funding this research.

References

[1] D. Amyot, L. Logrippo (Eds.), Feature Interactions in Telecommunications and Software Systems VII, Ottawa, Canada, June, IOS Press, 2003.

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

[3] D. Bosnacki, D. Dams, L. Holenderski, Symmetric spin, International Journal on Software Tools for Technology Transfer 4 (1) (2002) 65–80.

[4] M. Browne, E. Clarke, O. Grumberg, Characterizing finite Kripke structures in propositional temporal logic, Theoretical Computer Science 59 (1988) 115–131.

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

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

[7] M. Calder, A. Miller, Generalising feature interactions in email, in: Amyot and Logrippo [1], pp. 187–205.

[8] M. Calder, A. Miller, Detecting feature interactions: how many components do we need? in: Mark Ryan, Dieter Ehrich, John-Jules Meyer (Eds.), Objects, Agents and Features, Lecture Notes in Computing Science, Springer-Verlag, 2004, pp. 45–66.

[9] M. Calder, A. Miller, An automatic abstraction technique for verifying featured, parameterised systems, Theoretical Computer Science, in press.

[10] M. Calder, A. Miller, Feature interaction detection by pairwise analysis of LTL properties – a case study, Formal Methods in System Design 28 (3) (2006) 213–261.

[11] E. Clarke, E. Emerson, S. Jha, A. Sistla, Symmetry reductions in model-checking, in: A. Hu, M. Vardi (Eds.), Proceedings of the 10th International Conference on Computer-aided Verification (CAV'98), Vancouver, British Columbia, Canada, June/July, Lecture Notes in Computer Science, vol. 1427, Springer-Verlag, 1998, pp. 147–158.

[12] E. Clarke, R. Enders, T. Filkhorn, S. Jha, Exploiting symmetry in temporal logic model checking, Formal Methods in System Design 9 (1–2) (1996) 77–104.

[13] E. Clarke, O. Grumberg, S. Jha, Verifying parameterized networks using abstraction and regular languages, in: Insup Lee, Scott A. Smolka (Eds.), Proceedings of the 6th International Conference on Concurrency Theory (CONCUR '95), Philadelphia, PA., August, Lecture Notes in Computer Science, vol. 962, Springer-Verlag, 1995, pp. 395–407.

[14] E. Clarke, O. Grumberg, D. Peled, Model Checking, The MIT Press, Cambridge, MA, 1999.

[15] S. Creese, A. Roscoe, Formal verification of arbitrary network topologies, in: H.R. Arabnia (Ed.), Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas, NV, USA, June–July, vol. II, CSREA Press, 1999.

[16] P. Darga, M. Liffiton, K. Sakallah, I. Markov, Exploiting structure in symmetry detection for CNF, in: Proceedings of the 41st Annual Conference on Design Automation, San Diego, CA, USA, ACM Press, 2004, pp. 530–534.

[17] A. Donaldson, A. Miller, M. Calder, Finding symmetry in models of concurrent systems by static channel diagram analysis, Electronic Notes in Theoretical Computer Science 128 (6) (2005) 161–177.

[18] A. Donaldson, A. Miller, M. Calder, SPIN-to-GRAPE a tool for analysing symmetry in Promela models, Electronic Notes in Theoretical Computer Science 139 (1) (2005) 3–23.

[19] A.F. Donaldson, A. Miller, Automatic symmetry detection for model checking using computational group theory, in: J. Fitzgerald, I. Hayes, A. Tarlecki (Eds.), Proceedings of the 13th International Symposium on Formal Methods (FM 2005), Newcastle, UK, July, Lecture Notes in Computer Science, vol. 3582, Springer-Verlag, 2005, pp. 481–496.

[20] E. Emerson, V. Kahlon, Reducing model checking of the many to the few, in: David A. McAllester (Ed.), Automated Deduction – Proceedings of the 17th International Conference on Automated Deduction (CADE 2000), Pittsburgh, PA, USA, June, Lecture Notes in Computer Science, vol. 1831, Springer-Verlag, 2000, pp. 236–254.

[21] E. Emerson, T. Wahl, Dynamic symmetry reduction, in: N. Halbwachs, L. Zuck (Eds.), Proceedings of the 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2005), Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April, Lecture Notes in Computer Science, vol. 3440, Springer-Verlag, 2005, pp. 382–396.

[22] A. Felty, K. Namjoshi, Feature specification and automatic conflict detection. In Calder and Magill [5], pp. 179–192.

[23] Gap Group, GAP – Groups Algorithms and Programming, Version 4.2. Aachen, St. Andrews, 1999. http://www-gap.dcs.st-and.ac.uk/gap.

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

[25] G. Holzmann, State compression in Spin: recursive indexing and compression training runs, in: R. Langerak (Ed.), Proceedings of the 3rd SPIN Workshop (SPIN'97), Twente University, The Netherlands, April 1997.

[26] G. Holzmann, M. Smith, A practical method for the verification of event-driven software, in: Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, CA, USA, May, ACM Press, 1999, pp. 597–607.

[27] G. Holzmann, M. Smith, Software model checking – extracting verification models from source code, in: J. Wu, S. Chanson, Q. Gao (Eds.), Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV'99), Beijing, China, October, International Federation For Information Processing, vol. 156, Kluwer, 1999, pp. 481–497.

[28] C. Norris Ip, David L. Dill, Verifying systems with replicated components in Murϕ, Formal Methods in System Design 14 (1999) 273–310.

[29] C. Norris Ip, D. Dill, Better verification through symmetry, Formal Methods in System Design 9 (1996) 41–75.

[30] B. Jonsson, T. Margaria, G. Naeser, J. Nystroem, B. Steffen, Incremental requirement specification for evolving systems, in: Calder and Magill [5], pp. 145–162.

[31] Y. Keston, A. Pnueli, E. Shahar, L. Zuck, Network invariants in action, in: L. Brim, P. Jancar, M. Kretínský, A. Kucera (Eds.), Proceedings of the 13th International Conference on Concurrency Theory (CONCUR 2002),

Brno, Czech Republic, August, Lecture Notes in Computer Science, 2421, Springer-Verlag, pp. 101–115.
[32] K. Kimbler, L. Bouma (Eds.), Feature Interactions in Telecommunications and Software Systems V, IOS Press, Amsterdam, 1998.
[33] M. Kolberg, E.H. Magill, Detecting feature interactions between SIP call control services, in: Reiff-Marganiec and Ryan [41], pp. 147–162.
[34] R.P. Kurshan, K.L. McMillan, A structural induction theorem for processes, in: Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, ACM Press, 1989, pp. 239–247.
[35] K. McMillan, S. Qadeer, J. Saxe, Induction in compositional model checking, in: E. Emerson, A. Sistla (Eds.), Proceedings of the 12th International Conference on Computer-aided Verification (CAV 2000), Chicago, IL, USA, July, Lecture Notes in Computer Science, vol. 1855, Springer-Verlag, 2000, pp. 312–327.
[36] S. Merz, Model checking: a tutorial overview, in: F. Cassez, C. Jard, B. Rozoy, M. Ryan (Eds.), Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June, Lecture Notes in Computer Science, vol. 2067, Springer-Verlag, 2000, pp. 3–38.
[37] A. Miller, M. Calder, A generic approach for the automatic verification of featured, parameterised systems, in: Reiff-Marganiec and Ryan [41], pp. 217–235.
[38] M. Müller-Olm, D. Schmidt, B. Steffen, Model-checking: a tutorial introduction, in: A. Cortesi, G. File (Eds.), Proceedings of the 6th International Static Analysis Symposium (SAS'99), Venice, Italy, September, Lecture Notes in Computer Science (LNCS), vol. 1694, Springer-Verlag, 1999, pp. 330–354.
[39] M. Nakamura, Y. Kakuda, T. Kikuno, Feature interaction detection using permutation symmetry, in: Kimbler and Bouma [32], pp. 187–201.
[40] M. Plath, M. Ryan, Plug-and-play features, in: Kimbler and Bouma [32], pp. 150–164.
[41] S. Reiff-Marganiec, M. Ryan (Eds.), Proceedings of the 8th international conference on Feature Interactions in Telecommunications and Software Systems VIII, Leicester, UK, June, IOS Press, Amsterdam, 2005.
[42] A. Roychoudhury, I.V. Ramakrishnan, Inductively verifying invariant properties of parameterized systems, Automated Software Engineering 11 (2) (2004) 101–139.
[43] P. Saffrey, Optimising communication structure for model checking. Ph.D. thesis, Department of Computing Science, University of Glasgow, July 2003.
[44] M. Smith, G. Holzmann, K. Etessami, Events and constraints: a graphical editor for capturing logic requirements of programs, in: Proceedings of the 5th IEEE International Symposium on Requirements Engineering, Toronto, Canada, August, IEEE Computer Society, 2001, pp. 14–22.
[45] M. Thomas, Modelling and analysing user views of telecommunications services, in: P. Dini, R. Boutaba, L. Logrippo (Eds.), Feature Interactions in Telecommunication Networks IV, June, IOS Press, Amsterdam, 1997, pp. 168–182.
[46] M. Weiss, Feature interactions in web services, in: Amyot and Logrippo [1], pp. 149–158.

Alice Miller, MIEE, CEng, is a lecturer in the Department of Computing Science, University of Glasgow. Her research interests include Combinatorics and Group Theory as well as Model Checking. Recently her work has involved developing abstraction, induction and symmetry reduction techniques for application to model checking, and the formal verification of sensor networks. She has worked at the Universities of Western Australia, East Anglia, Stirling and Glasgow, and was awarded a Daphne Jackson Fellowship in 1999. Alice has a BSc and Ph.D in Mathematics from the University of East Anglia.

Muffy Calder, FIEE, FRSE, is Head of the Department of Computing Science and Professor of Computing Science, University of Glasgow. Her research is in modelling and reasoning about the behaviour of complex software and biochemical systems using mathematics and automated reasoning tools. Her research interests include concurrent systems, process algebras and stochastic process algebras, model checking, protocol and service description languages, protocol analysis, and safety critical and biomedical applications. She has led numerous externally funded research projects and co-chaired an international conference on feature interactions. She is a member of the Scottish Science Advisory Committee, reporting to the Scottish Executive. She has long-standing industrial collaborations with many world-leading IT companies and has been a research fellow at BT Laboratories and DEC in California. She is a member of the IEE (Institution of Electrical Engineers) research policy group. Professor Calder has a PhD in Computational Science from the University of St. Andrews and a BSc in Computing Science from the University of Stirling.

Alastair Donaldson is completing a Ph.D. in Computing Science at the University of Glasgow, funded by the Carnegie Trust for the Universities of Scotland. His Ph.D. work has involved the development of novel automatic techniques for symmetry detection and exploitation in model checking, with implementations for the explicit-state model checker SPIN and the probabilistic, symbolic model checker PRISM. His interest in software engineering has led to employment with Reuters and Graham Technology. Alastair recently co-chaired an international workshop on symmetry in constraint satisfaction problems. When not studying Computing Science, he plays drums for his rock band, Latonic. Alastair graduated with a BSc in Computing Science and Mathematics from the University of Glasgow in 2003.