Theoretical Computer Science 404 (2008) 235–255

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

An automatic abstraction technique for verifying featured, parameterised systems

M. Calder, A. Miller *

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

A B S T R A C T

A general technique combining model checking and abstraction is presented that allows property based analysis of systems consisting of an *arbitrary* number of featured components. We show how parameterised systems can be specified in a *guarded command* form with constraints placed on variables which occur in guards. We prove that results that hold for a small number of components can be shown to scale up. We then show how featured systems can be specified in a similar way, by relaxing constraints on guards. The main result is a generalisation theorem for featured systems which we apply to two well known examples.

1. Introduction

Model-checking is a popular and effective technique for reasoning about distributed, concurrent systems, particularly networks of communicating components. But, there is a limitation – only a single, tractable model can be checked. In this paper we consider the problem of how to relate an individual model checking result about a system of fixed size and configuration, to the general case. Namely, does a result for a given system scale to a system of any size – can we leverage a general result from a specific one? This question cannot be answered by model-checking alone because it is an example of the well known *parameterised model checking problem* (PMCP) which is, in general, undecidable [3]. But, for some classes, we can find a model-checking solution. This paper introduces a model-checking solution for systems of communicating components. The constraint is that the components fulfill criteria which allow them to be safely abstracted. We call this *safe* with respect to abstraction.

An example follows. We can prove a property ϕ, say, holds for a model of a system with 3 concurrent components, p_0, p_1 p_2 i.e. $\mathcal{M}(p_0||p_1||p_2) \models \phi$.

Now consider the question, given another component p_3, under what conditions does $\mathcal{M}(p_0||p_1||p_2||p_3) \models \phi$ hold? More generally, given a finite number of further components, under what conditions does the property still hold? How can we *leverage* the proof of the property for the system of fixed size (i.e. for 3 components) to the proof of the more general case? Moreover, when would the property *not* hold?

To answer these questions, there are a number of aspects to consider

- what is the form of ϕ? Can it refer to propositions about any local or global variable, or variables indexed by any component?
- what is the communication topology of the system? Can the components communicate peer to peer, or in fixed a topology such as a star or hypercube?
- what is the relationship between components? Must they be isomorphic? If not, what are the constraints on the behaviour of the components?

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

0304-3975/$ – see front matter
doi:10.1016/j.tcs.2008.03.034

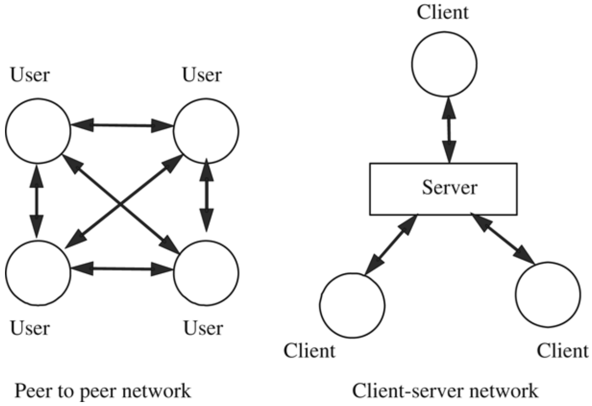

Fig. 1. Example networks.

To illustrate all of these points, consider two paradigms: a network of peer to peer *User* components and a network of *Client* components with a single *Server* component (see Fig. 1). Suppose we can show that, in the former paradigm with four *User* components, if two *User* components have established each other as partner they will eventually become connected. Would the result hold if there were five *User* components in the network? Would the result hold if the property referred to specific *User*s, for example it stated that *User* 1 could eventually be connected to *User* 5? Clearly the result would not hold for systems of less than six components. Similarly, suppose we can show, in the second paradigm with three *Client* components, that a message sent to the *Server* will eventually be delivered to its destination. Would the same be true if there were more *Client* components? What if the *Client*s had different behaviour? For example, would the property still hold if some of the *Client*s had a forwarding capability, or some of the *Client*s had the ability to invoke a forwarding capability on the destination *Client*? We would expect the former to be true, but not necessarily the latter.

The aim of our approach is a technique which makes these aspects explicit. The approach relies on partitioning components into two distinct subsets: *concrete* components and *abstract* components. The former are the components involved in fixed system analysis, i.e. the components p_0, p_1, p_2 above. Abstract components are the remaining components in systems of larger or arbitrary size. For example, p_3, or more generally, $p_3 \ldots, p_{n-1}$ are the abstract components. The property ϕ can only refer to global variables, or variables indexed by concrete components. Concrete and abstract components do *not* need to be isomorphic, but abstract components must be *safe* with respect to the abstraction in the sense that their presence or otherwise does not affect the underlying behaviour of the overall system, with respect to a given property. The topology is assumed to be either static and regular, or dynamic and peer to peer (fully connected). There is one communication channel associated with each component.

The main contribution of this paper is to define an abstraction and prove that *basic* components and components with certain categories of features which conform to syntactic criteria are safe with respect to our abstraction.

1.1. Overview of paper

In the next section we review background material, e.g. parameterised systems, features, Kripke structures, temporal logics and model checking. In Section 3 we give an overview of our approach to solving PMCP by abstraction and introduce the concept of a safe component. In Section 4 the approach is developed in more detail for basic parameterised systems. We apply the techniques to two example systems: peer to peer telephony and client–server email and demonstrate that the components in these systems are safe with respect to the abstraction. In Section 5 we extend the abstraction approach to featured systems. We extend the two examples to more complex ones with features. We show that when features conform to certain syntactic criteria, components are still safe, thus we can again solve PMCP. In Section 6 we discuss the implications of our approach for failed formulae, i.e. what can we conclude when a property fails to be satisfied. Automation and experimental results are discussed in Section 7, related work is discussed in Section 8 and conclusions are given in Section 9.

2. Background

2.1. Parameterised systems and network invariants

The type of system we are interested in is *parameterised, concurrent systems*. A parameterised system has the form $S_n = p_0||p_1|| \cdots ||p_{n-1}$ or $S_n = C||p_0||p_1|| \cdots ||p_{n-2}$ where $p_0, p_1, \ldots, p_{n-1}$ are instantiations of the same parameterised process p, and C a distinguished *context* process (sometimes called an *environment* process) — for example a *hub* or *server* process. $||$ is parallel composition. The verification of such systems — that is, the proof that properties hold for such systems for *any* value of n greater than some lower bound n_0, is both challenging and important. Parameterised systems occur frequently — in distributed algorithms for example.

It is not possible to verify such systems (for any n) using model checking alone [3]. However, one approach that has proved successful for verifying some parameterised systems involves the construction of a *network invariant* [6,29,15]. The network invariant I represents an arbitrary member of the family $\mathcal{F} = \{S_n : n \geq n_0\}$ and proof of a given property ϕ for I can be shown to imply that any member of the family $\mathcal{F}$ satisfies ϕ.

Some other techniques that have been used to verify parameterised systems include those based on theorem proving [17], on abstraction [27], or on a combination of the two [28]. A further method is to use explicit inductive techniques combined with model checking [23,20,32,35].

We introduce an invariant-based approach which combines abstraction and induction to verify parameterised systems. 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 invariant processes can be constructed in practice, to extend results proved for small, fixed sized models, and to results which hold for models of any size.

We show how our approach can be extended to systems in which components are still expressed in a well-defined way, but individual components may be distinguished by way of *features*.

Like all network invariant approaches, this approach is limited to systems with regular topology, which grow in a regular way as the number of components increases. The example networks we consider have either a peer to peer topology (a telephone system) or a client–server topology (email). We choose asynchronous communication to reflect realistic systems, and allow dynamic communication (channels are passed on channels).

2.2. Features

Network components 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 basic call behaviour. Features fundamentally affect basic behaviour in different ways, and so components with features are not, in general, isomorphic. Moreover, features associated with one component can affect the behaviour of other (possibly featured) components.

A parameterised component is said to *subscribe* to a feature f (belonging to a given set of features), and a parameterised system $S_n = p_0||p_1|| \cdots ||p_{n-1}$ (or $C||p_0||p_1|| \cdots ||p_{n-2}$) is *featured* when (at least one of) the components $p_0, p_1, \ldots, p_{n-1}$ (or C, $p_0, p_1, \ldots, p_{n-2}$) subscribes to at least one feature.

2.3. Temporal logic

We provide a description of the syntax and semantics of the logics CTL^* and LTL. We use LTL to define our particular properties of simple telephone and email systems in Section 4.4.

Logic CTL^* is defined as a set of state formulas, where the CTL^* state and path formulas are defined inductively below. Quantifiers A and E are used to denote *for all paths*, and *for some path* respectively (where $E\phi = \neg A \neg \phi$). In addition, X, $\cup$, $\langle\rangle$ and $[]$ represent standard *nexttime*, *strong until*, *eventually* and *always* operators (where $\langle\rangle\phi = true \cup \phi$ 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 state formula
- if ϕ and ψ are state formulas, then so are $\neg\phi$, $\phi \wedge \psi$ and $\phi \vee \psi$
- if ϕ is a path formula, then $A\phi$ and $E\phi$ are state formulas
- any state formula ϕ is also a path formula
- if ϕ and ψ are path formulas, then so are $\neg\phi$, $\phi \wedge \psi$ and $\phi \vee \psi$, $X\phi$, $\phi \cup \psi$, $\langle\rangle\phi$ and $[]f$.

Logic LTL is obtained by restricting the set of (CTL^*) formulas to those of the form $A\phi$, where ϕ does not contain A or E. 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 $\mathcal{M}$, if the CTL^* formula ϕ holds at a state $s \in S$ then we write $\mathcal{M}, s \models \phi$ (or simply $s \models \phi$ when the identity of the model is clear from the context). The relation $\models$ is defined inductively below. Note that for a path $\pi = s_0, s_1, \ldots$, starting at s_0, $first(\pi) = s_0$ and, for all $i \geq 0$, π_i is the suffix of π starting from state s_i.

- $s \models p$, for $p \in AP$ if and only if $p \in L(s)$
- $s \models \neg\phi$ if and only if not $s \models \phi$ $s \models \phi \wedge \psi$ if and only if $s \models \phi$ and $s \models \psi$, and $s \models \phi \vee \psi$ if and only if $s \models \phi$ or $s \models \psi$
- $s \models A\phi$ if and only if $\pi \models \phi$ for every path π starting at s
- $\pi \models \phi$, for any state formula ϕ, if and only if $first(\pi) \models \phi$
- $\pi \models \neg\phi$ if and only if not $\pi \models \phi$ $\pi \models \phi \wedge \psi$ if and only if $\pi \models \phi$ and $\phi \models \psi$, and $\pi \models \phi \vee \psi$ if and only if $\pi \models \phi$ or $\phi \models \psi$
- $\pi \models \phi \cup \psi$ if and only if, for some $i \geq 0$, $\pi_i \models \psi$ and $\pi \models \phi$ for all $0 \leq j \leq i$
- $\pi \models X\phi$ if and only if $\pi_1 \models \phi$
- $\pi \models \langle\rangle\phi$ if and only if $\pi_i \models \phi$, for some $i \geq 0$
- $\pi \models []\phi$ if and only if $\pi_i \models \phi$, for all $i \geq 0$.

2.4. Kripke structures

Model checking involves checking *Kripke structures* [14] 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 $L : S \to 2^{AP}$ 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. That is, we assume that $S_0 = \{s_0\}$. We write $\mathcal{M} \models \phi$ to represent $s_0 \models \phi$. We also assume that the transition is *total*, that is, for all $s \in S$ there is some $s' \in S$ such that $(s, s') \in R$.

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 $\mathcal{M} \preceq \mathcal{M}'$.

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

Lemma 3. *Suppose that $\mathcal{M} \preceq \mathcal{M}'$. Then for every LTL formula ϕ with atomic propositions in AP', $\mathcal{M}' \models \phi$ implies $\mathcal{M} \models \phi$.*

2.5. Symmetry groups

In this section we summarise some definitions from group theory which we will use to define *open symmetric* components in Section 4.

Definition 4. 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 $x \in G$ there is an inverse element $x^{-1} \in G$ such that $x \circ x^{-1} = x^{-1} \circ x = 1_G$.

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.

Definition 5. Let X be a finite set. A permutation of X is a bijection from X to X. The set of all permutations of X, $Sym(X)$, forms a group under composition of mappings. For any $x \in X$, and any $\alpha \in Sym(X)$, we denote the image of x under α by $\alpha(x)$.

2.6. Promela and Spin

Promela is an imperative 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 bespoke model-checker for Promela and provides several reasoning mechanisms: assertion checking, acceptance and progress states and cycle detection, and satisfaction of temporal properties.

Given a Promela parameterised system system with form $S_n = p_0||p_1|| \cdots ||p_{n-1}$ (or $C||p_0||p_1|| \cdots ||p_{n-2}$), the associated model, or Kripke structure, is denoted by $\mathcal{M}_n$. In order to perform verification on a model, Spin translates each process template into a finite automaton and then computes an asynchronous interleaving product of these automata to obtain the global behaviour of the concurrent system. This interleaving product is referred to as the *state-space*.

As well as enabling a search of state-space to check for deadlock, assertion violations etc., Spin allows checking of the satisfaction of an LTL formula over all execution paths. The mechanism for doing this is via *never claims* – processes which describe *undesirable* behaviour, and Büchi automata – automata that accept a system execution if and only if that execution forces it to pass through one or more of its accepting states infinitely often [26,24]. Checking satisfaction of a formula involves a depth-first search of the synchronous product of the automaton corresponding to the concurrent system (model) and the Büchi automaton corresponding to the never-claim.

Note that in Promela, the symbol '!' is used to denote negation. We use this form when referring to *LTL* properties, or propositions in Promela.

2.7. Guarded command form

For reasoning purposes, we require to assume that components are defined in a given, well defined way. Namely, we assume the *guarded command*, *GC*, form which consists of one, global loop over a choice of statements of the form *guard* $\to$ *command*. Guards will be over-lapping when the system behaviour is non-deterministic. Precise definition of the form depends upon the specification language; we have defined it for Promela. In fact, we assume that each component

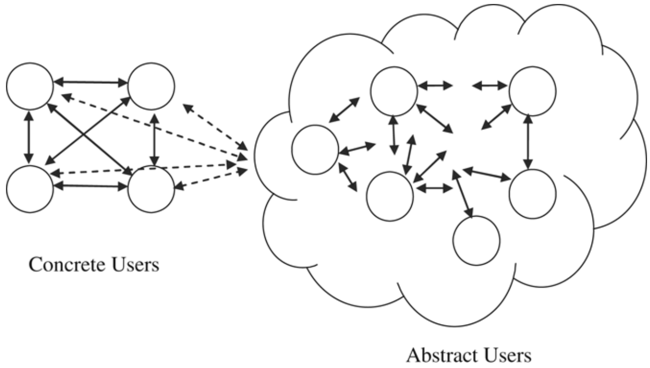

Fig. 2. Peer to peer abstraction.

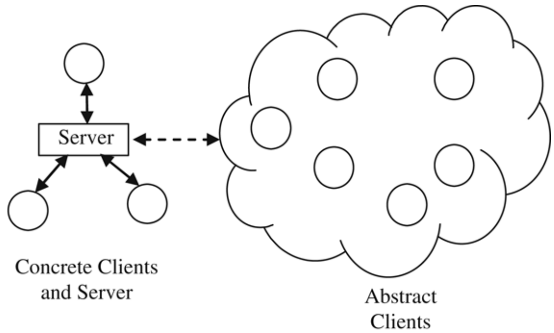

Fig. 3. Client-server abstraction.

type is defined within a process (specifically a *proctype* declaration) and (modulo initial variable set up) the proctype definitions themselves have guarded command form. In the Promela form, we add additional program counter variables, *p_c*, to represent local program control. We note that in some model checking tools (e.g. Murϕ [18] and SMV [33]), models are specified directly in this form.

Some examples of programs expressed in this modular guarded command form are given in Section 4.3. Note, the Promela *do . . . od* construct provides a way of expressing a loop in which commands are repeatedly selected non-deterministically until a *break* statement is executed (there are no *break* statements in our examples). Choices are denoted :: *statement*. In addition, Promela allows us to group together statements that should be executed *at the same time* (i.e. before another component executes a transition) using an *atomic* statement. We will henceforth therefore assume that statement choices are expressed thus:

$$:: \textit{atomic}\{\textit{guard} \rightarrow \textit{command}\}.$$

We assume that atomic statements can not block (strictly, they can block on the first statement). This means that if a statement choice has a command which involves writing to (reading from) a channel (*chan* say), we must be sure that *chan* is not full (empty). Thus the corresponding guard must include the proposition $\textit{nfull}(\textit{chan})$ ($\textit{nempty}(\textit{chan})$).

3. The abstraction approach and safe components

3.1. Abstraction of parameterised systems

Given a parameterised system S_n of size n, with associated model $\mathcal{M}_n$, and a fixed m ($1 \leq m \leq n$), we partition system components into m *concrete* components and n-m *abstract* components. We encapsulate the observable behaviour of abstract components, with respect to to a given property, by a new component called *Abs* and replace all abstract components by *Abs*. Since the concrete components may communicate directly with abstract components, we may need to modify the communication to/from concrete components. The new abstract system is $p'_0||\cdots||p'_{m-1}||Abs$ (or, when there is a context component, $C'||p'_0||\cdots||p'_{m-2}||Abs$) where ($C'$ and) the p'_i denote suitably altered concrete components. The associated model is denoted by $\mathcal{M}^m_{abs}$.

We illustrate the abstraction approach for a peer to peer network, and a client–server network in Figs. 2 and 3 respectively.

3.2. Safe components

Before we describe our abstraction approach in detail, we define what is meant by a *safe* component with respect to our abstraction. Assume that the abstract model is $\mathcal{M}^m_{abs}$.

Definition 6. Given a parameterised system S_n, m $(1 \le m \le n)$, and formula ϕ indexed by elements of $\{0, \ldots, m-1\}$, the components of S_n are *safe* with respect to $\mathcal{M}^m_{abs}$ if and only if

$$\mathcal{M}^m_{abs} \models \phi \Rightarrow \forall n.\ \mathcal{M}_n \models \phi.$$

In other words, components are *safe* with respect to abstraction if the abstraction of components indexed $\{m, \ldots, n-1\}$ does not alter the behaviour of the system, with respect to ϕ.

Note that the term *safe* here means the same as *abstractable* (i.e. our method is applicable). We prefer *safe* because it captures the notion that, unless strict guidelines are followed, abstractability (safety) will be violated. In the remainder of this paper we omit the condition *with respect to the abstraction*, when it is clear from the context.

In the next section we describe our assumptions on the way basic parameterised systems are specified, and show how abstract models are constructed for such systems in such a way as to preserve given properties. Thus we demonstrate that basic components are safe with respect to our abstraction approach.

In Section 5 we extend the approach to featured systems and show that if features are restricted in some way, components remain safe.

4. Abstraction of basic parameterised systems

We assume that all models are specified in modular GC form (see Section 2.7). Model descriptions consist of either $n-1$ instantiations of a single module declaration, or a single instantiation of a context module declaration together with $n-2$ instantiations of a further module declaration.

Local variables associated with each component are either: *p-variables*, the values of which are drawn from the set of component indices $V = \{D, 0, 1, \ldots, m\}$; *c-variables*, the values of which are channel names; 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 restrict our attention to indexed components whose behaviour does not depend upon a given index value, we refer to this as "open symmetry", see Definition 7. Note that, if ψ is a statement choice, and $\alpha \in Sym(V)$ a permutation (see Definition 5), then $\alpha(\psi)$ is a statement choice obtained from ψ by

(1) replacing all propositions $x == val_1$, where x is a *p_variable* and $val_1 \in V$, contained in the guard of ψ, with $x == \alpha(val_1)$, and
(2) replacing all assignments of the form $y = val_2$, where y is a *p_variable* and $val_2 \in V$ contained in the command of ψ, with $y == \alpha(val_2)$.

Definition 7. Let $\mathcal{M}_n$ be the model associated with a system of n components, expressed in GC form, and let V denote the set of component indices. A set of parameterised components is called open symmetric if for any statement choice ψ contained in a component specification, $\alpha(\psi)$ is also contained in the component specification, for all $\alpha \in Sym(V)$.

As an example, the single statement $(x == 1) \rightarrow y = 42$, would violate open symmetry (unless there were also statements $(x == 0) \rightarrow y = 0$, $(x == 0) \rightarrow y = 1$ etc., for every permutation), but the statement $(x == y) \rightarrow x = partner[z]$ would preserve open symmetry.

Definition 8. Components of a system are said to be *basic* if they are open symmetric and satisfy the following conditions:

(1) The only global variables present in the system are channels or variables that are used for verification purposes only. All channels are finite buffers, and there is one channel associated with each component. Variables that are used for verification purposes only do not appear in guards.
(2) Each component has, amongst its local variables, the standard variable *p_c*, denoting its program counter. In addition, all components (except possibly the context component, when one exists) have the p-variable *selfid* denoting the component index. No operations on this variable are permitted. The value of any other p-variables (general p-variables) can only be changed by reading from a channel, or via non-deterministic choice. No other operations, apart from resetting to D, are permitted. In addition, local variables may include c-variables which denote the channel names associated with the component itself and the component with which the current component is communicating. Operations on these variables are restricted as for *selfid* and general p-variables respectively.
(3) All statement choices within the model specification of basic parameterised systems are assumed to have the form:

$$:: atomic\{((localprop)\&\&(varprop)) \rightarrow command\}$$

where *localprop* and *varprop* are conjunctions of propositions concerning local variables of a component, and global variables (channels) respectively. We assume that in all cases *varprop* contains only propositions concerning the component's own channel and/or any other channel. These propositions may only take the form of a check on the status of a channel (whether it is full, empty etc.), or a poll, which has the form *chan_name*?[*const*] and takes the value *true* if the next message on the channel with name *chan_name* has a value equal to the constant *const*, and *false* otherwise.

4.1. Constructing the abstract model

In this section we show how concrete components are modified and how the abstract process *Abs* is constructed. In Section 4.2 we show that due to the nature of our construction of the abstract system, basic components, as defined above, are safe.

In all cases we have a fixed number of concrete components (m say). The total number of components is $N = m + 1$. The abstract component is assumed to have an index *Absid* which is set to m, and associated channel *abs_channel*. The default value D is set to $Absid + 1$. There is one channel for each concrete component.

We start with peer to peer networks. In this case, all concrete components have the same form and are modified in the same way.

Recall, any statement choice in the component specification has the form

$$:: \ atomic\{((localprop)\&\&(varprop)) \rightarrow command\}.$$

If a statement choice contains no propositions concerning global variables (i.e. *varprop* is empty) and the corresponding *command* involves updating local standard variables or resetting *p*_variables to D only, the statement choice is unchanged in the modified component.

Suppose that *varprop* is empty and *command* involves updating (not resetting) local p-variables to a value from $V \setminus D$ or updating a c-variable. We assume that a given command updates all p-variables to the same value, and any c-variables to the same value. If a command contains updates to both p-variables and c-variables then the c-variables are updated to the channel name associated with the value to which the p-variables are updated. Since our components are assumed to be open symmetric, the original component specification will contain n equivalent statements, one for each component index. For example, suppose the component specification contains the statement choice:

$$:: \ atomic\{(p\_c == 4) \rightarrow partnerid = 0; partner = zero\}$$

where *zero* is the channel name associated with the component with index 0. Then the component specification will also contain the statement choices:

$$:: \ atomic\{(p\_c == 4) \rightarrow partnerid = 1; partner = one\}$$
$$:: \ atomic\{(p\_c == 4) \rightarrow partnerid = 2; partner = two\}$$

etc. (Note that *p_c* is a standard variable, and so is not permuted.) This list of statements should be replaced with a list of m statement choices, corresponding to selecting *partnerid* as 0, 1, up to $m - 1$ together with a final statement:

$$atomic\{(p\_c == 4) \rightarrow partnerid = Absid; partner = abs\_channel\}.$$

Statement choices in which *varprop* is non-empty contains a proposition concerning the status of a channel (whether it contains a given message, for example). We consider the case where *varprop* is non-empty and *command* does not involve reading or writing from/to channels. Suppose a statement choice has associated *varprop* which only contains propositions concerning *self*. The statement choice should be left unchanged if the value of *partnerid* (or *partner*) is currently set to the default value (i.e. a communication has yet to be established). However, if communication has been established, the statement choice should be replaced by a set of statement choices. The first choice is simply the original choice in which the guard is enhanced with the proposition ($partner! = abs\_channel$) (or ($partnerid! = Absid$)). In the other choices the proposition ($partner == abs\_channel$) (or ($partnerid == Absid$)) is added to the guard and the proposition querying the status of *self* is removed. Each choice will have a different command depending on the assumed status of the channel. For example, consider the statement choice:

$$\begin{aligned}&:: \ atomic\{((p\_c == 2)\&\&(self?[eval(partner)])) \rightarrow \\ &\quad MYSTATE = talk; p\_c = 4\}\end{aligned}$$

which would block if channel *self* did not contain the current value of *partner* (in Promela, $eval(partner)$ is a constant assigned to the channel name currently assigned to the c-variable *partner*). This would be replaced by the statement choices:

$$\begin{aligned}&:: \ atomic\{((p\_c == 2)\&\&(partner! = abs\_channel) \\ &\quad \&\&(self?[eval(partner)])) \rightarrow MYSTATE = talk; p\_c = 4\} \\ &:: \ atomic\{((p\_c == 2)\&\&(partner == abs\_channel)) \rightarrow \\ &\quad MYSTATE = talk; p\_c = 4\} \\ &:: \ atomic\{((p\_c == 2)\&\&(partner == abs\_channel)) \rightarrow p\_c = 2\}.\end{aligned}$$

All statement choices in which the *varprop* contains a query of *partner* and *command* do not involve a read from, or write to, a channel, are treated in the same way.

If *command* involves a read from or write to *partner* after communication has been established then, since we have assumed that (atomic) statement choices will not block, *varprop* will be non-empty. Assume that *varprop* only contains the associated proposition ($nempty(partner)$), or ($nfull(partner)$). The statement choice should be replaced by three choices. In the first

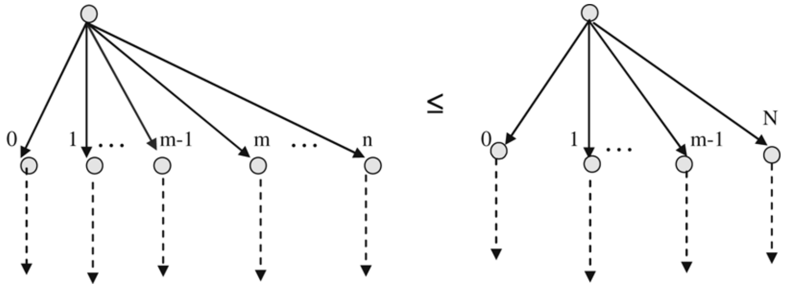

Fig. 4. Data abstraction.

choice, as before, the guard is enhanced with the proposition (*partnerid*! = *Absid*) (or equivalently, (*partner*! = *abs_channel*)) and the command unchanged. In the other choices, we add the proposition (*partnerid* == *Absid*) (or (*partner* == *abs_channel*)) to the guard and remove the associated (*nempty*(*partner*)), or (*nfull*(*partner*)) proposition from the guard. In the second statement choice we assume that read (from *partner*) or write (to *partner*) is enabled, and the command simply has the read or write command removed (we refer to this as a *virtual* read or write.) In the third choice we assume that the read or write statement is not enabled, and so the command is replaced with a simple command to keep *p_c* at its current value.

We have described how simple statement choices are replaced in the modified concrete components. Clearly statement choices can be more complicated (the guard may contain a non-empty *varprop* and *command* an assignment of a value to a *p*-variable, for example). However, by iteratively applying simple modifications, more complex statement choices can be modified in a natural way.

In client–server networks, concrete client components require little modification, since they communicate only via the server component. However, when selecting a destination for messages, for example, they must now choose from the set of concrete client components together with the abstract component.

The server component however, requires more modification. Communication with concrete clients is unchanged, but communication with the abstract component is modified as for concrete components in peer to peer networks (see Fig. 3).

Note that (in either network topology) an alternative solution to deciding whether or not a write (say) to an abstract partner is blocked (other than choosing non-deterministically), is to include a global variable *blocked* say, which is non-deterministically set to 0 or 1 by the abstract process. In our abstract email example (see Section 4.3) we use this alternative approach.

Finally we show how the abstract process, *Abs* is constructed.

The role of the abstract process is to initiate messages. Therefore, *Abs* places messages on channels of any concrete component with which it can directly communicate. In a peer to peer network this includes all concrete components, and in a client–server network this only includes the server component (via the *network* channel). In our telephone example (see Section 4.3) there is no more behaviour associated with *Abs*. In the email example, the *Abs* component also has the ability to set the *blocked* variable (see above).

4.2. Proving that basic components are safe

We show that basic components, as defined in Definition 8 are safe.

Theorem 9. *Given a parameterised system*

$$S_n = p_0||p_1||\cdots||p_{n-1} \text{ or } C||p_0||p_1||\cdots||p_{n-2}$$

with model $\mathcal{M}_n$, m $(1 \le m \le n)$, *abstract model* $\mathcal{M}_{abs}^m$ *constructed as described above, and formula* ϕ *indexed by elements of* $\{0, \ldots, m-1\}$, *if the components of* S_n *are basic, then they are safe.*

Proof (*sketch*). Components are safe if $\mathcal{M}_n \preceq \mathcal{M}_{abs}^m$. The simulation follows from construction of $\mathcal{M}_{abs}^m$ as described above: each statement in the unabstracted specification can be matched (or replaced) by a statement in the abstracted specification. At the model level, matching is best illustrated by Figs. 4–6. Fig. 4 illustrates data abstraction [16], where a choice over n possibilities is matched by $m+1$ possibilities, and when appropriate, N represents the values $\{m, \ldots, n-1\}$. Fig. 5 illustrates behavioural abstraction: a choice over (sub) paths of arbitrary length is matched by a loop. Note that when this kind of loop is required, for example in the email system to represent the possibility of blocking, then some liveness properties may not hold in the abstracted model (see Section 6). Fig. 6 illustrates another form of behavioural abstraction: stuttering. States s_1 and s_2 are distinct states, but they are both matched by t_1 because the transition from s_1 to s_2 results from an update to a variable that does not change in the abstract model. For example, this could correspond to the case of empty commands in the abstracted model, representing, say, *virtual* read or writes to communication channels. □

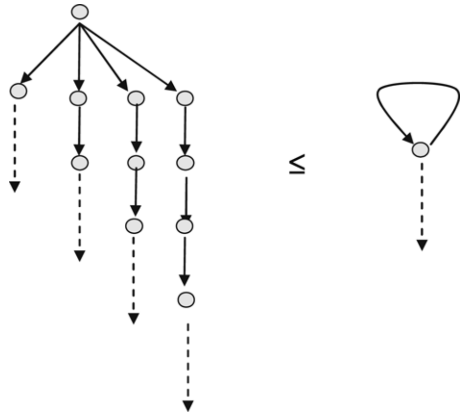

Fig. 5. Behavioural abstraction.

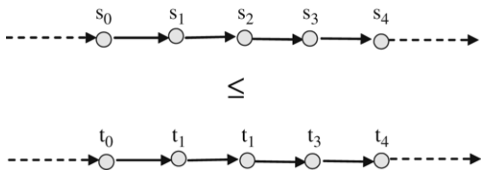

Fig. 6. Stuttering.

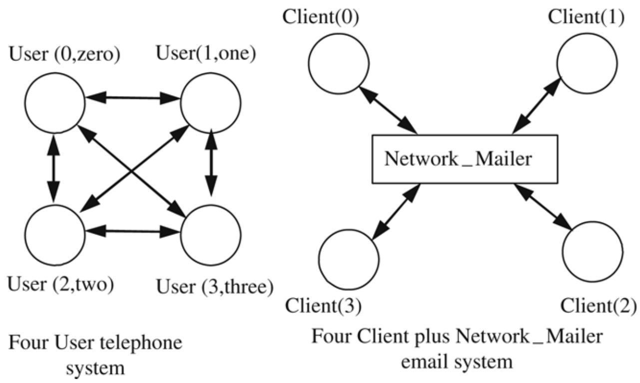

Fig. 7. Example telephone and email systems.

We now present some examples in detail.

4.3. Some examples

We illustrate our approach via a simple telephone system and a simple email system. We first describe the systems informally together with a property that we wish to verify in each case. We then provide Promela descriptions of the systems in modular guarded command form and show how these descriptions are modified to create an abstract specification in each case.

4.3.1. An informal description of simple telephone and email systems

The telephone system consists of four instantiations of a *User* process, there is no context process. Processes are parameterised via process identifier (*selfid*) and designated channel name (*self*). The email system consists of four instantiations of a *Client* process which communicate via a server process (the *Network_Mailer* process). The topologies are illustrated in Fig. 7. Note that arrows indicate the direction of communication, not ownership of channels.

In the simple telephone system, User components change state (between *idle*, *calling* and *talk*) as a result of communication with other processes (see Fig. 8). Suppose a User is in the *idle* state. It will first check to see if their own channel is empty, and, if so, choose a partner whose channel is also empty. It then places its own channel name *self* on both its own channel and that of their partner, and proceeds to the *calling* state to wait for a "reply". The User detects a reply when the contents of its channel have been replaced with the channel name of *partner*, and proceeds to the *talk* state. Once in the *talk* state, as the initiator of the call, the User can end the call (hang up) by replacing the message on its partner's channel

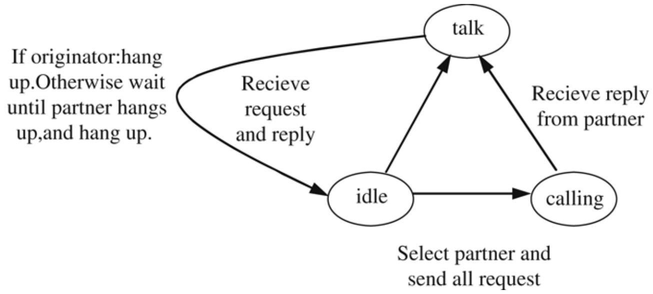

Fig. 8. State transition diagram for simple telephone system (user component).

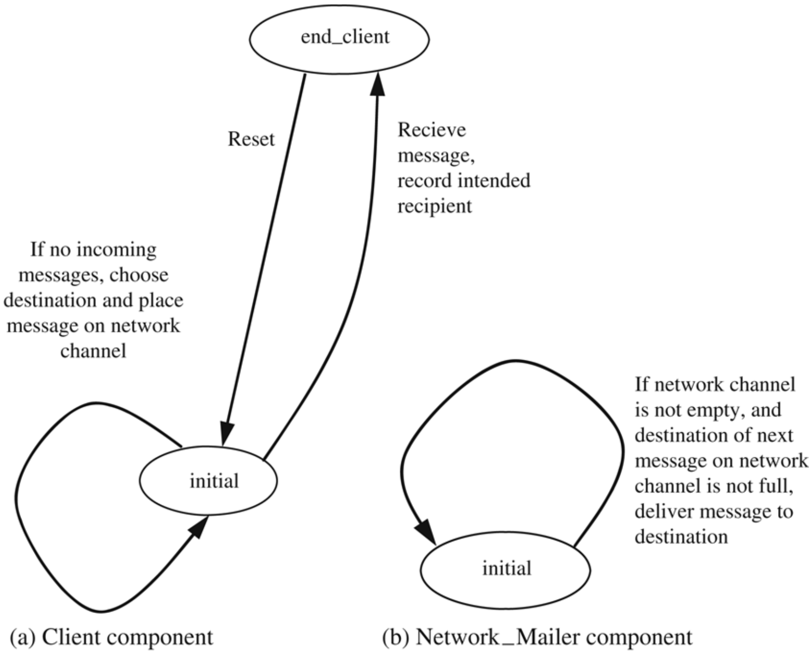

Fig. 9. State transition diagram for simple email system.

with the partner's channel name, and removing the contents of *self*. Alternatively, a User in the idle state that has a full channel will replace the contents of its partner's channel with *self* and proceed straight to the *talk* state. The User then waits for its partner to hang up, removes the contents of *self* and returns to *idle*. An example property for the simple telephone system is:

Property 1. *If User*[0] *has User*[1] *as its partner, and User*[1] *has User*[0] *as its partner, then User*[0] *and User*[1] *will be connected before one of them returns to the idle state.*

In the email example, the *Client* components move between two states, namely *initial* and *end_Client* (see Fig. 9(a)). If a *Client* component in the *initial* state receives a message, it reads the message, records the identity of the intended recipient, and moves to the *end_client* state. In *end_client* the value of this record is reset and the *Client* returns to the *idle* state. From the *initial* state the *Network_Mailer* process continuously loops around a single state to check if there are any messages on its associated channel (*network*), and if so, whether the channel associated with the next message on the channel is not full. If so, the message is passed on accordingly (see Fig. 9(b)). An example property for the simple email system is:

Property 2. *All messages received by Client*[0] *are addressed to Client*[0].

4.4. Promela specifications for example systems

Promela specifications for simple telephone and email systems (expressed in modular guarded form) are given below. Note that this is not the most natural way to express Promela programs — it prevents us from using *goto* statements and *labels* for example (thus in practice we transform a given Promela specification into this form). Assuming Kripke structures $\mathcal{M}_4$ associated with these specifications, we show how these simple programs are be adapted to construct abstract specifications, with associated models $\mathcal{M}^2_{abs}$ in each case.

```
/*simplified telephone example*/
mtype={idle,calling,talk};
chan zero = [1] of {chan};
chan one = [1] of {chan};
chan two = [1] of {chan};
chan three = [1] of {chan};
chan null = [1] of {chan};
#define N 4 /*number of processes*/
typedef array { byte to[N] };
array connected[N];
#define default 5
byte MYSTATE[N]=idle;
chan partner[N];
short p0=-1;
short p1=-1;

proctype User(mtype selfid;chan self)
{chan messchan=null;
 bit originating=0;
 byte partnerid=default;
 byte p_c=0;
 do
 ::/*idle*/atomic{((p_c==0)&&empty(self)&&empty(zero)&&(self!=zero))->
    self!self;zero!self;partner[selfid]=zero; partnerid=0;originating=1;
    MYSTATE[selfid]=calling; p_c=1}
 ::/*idle*/atomic{((p_c==0)&&empty(self)&&empty(one)&&(self!=one))->
    self!self; one!self; partner[selfid]=one; partnerid=1;originating=1;
    MYSTATE[selfid]=calling; p_c=1}
::/*idle*/atomic{((p_c==0)&&empty(self)&&empty(two)&&(self!=two))->
    self!self; two!self; partner[selfid]=two; partnerid=2;originating=1;
    MYSTATE[selfid]=calling; p_c=1}
::/*idle*/atomic{((p_c==0)&&empty(self)&&empty(three)&&(self!=three))->
    self!self; three!self; partner[selfid]=three; partnerid=3;originating=1;
    MYSTATE[selfid]=calling; p_c=1}
::/*idle*/atomic{((p_c==0)&&(self?[messchan]))->
    self?<messchan>; partner[selfid]=messchan;
    partner[selfid]?messchan;partner[selfid]!self; messchan=null;
    MYSTATE[selfid]=talk;p_c=2}
::/*calling*/atomic{((p_c==1)&&(self?[eval(partner[selfid])]))->
    connected[selfid].to[partnerid]=1; connected[partnerid].to[selfid]=1;
    MYSTATE[selfid]=talk;p_c=2}
::/*talk*/atomic{((p_c==2)&&(originating==1))->
    partner[selfid]?messchan; partner[selfid]!partner[selfid]; self?messchan;messchan=null;
    connected[selfid].to[partnerid]=0; connected[partnerid].to[selfid]=0;
    partner[selfid]=null;partnerid=default;originating=0; MYSTATE[selfid]=idle;p_c=0}
::/*talk*/atomic{((p_c==2)&&(originating==0)&&(self?[eval(self)]))->
   self?messchan;messchan=null;partner[selfid]=null; MYSTATE[selfid]=idle;p_c=0}
od}

init{atomic{p0=run User(0,zero);
            p1=run User(1,one);
            run User(2,two);
            run User(3,three)}}
```

Fig. 10. Promela specification for telephone example with four *User* components.

The Promela specification for the simple four *User* telephone system is given in Fig. 10; Property 1 is given by:

$$[]((s \wedge t) \rightarrow !((!r) \cup (v \vee w))).$$

Here r is $(connected[0].to[1] == 1)$, s is $(partner[0] == one)$, t is $(partner[1] == zero)$, v is $(MYSTATE[0] == idle)$ and w is $(MYSTATE[1] == idle)$.

Here *connected.to* is an array, the elements of which are variables used for reasoning purposes only (and so do not appear in guards). When a connection has been established between i and j, $connected[i].[j]$ is set to 1 and is (re)set to 0 otherwise. The *partner* variables are global here. This is to allow their values to be “visible” to the never-claim. In all other ways they are

```
/*simplified email example*/
#define no_clients 4
#define M 4
typedef Mail {byte sender; byte receiver};
chan null = [1] of {Mail};
chan zero = [1] of {Mail};
chan one = [1] of {Mail};
chan two = [1] of {Mail};
chan three = [1] of {Mail};
chan network = [1] of {Mail};
byte last_del_to_to[no_clients]=M;

proctype Client(byte id;chan mybox)
{Mail msg;
 byte p_c=0;
 atomic{msg.sender=M;msg.receiver=M};
do
::/*initial*/atomic{((p_c==0)&&(nempty(mybox)))->
   mybox?msg; last_del_to_to[id]=msg.receiver; msg.sender=M;
   msg.receiver = M; p_c=1}
::/*initial*/atomic{((p_c==0)&&(empty(mybox)) && (nfull(network)))->
   msg.receiver=0; msg.sender=id; network!msg; msg.sender=M;
   msg.receiver = M; p_c=0}
::/*initial*/atomic{((p_c==0)&&(empty(mybox)) && (nfull(network)))->
   msg.receiver=1; msg.sender=id; network!msg; msg.sender = M;
   msg.receiver=M; p_c=0}
::/*initial*/atomic{((p_c==0)&&(empty(mybox))&&(nfull(network)))->
   msg.receiver=2; msg.sender=id; network!msg; msg.sender=M;
   msg.receiver=M; p_c=0}
::/*initial*/atomic{((p_c==0)&&(empty(mybox)) && (nfull(network)))->
   msg.receiver=3;msg.sender=id; network!msg; msg.sender=M;
   msg.receiver=M; p_c=0}
::/*end_client*/atomic{(p_c==1)->last_del_to_to[id]=M; p_c=0}
od;}

proctype Network_Mailer()
{Mail msg; byte p_c=0;
 atomic{ msg.sender=M; msg.receiver= M;}
do
::/*initial*/atomic{((p_c==0)&&(network?[msg.sender,0])&&(nfull(zero)))->
   network?msg; zero!msg;msg.sender=M; msg.receiver=M; p_c=0}
::/*initial*/atomic{((p_c==0)&&(network?[msg.sender,1])&&(nfull(one)))->
   network?msg; one!msg; msg.sender=M; msg.receiver=M; p_c=0}
::/*initial*/atomic{((p_c==0)&&(network?[msg.sender,2])&&(nfull(two)))->
   network?msg; two!msg; msg.sender=M; msg.receiver=M; p_c=0}
::/*initial*/atomic{((p_c==0)&&(network?[msg.sender,3])&&(nfull(three)))->
   network?msg; three!msg; msg.sender = M; msg.receiver=M; p_c=0}
od}

init{atomic{ run Network_Mailer(); run Client(0,zero);
      run Client(1,one); run Client(2,two); run Client(3,three)}}
```

Fig. 11. Promela specification for email example with four *Client* components and a *Network_Mailer* component.

treated the same as the local channel names *partner* described in Section 4. The global variable array *MYSTATE* is also used for verification purposes only.

The Promela specification for the simple email system consisting of three *Client* components and the *Network_Mailer* component is given in Fig. 11; Property 2 is given by:

$$[](p \vee q)$$

where p is ($last\_del\_to\_to[1] == 1$), and q is ($last\_del\_to\_to[1] == M$). Here, *last_del_to_to* is for verification purposes only and records the identity of the intended recipient; M is a default value.

4.4.1. The example abstract models

Abstract Promela specifications are given in full in Figs. 12 and 13. Note that in the email example, no abstract channel is required because channel names are not passed between components, and all messages delivered *to* the abstract process are *virtual* (see Section 4.1).

```
/*abstract simplified telephone example*/
mtype={idle,calling,talk};
chan zero = [1] of {chan}; chan one = [1] of {chan};
chan abs_channel =[1] of {chan}; chan null = [1] of {chan};
#define N 3 /*number of processes*//*two concrete, one abstract*/
#define Absid 2
typedef array { byte to[N] }; array connected[N];
#define default 3
byte MYSTATE[N]=idle; chan partner[N]; short p0=-1; short p1=-1;

proctype User (mtype selfid;chan self)
{chan messchan=null; bit originating=0; byte partnerid=default; byte p_c=0;
 do
 ::/*idle*//*abstract choices*/  /*if abstract partner is not busy*/
     atomic{((p_c==0)&&empty(self))->self!self;  partner[selfid]=abs_channel;
     partnerid=Absid;originating=1;  MYSTATE[selfid]=calling; p_c=1}
 ::/*idle*//*abstract choices*//*abstract partner busy, no other component free*/
     atomic{((p_c==0)&&(empty(self))&&(self==zero)&&(full(one)))-> p_c=0}
   /*abandon call*/
 ::/*idle*//*abstract choices*//*abstract partner busy, no other component free*/
   atomic{((p_c==0)&&(empty(self))&&(self==one)&&(full(zero)))->p_c=0}
   /*abandon call*/
 ::/*idle*/atomic{((p_c==0)&&(empty(self))&&(empty(zero))&&(self!=zero))->
   self!self;zero!self;partner[selfid]=zero; partnerid=0; originating=1;
   MYSTATE[selfid]=calling;p_c=1}
 ::/*idle*/atomic{((p_c==0)&&empty(self)&&empty(one)&&(self!=one))->
   self!self;one!self;partner[selfid]=one; partnerid=1; originating=1;
   MYSTATE[selfid]=calling;p_c=1}
 ::/*idle*//*abstract choice*/atomic{((p_c==0)&&(self?[eval(abs_channel)]))->
   self?<messchan>;assert(messchan!=self); partner[selfid]=messchan;
   messchan=null; MYSTATE[selfid]=talk;p_c=2}
 ::/*idle*/atomic{((p_c==0)&&(!(self?[eval(abs_channel)]))&&(self?[messchan]))->
   self?<messchan>;assert(messchan!=self); partner[selfid]=messchan;
   partner[selfid]?messchan; partner[selfid]!self; messchan=null;
   MYSTATE[selfid]=talk;p_c=2}
 ::/*calling*//*abstract choice*//*if partner has responded*/
   atomic{((p_c==1)&&(partner[selfid]==abs_channel))->
   connected[selfid].to[partnerid]=1; connected[partnerid].to[selfid]=1;
   MYSTATE[selfid]=talk; p_c=2}
 ::/*calling*//*abstract choice*/ /*if partner has not responded*/
   atomic{((p_c==1)&&(partner[selfid]==abs_channel))->p_c=1}
 ::/*calling*/atomic{((p_c==1)&&(self?[eval(partner[selfid])]))->
    connected[selfid].to[partnerid]=1; connected[partnerid].to[selfid]=1;
    MYSTATE[selfid]=talk;p_c=2}
 ::/*talk*//*abstract choice*/
   atomic{((p_c==2)&&(originating==1)&&(partner[selfid]==abs_channel))->
   self?messchan;messchan=null; connected[selfid].to[partnerid]=0;
   connected[partnerid].to[selfid]=0; partner[selfid]=null;
   partnerid=default;originating=0; MYSTATE[selfid]=idle; p_c=0}
::/*talk*/atomic{((p_c==2)&&(originating==1)&&(partner[selfid]!=abs_channel))->
   partner[selfid]?messchan;partner[selfid]!partner[selfid]; self?messchan;
   messchan=null; connected[selfid].to[partnerid]=0;
   connected[partnerid].to[selfid]=0; partner[selfid]=null;
   partnerid=default;originating=0; MYSTATE[selfid]=idle;p_c=0}
::/*talk*//*abstract
choice*/atomic{((p_c==2)&&(originating==0)&&(partner[selfid]==abs_channel))->
  /*if other party has hung up*/
  self?messchan;messchan=null;partner[selfid]=null; MYSTATE[selfid]=idle; p_c=0}
::/*talk*//*abstract
choice*/atomic{((p_c==2)&&(originating==0)&&(partner[selfid]==abs_channel))->
  /*if other party has not hung up*/p_c=2}
::/*talk*/atomic{((p_c==2)&&(originating==0)&&(partner[selfid]!=abs_channel)
  &&(self?[eval(self)]))-> self?messchan;messchan=null;partner[selfid]=null;
  MYSTATE[selfid]=idle;p_c=0}od}

proctype Abs (chan self){
do
:: zero!self
:: one!self
od}

init{atomic{p0=run User(0,zero); p1=run User(1,one); run Abs(abs_channel)}}
```

Fig. 12. Promela specification for telephone example with abstraction.

5. Adding features

Features are a mechanism for structuring functionality additional to a basic behaviour (see Section 2.2).

```
/*Abstract simplified email model*/
typedef Mail {byte sender; byte receiver};
#define no_clients 3 /*2 concrete, 1 abstract*//*Network_Mailer does not have its own id*/
#define Absid 2
#define M 3 /* default value of variables*/
bit blocked=0;
chan null = [1] of {Mail};
chan zero = [1] of {Mail};
chan one = [1] of {Mail};
chan network = [1] of {Mail};
byte last_del_to_to[no_clients]=M;

proctype Client(byte id;chan mybox)
{Mail msg; byte p_c=0; atomic{msg.sender=M;msg.receiver=M};
do
::/*initial*/atomic{((p_c==0)&&(nempty(mybox)))->
   mybox?msg; last_del_to_to[id]=msg.receiver; msg.sender = M; msg.receiver = M; p_c=1}
::/*initial*/atomic{((p_c==0)&&(empty(mybox)) && (nfull(network)))->
   msg.receiver=0; msg.sender = id; network!msg;  msg.sender = M; msg.receiver = M; p_c=0}
::/*initial*/atomic{((p_c==0)&&(empty(mybox)) && (nfull(network)))->
    msg.receiver=1;msg.sender = id;network!msg;  msg.sender = M; msg.receiver = M; p_c=0}
::/*initial*/atomic{((p_c==0)&&(empty(mybox)) && (nfull(network)))->
   msg.receiver=Absid; msg.sender = id; network!msg;  msg.sender = M; msg.receiver = M; p_c=0}
::/*endClient*/atomic{(p_c==1)->last_del_to_to[id]=M; p_c=0}
od;} /*end Client process*/

proctype Network_Mailer()
{Mail msg; byte p_c=0; atomic{ msg.sender = M;  msg.receiver= M;}
do
::/*initial*/atomic{((p_c==0)&&(network?[msg.sender,0])&&(nfull(zero)))->
   network?msg;  zero!msg;msg.sender = M; msg.receiver=M; p_c=0}
::/*initial*/atomic{((p_c==0)&&(network?[msg.sender,1])&&(nfull(one)))->
   network?msg;  one!msg; msg.sender = M; msg.receiver=M; p_c=0}
::/*initial*//*Abstract choice*/ atomic{((p_c==0)&&(network?[msg.sender,2])&&(blocked==0))->
   network?msg; /*send on abstract message*/  p_c=0}
od}

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

init{atomic{ run Network_Mailer(); run Client(0,zero); run Client(1,one);
             run Abs(2)}}
```

Fig. 13. Promela specification for email example with abstraction.

We have added features to a basic telephone system and email system [8,10]. Note that these specifications are far more complex than those given in Section 4.3, which were provided merely to illustrate the basic approach. We therefore give

Table 1
Features (telephone system)

Feature	Description
Call forwarding unconditionally (CFU)	If $CFU[i] == j$, all calls for $User[i]$ are forwarded to $User[j]$
Call forwarding on busy (CFB)	If $CFB[i] == j$, if $User[i]$ is busy then all calls for $User[i]$ are forwarded to $User[j]$
Outgoing dial screening (ODS)	If $ODS[i] == j$, then $User[i]$ may not dial $User[j]$'s number
Outgoing call screening (OCS)	If $OCS[i] == j$, then a call from $User[i]$ to $User[j]$ is not possible
Terminating call screening (TCS)	If $TCS[i] == j$, then a call from $User[j]$ to $User[i]$ is not possible
Ring back when free (RBWF)	If $RBWF[i]! = D$, and $User[i]$ requests a ringback to $User[j]$ then a ringback (from $User[i]$) will ensue when $User[j]$ becomes free
Outgoing calls only (OCO)	If $OCO[i]! = D$ then $User[i]$ may not receive any calls
Terminating calls only (TCO)	If $TCO[i]! = D$ then $User[i]$ may not initiate any calls
Return when free (RWF)	If $RBW[i]! = D$, and $User[j]$ requests a ringback to $User[i]$ then a ringback (from $User[i]$) will ensue when $User[i]$ becomes free

Table 2
Features (email system)

Feature	Description
Encryption (ENC)	If $ENC[i]! = D$ then all messages sent by $Client[i]$ will be encrypted
Decryption (DEC)	If $DEC[i]! = D$ then $Client[i]$ can decrypt all messages delivered to $Client[i]$
Filtering (FT)	If $FT[i] == j$ then all messages sent to $Client[i]$ by $Client[j]$ will not be delivered
Forwarding (FW)	If $FW[i] == j$ then all messages sent to $Client[i]$ will be forwarded to $Client[j]$
Autorespond (AR)	If $AR[i]! = D$ then the first time $Client[j]$ sends a message to $Client[i]$, an autoresponse message will be sent to $Client[j]$
Mailhost (MH)	If $Client[i]$ is not on the mailhost list, it can not receive messages
Remail (RM)	If $RM[i]! = D$ then all messages sent by $Client[i]$ will be delivered under $Client[i]$'s pseudonym, and all messages addressed to $Client[i]$'s pseudonym will be delivered to $Client[i]$

only an overview here of relevant aspects and assumptions. Lists of features for each of these systems are given in Tables 1 and 2; D is a default value and we assume $i \neq j$.

We add features to components via *feature arrays* which determine which features are subscribed to by which components. Thus additional global variables are now allowed to appear in guards. This is the major difference between basic components and featured components.

Suppose then that 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 assume that all global variables $glob\_var$ are feature related (either concerning the elements of a feature array, or a *feature-flag array*, see Section 5.1) or are used for verification purposes only (and so do not appear in guards, as before).

Now we assume that all statement choices have the form:

$$:: \ atomic\{((feature\_prop)\&\&(localprop)\&\&(varprop)) \rightarrow command\}$$

where *feature_prop* is either empty, or refers to feature-related global variables. If *feature_prop* is not empty, we refer to the statement choice as a *feature statement choice*, otherwise it is a *basic statement choice*.

No component with index i can carry out any operation on a global variable $glob\_var[j]$, for any $j \in V$, $j \neq i$, unless $glob\_var$ is a *feature-flag array* and the operation occurs within a feature statement choice.

Definition 10. Components are said to be safely featured if they satisfy the assumptions detailed above.

5.1. Categorising features

Recall feature statement choices have the form

$$:: \ atomic\{((feature\_prop)\&\&(localprop)\&\&(varprop)) \rightarrow command\}.$$

Depending on the form of *feature_prop* it is possible to develop a feature categorisation. We will subsequently use our categorisation to determine which features can be considered *safe* with respect to our abstraction technique.

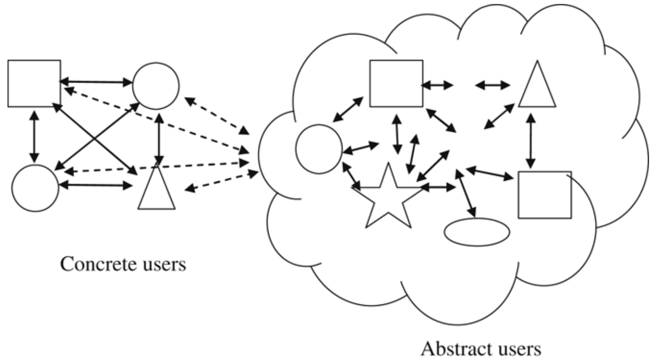

Fig. 14. Featured peer to peer abstraction.

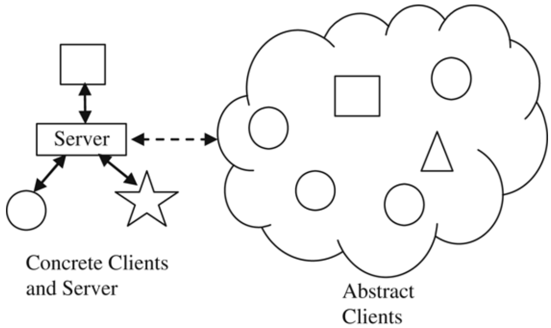

Fig. 15. Featured client server abstraction.

Let us first consider *feature_prop*. This has one of the following forms:

$$feature\_name[myvar_1] \;==\; myvar_2 \quad \text{or}$$
$$feature\_name[myvar_1] \;\,!=\; D$$

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*}.

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 statement choices, within all *feature_prop* guards, *myvar*1 is *selfid*, *partnerid*, or some other p-variable. Examples of the first category are ODS (telephone) and ENC (email). An example of a *partner owned* feature is CFU. 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 FT and FW which are owned by a *Client* process, but managed by the *Network_Mailer* process.

Note that the only one of our example features that cannot be described in these terms is RWF. 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*.

Definition 11. A feature is said to be multi-owned if it is not host, partner or third party owned.

5.2. Constructing the abstract model for featured systems

In this section we provide a sketch of our abstraction approach in the presence of features. We extend the modifications of statement choices in concrete components (see Section 4.1) to feature statement choices. We then show for which features our abstraction approach is still *safe*. Figs. 14 and 15 illustrate the approach; different shapes indicate that components may not be isomorphic (because of the presence of features).

In earlier work [34] we have shown how feature statement choices should be treated for host owned, partner owned and third party features. We do not provide full details here, but give an overview.

In all cases, statement choice must be split (in the same way as the treatment of basic statement choices described in Section 4.1) according to whether the current partner is abstract or not. In the former case, if the feature is partner owned, two possibilities must be considered: whether the partner has the feature or not. If so, different possible results of applying the feature must be considered.

For example, the following statement choice is for *CFU*:

$$\begin{aligned}
:: atomic\{(\quad & (state == st_diall)\&\&(CFU[partnerid]! = default1) \\
& \&\&(position_prop)) \rightarrow \\
& partnerid = CFU[partnerid]; \\
& partner[selfid] = chan_name[partnerid]\}.
\end{aligned}$$

Note that *state* is a local variable, and *position_prop* a local variable containing a disjunction of propositions regarding the current value of *p_c* (associated with points in the specification at which features are implemented). When *position_prop* is true but all guards of feature statement choices are false, *p_c* is incremented (via another statement choice, not given here).

Assuming two concrete components, this choice is replaced in the modified component specification with the following choices:

$$\begin{aligned}
:: atomic\{(\quad & (state == st_diall)\&\&(partnerid! = Absid) \\
& \&\&(CFU[partnerid]! = default1)\&\&(position_prop)) \rightarrow \\
& partnerid = CFU[partnerid]; \\
& partner[selfid] = chan_name[partnerid]\} \\
:: atomic\{(\quad & (state == st_diall)\&\&(partnerid == Absid) \\
& \&\&(forwarding_feature == on)\&\&(position_prop)) \rightarrow \\
& partnerid = 0; partner[selfid] = zero; \\
& forwarding_feature = off\} \\
:: atomic\{(\quad & (state == st_diall)\&\&(partnerid == Absid) \\
& \&\&(forwarding_feature == on)\&\&(position_prop)) \rightarrow \\
& partnerid = 1; partner[selfid] = one; \\
& forwarding_feature = off\} \\
:: atomic\{(\quad & (state == st_diall)\&\&(partnerid == Absid) \\
& \&\&(forwarding_feature == on)\&\&(position_prop)) \rightarrow \\
& forwarding_feature = off\}.
\end{aligned}$$

Here *forwarding_feature* is a local variable that is non-deterministically set to *on* or *off* in the preceding statement (when the partner is abstract). The first choice corresponds to the case when the current partner is not abstract and subscribes to CFU. The remaining choices correspond to the case when the current partner is abstract and forwards to a concrete component or to another abstract component. The *forwarding_feature* variable is reset after the feature has been applied. One reason for this is that a chain of forwarding within abstract components is observably equivalent to a single forward, so the feature need not be repeatedly applied.

5.3. Proving that featured components are safe

Our main result is a theorem which shows that the abstraction approach is sound for safely featured components (see Definition 10) that are not multi-owned (see Definition 11).

Theorem 12. *Given a featured, parameterised system*

$$S_n = p_0||p_1||\cdots||p_{n-1} \quad \text{or } C||p_0||p_1||\cdots||p_{n-2}$$

with model $\mathcal{M}_n$, m ($1 \le m \le n$), abstract model $\mathcal{M}^m_{abs}$ constructed as described above, and formula ϕ indexed by elements of $\{0, \ldots, m-1\}$, if the components of S_n are safely featured and none of the features are multi-owned, then the components are safe.

Proof. The proof is similar to that for basic components (see Section 4.2). For all feature statement choices that are not multi-owned, transitions arising from executing the associated statement can be matched by transitions arising from modified statement choices in the abstract model. However, this is not true for feature statement choices pertaining to features that are multi-owned.

Multi-owned features sometimes trigger a change in behaviour because the host component has the feature, and sometimes because the partner component has the feature. The feature is implemented when either the host or partner component has set a *feature_flag*. As the *feature_flag* could have been reset by an abstract component, we cannot simulate the possibility of an abstract component resetting this variable *at any time*. We cannot simply use non-deterministic choice to decide whether the *feature_flag* has been set (presumably to *Absid*) because to do so would assume that at some point an existing, non-default value of the flag may have been overridden. This would imply an earlier transition which would not have been reflected in our simulated model. □

6. Interpreting results

From Theorem 12 we can see that if a formula ϕ indexed by elements of $\{0, 1, \ldots, m-1\}$ holds for abstract model $\mathcal{M}^m_{abs}$ (with concrete components $p_0, p_1, \ldots, p_{m-1}$), then ϕ holds for any model $\mathcal{M}_n$, $(1 \le m \le n)$, consisting of components $p_0, p_1, \ldots, p_{m-1}$ and $n-m$ other components, subscribing only to safe features.

However, what can we conclude if, for some ϕ, $\mathcal{M}^m_{abs} \not\models \phi$?

If we can show that for the small finite model $\mathcal{M}_m = \mathcal{M}(p_0||p_1||\cdots||p_{m-1})$, $\mathcal{M}_m \not\models \phi$, then the counterexample generated for $\mathcal{M}_m$ will extend to $\mathcal{M}_n$, for all $1 \le m \le n$. So we can conclude that $\mathcal{M}_n \not\models \phi$.

However, it is possible that $\mathcal{M}_m \models \phi$ but $\mathcal{M}^m_{abs} \not\models \phi$ (possibly due to additional non-determinism introduced via the abstraction process. This is likely to be the case if ϕ is a liveness property). In some instances it might be possible to improve our abstraction via a method of refinement [13,30,5]. This would involve making the abstract model more concrete, thereby allowing ϕ to become true. This is the subject of future work.

7. Applying the approach

In this section we consider how models are constructed automatically, and we also give some experimental results.

7.1. Constructing an abstract model

Given a Promela specification of a parameterised component (and a context component, as required), and a fixed m, the abstract specification is constructed as follows. First, transform the parameterised component(s) into modular GC form. Second, modify the component(s) to become the (parameterised) concrete component (or modify the context component), and construct the component *Abs*, as described in Sections 4.1 and 5.2. Third, define a process which runs m instantiations of the concrete component, along with *Abs*. Finally, model check the resulting specification.

Each of these steps can be automated, for example, we have implemented them via Perl scripts.

Note that, if Promela components are expressed in GC form it is possible to perform a syntactic check to ensure that they are indeed safe with respect to the abstraction approach. For example, a tool similar to SymmExtractor [19] can be used to check that local variables *selfid* and *partnerid* are used appropriately and that components are open symmetric. We have not used such a tool here (the component descriptions were constructed in such a way as to ensure safety). However, we intend to exploit this method in future work to investigate the applicability of our abstraction approach to pre-existing Promela specifications of other parameterised systems.

7.2. Experimental results

Our approach holds for arbitrary verification, but primarily we are interested in *feature interaction analysis*: For a given pair of features f_1 and f_2 check whether a property ϕ defining feature f_1 is violated in the presence of feature f_2.

Below we give experimental results for feature interaction analysis using our approach. All of our experiments were performed on a PC with a 2.4 GHz Intel Xenon processor, 3 Gb of available main memory, running Linux (2.4.18), with Spin version 4.2.3.

In Tables 3 and 4 we give results for analysing example pairs of telephone features, f_1 and f_2, using Spin. The examples chosen are ones which do not interact (that is, the property being checked is *true* in all cases) and therefore can not be fully analysed except for small, finite sized systems, without using our abstraction approach. The first feature, f_1, in all cases is $TCS[0] = 1$ (see Table 1) and ϕ is $[](connected[1].to[0] == 0)$ (no connection from $User[1]$ to $User[0]$ is possible).

In Table 3, all of the feature pairs are subscribed to by the same *User* (*User*[0] in this case) and are therefore referred to single user (SU) pairs. Indices of the second feature are chosen so that the size of the set indexed by ϕ and the pair of features (i.e. m) is 3. For example, when f_2 is *CFU*, the pair of features under consideration is $TCS[0] = 1$ and $CFU[0] = 2$. The index set is $\{0, 1, 2\}$ and $m = 3$.

In Table 4 feature pairs are subscribed to by different Users (known as multi user (MU) pairs); indices are chosen so that $m = 4$.

In all cases we check ϕ for a model with m components, a model with $m+1$ components and an abstract model representing n components, where n is at least $m+1$. Note that in some cases we were unable to check the MU model for $m+1$ components, due to insufficient memory.

Table 3
SU results, telephone ($m = 3$)

	m			$m+1$			n		
f_2	states	mem	time	states	mem	time	states	mem	time
CFU	3.2	0.4	0.1	198.1	8.5	5.8	12.7	0.8	0.4
CFB	5.3	0.5	0.1	409.0	17.1	11.9	17.3	1.0	0.5
ODS	4.4	0.4	0.2	359.7	15.2	10.0	15.1	0.9	0.5
OCS	4.7	0.5	0.9	376.2	15.9	10.7	15.8	0.9	0.5

Table 4
MU results, telephone ($m = 4$)

	m			$m+1$			n		
f_2	states	mem	time	states	mem	time	states	mem	time
CFU	162.9	7.2	4.7	15185.6	689.0	1297.7	720.0	32.9	33.3
CFB	400.7	16.6	15.3	–	–	–	1363.2	59.5	60.7
ODS	376.9	16.0	15.1	–	–	–	1278.8	56.6	58.5
OCS	396.2	16.9	15.6	–	–	–	1303.6	57.9	58.2

States is the number of states ($\times 10^3$) stored during a search, *mem* the memory (in Mb) required for state storage and *time* the the total (user + system) time (in seconds) taken for complete verification. All measurements are given to one decimal place. We use Spin's inbuilt compression algorithm to minimise the memory requirements.

In all cases the cost of model checking the abstract specification (in terms of number of states, memory and time) is less than that for checking a system of fixed size $m + 1$ (and greater than that for a system of fixed size m). Similar results hold for the email example.

8. 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 [29] prove a structural induction theorem for processes using simulation pre-order (see Section 2) to generate an invariant when there is no context process. Similar results are achieved [7,38] 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 [25,4,29,1]. In some cases [36,15] network grammar is used to generate both suitable families and an invariant.

A fully automated approach for verifying parameterized networks with synchronous communication is proposed in [21, 22], and a tool based on the network grammar approach [31] is designed to help in the construction of invariants.

In [9] we introduced our generalisation technique for feature interaction analysis of a telephone system with any number of components. In [10,11] we applied a similar approach to an email system, allowing limited sets of features in abstract components. In [12] 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. In [34] we introduced the concept of safe features and developed a categorisation of safe features. We applied our abstraction approach in the context of feature interaction analysis, giving a detailed analysis of a realistic, featured telephony network.

Here we bring together all results in one comprehensive treatment and illustrate our approach via a set of simple and complex examples.

9. Conclusions

A general technique combining model checking and abstraction is presented, that allows property based analysis of communicating, concurrent systems consisting of an *arbitrary* number of components. The technique is based on leverage of a model checking result about a system of fixed size, to results about systems of arbitrary size. Components do not need to be isomorphic, but their individual behaviour must fulfill criteria which we call *safe*. We present a theorem that expresses how component safety can be ensured by inspection of the form of guards, when components are expressed in guarded command form. The approach is further extended to allow featured components, where features define additional functionality. We extend the notion of safe components to include features, and present a theorem that expresses how component safety can be ensured by inspection of the form of feature guards, when features are expressed in guarded command form.

The main contribution of this paper is to define *safe* components, which ensure that the parameterised model checking problem is solvable, and to prove that *basic* components and components with certain categories of features which conform to syntactic criteria are safe.

Acknowledgements

The authors would like to thank the anonymous referees for their valuable comments on this paper.

References

[1] Parosh Aziz Abdulla, Bengt Jonsson, On the existence of network invariants for verifying parameterized systems, in: Ernst-Rüdiger Olderog, Bernhard Steffen (Eds.), Correct System Design, Recent Insight and Advances, in: Lecture Notes in Computer Science, vol. 1710, Springer-Verlag, 1999, pp. 180–197.
[2] Rajeev Alur, Thomas A. Henzinger (Eds.), Proceedings of the Eighth International Conference on Computer Aided Verification, CAV '96, in: Lecture Notes in Computer Science, New Brunswick, NJ, USA, vol. 1102, Springer-Verlag, July–August 1996.
[3] Krzysztof R. Apt, Dexter C. Kozen, Limits for automatic verification of finite-state concurrent systems, Information Processing Letters 22 (1986) 307–309.
[4] Paul C. Attie, E. Allen Emerson, Synthesis of concurrent systems with many similar processes, ACM Transactions on Programming Languages and Systems 20 (1) (1998) 51–115.
[5] F. Balarin, A. Sangiovanni-Vincentelli, An iterative approach to language containment, in: Costas Courcoubetis (Ed.), Proceedings of the Fifth International Conference on Computer Aided Verification, CAV '93, in: Lecture Notes in Computer Science, vol. 697, Springer-Verlag, Elounda,Greece, June–July 1993, pp. 29–40.
[6] M. Browne, E. Clarke, O. Grumberg, Characterizing finite Kripke structures in propositional temporal logic, Theoretical Computer Science 59 (1988) 115–131.
[7] M.C. Browne, E.M. Clarke, O. Grumberg, Reasoning about networks with many identical finite state processes, Information and Computation 81 (1989) 13–31.
[8] M. Calder, A. Miller, Using SPIN for feature interaction analysis — a case study, in: M.B. Dwyer (Ed.), Proceedings of the 8th International SPIN Workshop, SPIN 2001, in: Lecture Notes in Computer Science, vol. 2057, Springer-Verlag, Toronto, Canada, May 2001, pp. 143–162.
[9] Muffy Calder, 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, IEEE Computer Society Press, Edinburgh, UK, September 2002, pp. 227–230.
[10] Muffy Calder, Alice Miller, Generalising feature interactions in email, in: D. Amyot, L. Logrippo (Eds.), Feature Interactions in Telecommunications and Software Systems VII, Ottawa, Canada, June 2003, IOS Press, Amsterdam, 2003, pp. 187–205.
[11] Muffy Calder, Alice Miller, Detecting feature interactions: How many components do we need? in: Mark Ryan, Dieter Ehrich, John-Jules Meyer (Eds.), Objects, Agents and Features, in: Lecture Notes in Computing Science, Springer-Verlag, 2004, pp. 45–66.
[12] Muffy Calder, Alice Miller, Verifiying parameterised, featured networks by abstraction, in: T. Margaria, B. Steffan, A. Philippou, M. Reitenspiess (Eds.), Proceedings of the first International Symposium on Leveraging Applications of Formal Methods, ISOLA'04, October–November 2004, pp. 227–234.
[13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, Journal of the ACM 50 (5) (2003) 752–794.
[14] Edmund M. Clarke, Orna Grumberg, Doron Peled, Model Checking, The MIT Press, Cambridge, Masachusetts, 1999.
[15] E.M. 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, in: Lecture Notes in Computer Science, vol. 962, Springer-Verlag, Philadelphia, PA, August 1995, pp. 395–407.
[16] E.M. Clarke, O. Grumberg, D Long, Model checking and abstraction, ACM Transactions on Programming Languages and Systems 16 (5) (1994) 1512–1542.
[17] S.J. Creese, A.W. Roscoe, Formal verification of arbitrary network topologies, in: H.R. Arabnia (Ed.), PDPTA'99, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, vol. II, CSREA Press, Las Vegas, Nevada, USA, June–July 1999.
[18] D.L. Dill, The Murϕ verification system, in: Alur and Henzinger [2], 390–393.
[19] A.F. Donaldson, A. Miller, Automatic symmetry detection for model checking using computational group theory. in: Proceedings of the 13th International Symposium on Formal Methods Europe, FME 2005, in: Lecture Notes in Computer Science, Springer-Verlag, Newcastle, UK, July 2005 (in press).
[20] E. Allen Emerson, Vineet 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, in: Lecture Notes in Computer Science, vol. 1831, Springer-Verlag, Pittsburgh, PA, USA, June 2000, pp. 236–254.
[21] E. Allen Emerson, Kedar S. Namjoshi, Automatic verification of parameterized synchronous systems (extended abstract), in: Alur and Henzinger [2], 87–98.
[22] E. Allen Emerson, Kedar S. Namjoshi, Verification of a paramaterized bus arbitration protocol, in: Alan J. Hu, Moshe Y. Vardi (Eds.), Proceedings of the Tenth International Conference on Computer-aided Verification, CAV '98, in: Lecture Notes in Computer Science, vol. 1427, Springer-Verlag, Vancouver, BC, Canada, June–July 1998, pp. 452–463.
[23] Steven M. German, A. Prasad Sistla, Reasoning about systems with many processes, Journal of the ACM 39 (3) (1992) 675–735.
[24] R. Gerth, D. Peled, M.Y. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, in: Proceedings of the 15th international Conference on Protocol Specification Testing and Verification, PSTV '95, Chapman & Hall, Warsaw, Poland, 1995, pp. 3–18.
[25] Mahesh Girkar, Robert Moll, New results on the analysis of concurrent systems with an indefinite number of processes, in: Bengt Jonsson, Joachim Parrow (Eds.), Proceedings of the 5th International Conference on Concurrency Theory, CONCUR '94, in: Lecture Notes in Computer Science, vol. 836, Springer-Verlag, Uppsala, Sweden, August 1994, pp. 65–80.
[26] Gerard J. Holzmann, The model checker Spin, IEEE Transactions on Software Engineering 23 (5) (1997) 279–295.
[27] C. Norris Ip, David L. Dill, Verifying systems with replicated components in Murϕ, Formal Methods in System Design 14 (1999) 273–310.
[28] Y. Keston, A. Pnueli, E. Shahar, L. Zuck, Network invariants in action, in: Lubos Brim, Petr Jancar, Mojmír Kretínský, Antonín Kucera (Eds.), Proceedings of the 13th International Conference on Concurrency Theory, CONCUR 2002, in: Lecture Notes in Computer Science, vol. 2421, Springer-Verlag, Brno, Czech Republic, August 2002, pp. 101–115.
[29] R.P. Kurshan, K.L. McMillan, A structural induction theorem for processes, in: Proceedings of the Eighth Annual ACM Symposium on Principles of Distrubuted Computing, ACM Press, 1989, pp. 239–247.
[30] R.P. Kurshan, Computer-aided Verification of Coordinating Processes: The automata-theoretic approach, in: Princeton Series in Computer Science, Princeton University Press, Princeton, NJ, 1995.
[31] David Lesens, Nicolas Halbwachs, Pascal Raymond, Automatic verification of parameterized networks of processes, Theoretical Computer Science 256 (1–2) (2001) 113–144.
[32] Kenneth L. McMillan, Shaz Qadeer, James B. Saxe, Induction in compositional model checking, in: E. Allen Emerson, A. Prasad Sistla (Eds.), Proceedings of the twelth International Conference on Computer-aided Verification, CAV 2000, in: Lecture Notes in Computer Science, vol. 1855, Springer-Verlag, Chicago, IL, USA, July 2000, pp. 312–327.
[33] K.L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, Boston, 1993.
[34] Alice Miller, Muffy Calder, A generic approach for the automatic verification of featured, parameterised systems, in: M. Reiff-Marganiec, M. Ryan (Eds.), Feature Interactions in Telecommunications and Software Systems VIII, Leicester, UK, June 2005, IOS Press, Amsterdam, 2005, pp. 217–235.

[35] A. Roychoudhury, I.V. Ramakrishnan, Inductively verifying invariant properties of parameterized systems, Automated Software Engineering 11 (2) (2004) 101–139 (extended version of rora1).
[36] Z. Shtadler, O. Grumberg, Network grammars, communication behaviors and automatic verification, in: Sifakis [37], 151–165.
[37] J. Sifakis (Ed.), Proceedings of the International Workshop in Automatic Verification Methods for Finite State Systems, in: Lecture Notes in Computer Science, vol. 407, Springer-Verlag, Grenoble, France, June 1989.
[38] Pierre Wolper, Vinciane Lovinfosse, Properties of large sets of processes with network invariants (extended abstract), in: Sifakis [37], 68–80.