A theorem-proving approach to deciding properties of finite-control agents

Extended abstract

Torkel Franzen
Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden

1 Introduction

In an effort to make this abstract more readable, the subject matter, the work done and the tentative conclusions drawn will first be presented in informal or semi-formal terms, in sections 1-3. The essentials of the formal details are then given in sections 4-8.

This report is concerned with deciding the validity of sequents of the form

(1) $c \vdash A:\phi$

where c is a conjunction of equalities and inequalities between names, A is a finite control Π-calculus agent, and ϕ is an extended μ-calculus assertion about A. These notions will be formally defined below. For the moment we just note that A is a description of a more or less complicated system of communicating processes, for which we have a formally defined notion of how it can change and evolve through acts of communication, including the passing around of communication channels (names), and that ϕ is a more or less complex assertion about the behavior of this system, e.g. that nothing bad can happen in the future (such as deadlock or loss or jumbling of data) or that something good is bound to happen eventually (such as the orderly emission of input data). That a sequent of the form (1) is *valid*

means that for any decision about which names are and which are not equal that conforms to the condition c, A has the property expressed by ϕ.

As established in [1], for finite control agents A, there are essentially only finitely many agents and names that can appear in the course of a series of transitions from A, and the validity of (1) can be mechanically decided, at least in principle, through an exhaustive search procedure. Here we will present one possible route to making this decidability more of a practical proposition. Of course we shouldn't expect too much, in view of the computational complexity of the problem of deciding sequents of the form (1). Whatever the inherent complexity of this problem may be, we can be pretty confident from experience and general reflection that no even remotely efficient general decision procedure is to be expected.

2 **The approach**

It is a fairly obvious approach, since ϕ is a logical formula, to make use of known logical inference systems to devise a system of rules for proving assertions of the form (1), and apply an automatic theorem proving technique to decide the validity of such assertions. Thus the tableau system of [2], from which an automatic model checker for finite-state CCS agents has been derived. When we consider the more complicated case of sequents of the form (1), some further technical problems arise, which is why the algorithm given in [1] proceeds by brute computation rather than by symbolic reasoning. The contribution of the work presented here consists in extending the automatic theorem-proving approach to this case.

The logical formalism on which the inference system is based is that of the classical cut-free sequent calculus. This means that we extend the class of sequents to those of the form

(2) $\quad c \vdash \Gamma$

where Γ is a finite sequence, interpreted as a disjunction, of assertions of the form $A{:}\phi$. That we have no cut rule entails that the system has the subformula property, i.e. the formulas ϕ occurring in the premisses of a rule are subformulas of the formulas occurring in the conclusion. This in turn means that there is an obvious mechanical bottom-up procedure in which

one looks for a proof of a sequent by constructing and trying to prove premisses from which the sequent follows by one of the rules. Improving the efficiency of such a procedure, particularly by finding and eliminating unnecessary backtracking and dead ends, characterizes much work on automatic theorem proving in such systems. This is in contrast to approaches based on trying to make intelligent guesses or combinations.

The rules of the system fall naturally into four classes: logical rules, agent rules, fixpoint rules, and structural rules. In the formulations given here, the rules have been brought fairly close to the implementation, but they should be easily understandable once one has grasped the semantics of the Π-calculus and the μ-calculus (as presented e.g. in [1]).

The decision procedure requires a rather lengthy explanation if it is to be given in full. Below only the essentials will be presented. To arrive at a basic algorithm (lacking optimizations) two basic problems must be solved: how to handle existential quantification in a bottom-up procedure, and how to ensure termination. The first of these problems is handled using methods adapted from [3], which involve the introduction of metalogical variables (taking names as values) and a suspension mechanism. The second problem concerns the fixpoint rules and the rule for using existential formulas. Since the underlying logic is classical, the latter rule requires *contraction*, i.e. the existential formula must sometimes occur not only in the conclusion but as part of the disjunction in the premiss, which means that a bottom-up procedure may loop. The fixpoint rules involve folding and unfolding of fixpoint formulas, and hence yield potential loops in the procedure. The contraction loops can be handled rather simply, but the checking for loops in the fixpoint rules requires a lot of syntactic examination and comparison, as well as checking logical relations between name conditions.

3 **Performance and prospects**

A basic version of the algorithm has been implemented in SICStus Prolog and tested on a set of examples on a Sparc 5 workstation. The results give, I think, a pretty good idea of what can be expected from this approach. The algoritm handles small examples well. To give an idea of the sort of formulas used, consider the example below.

```
  (max L).(all([co(o)]).sigma(z).L and [tau].L and
all([i]).all(z).(L and (max O.[x]).((min
M.[v]).(some({co(o)}).sigma(w).(w=v or M.[v]) or <tau>.M.[v]
or some({i}).some(w).(w#v and M.[v])).[x] and
all([i]).all(w).O.[x] and all([co(o)]).sigma(w).(w=x or
O.[x]) and [tau].O.[x]).[z]))
```

This is a fairly complicated formula (concocted by Joachim Parrow). What it says is that a certain property L holds invariantly, i.e. throughout all possible transitions of the agent at issue, and furthermore, whenever a value `a` is input to the agent, a condition which we may designate `notlost(a)` holds for this value. The meaning of `notlost(a)` is that throughout the future evolution of the agent, as long as `a` has not been output, a condition `possreach(a)` will hold. This condition `possreach(a)` in turn states that there is some possible sequence of transitions leading to `a` being output without previously having been input. Thus the meaning of L is that in a certain weak sense, no input is ever lost: as long as an input has not been output, there is always a *possible* future sequence of transitions leading to it being output.

Now, testing for this property works well with small recursively defined agents, such as a 3-element buffer with around 30 states. (It also works well with non-recursive agents having around 100 states.) That is, the property is checked for within a time span of 1-15 seconds, which is bearable. But when we come to more complicated recursive agents, having around 100 states, the performance of the algorithm is unacceptable.

So the algorithm is satisfactory in being able to handle complicated properties, albeit only for simple agents. But the difficulties experienced by the algorithm in attacking complicated properties for bigger agents are not compensated for by any ease of handling simple properties of bigger agents. For the very general procedure of the algorithm, when set e.g. merely to traverse every state of a big recursive agent, entails a lot of overhead. Thus for example checking big agents for deadlocks using the general algorithm is very inefficient.

Of course the above comments are based on a vanilla version of the algorithm, and there are many more or less obvious optimizations. If we consider how and why the algorithm as sketched below has an edge over brute search, there are just two main points: laziness and clumping together

of cases. (The laziness derives from the suspension mechanism, the clumping together from the symbolic treatment of universal quantification.) By various methods, such as basing the logical part of the algorithm on a static analysis of the agent to be investigated, the performance of the algorithm can be expected to improve considerably. However, the following conclusions seem fairly safe. The algorithm in its full generality can not be expected to be practically useful in any industrial sense. It can however be developed into an attractive pedagogical and experimental tool for gaining familiarity with the Π and μ-calculi, and for testing ideas on small examples. Possibly, stripped-down or specialized versions of the algorithm can be devised that can be useful in the verification of specific properties of large systems, but this remains to be decided.

4 **Agents**

We presuppose an infinite set of *names* x,y,.. each with its *coname* co(x), co(y),.... An *action* α,β,.. is either a name or a coname or the silent action τ. If α is a coname co(x), n(α) is x, otherwise n(α) is α.

Recursive actions and formulas are implemented as infinite (rational) trees, and at least the agents may as well be defined as such from the outset.

An *agent* is either a *process*, an *abstraction*, or a *concretion*. To define these classes, we consider the class of rational trees with nodes annotated with 0 (no successor) or + or | (two successors) or cond([x=y]) (two successors) or one of (λx), (νx), [x], α. for some name x or action α (one successor), and invoke an inductive definition given by the following clauses:

a process is 0, or P+Q, or P | Q, or (νx)P, where P and Q are processes, or α.A where A is an agent;

an abstraction is (λx)A, where A is a process or an abstraction;

a concretion is [x]A, where A is a process or a concretion;

and finally cond(x=y,A,B) is an agent of the same type as A and B, provided A and B have the same type.

These clauses are to be understood as defining the *maximal* fixpoint of the corresponding monotone operator (details omitted). Thus e.g. C is a process if C=u.(C | C).

The set of free names occurring in an agent is defined in the obvious way, given that the name-binding constructs are (λx) and (νx). For technical reasons, we restrict the set of agents further, stipulating that an agent C occurs in (i.e. as a proper subtree of) itself only within the scope of an event, i.e. as a proper or improper subtree of A, where α.A occurs in C.

We will use a syntactic equivalence relation between agents which is a kind of remnant of the ordinary Π-calculus congruence relation: A and B are syntactically equivalent if A is obtainable from B by the operations of alpha-conversion and the replacing of (νx)C by C, where x does not occur free in C.

The result B[y/x] of substituting y for every free occurrence of x in the agent B is defined in the obvious way, with alpha-conversion used if necessary.

If A is an abstraction (λx)B, the application Ay of A to y is defined as B[y/x]. Application is not defined for an A that is not an abstraction.

5 **The commitment relation**

The commitment relation $A>_c\alpha.B$ is defined using the same rules as in [1], with some minor adjustments. These adjustments, which have been made to allow us to make the corresponding formal rules very specific, can be described as follows.

We first define A>α.B for the case where α is not τ. This is done in the standard way, and does not involve c.

Next we define $A>_U\tau.B$, where U is a set which is either empty (in which case we also write A>τ.B) or contains exactly two syntactically different names x,y. This relation is again specified in the ordinary way (with the argument U just being carried along), except for the following rule (and its trivial variants):

$$\frac{A_1 > x.B_1 \quad A_2 > co(y).B_2}{A_1 \mid A_2 >_{\{x,y\}} \tau.(B_1 \bullet B_2)}$$

Here x,y are two syntactically different names, and $(B_1 \bullet B_2)$ is as usual the pseudo-application of B_1 to B_2.

Finally we define $A >_c \alpha.B$ to hold if either $A > \alpha.B$ or $A >_{\{x,y\}} \alpha.B$ where $c \models x=y$.

6 **Formulas**

Formulas, also implemented as rational trees, are more easily described as finite trees. We presuppose an infinite supply of *set variables* X,Y..., each with a specific arity. A *preformula* is either

a literal x=y or x≠y, where x and y are names, or

an application app(X,**x**), where X is an n-place set variable and **x** a sequence of n names, or

a conjunction $(\phi \& \varphi)$ or disjunction $(\phi \vee \varphi)$, or

a universal or existential quantification $\forall x\phi$ or $\exists x\phi$, or

a projection $\Sigma x\phi$, or

a fixpoint formula $(\max X)\phi$ or $(\min X)\phi$.

A formula is now defined as a preformula in which every set variable X occurs in within the scope of a binding operator (max X) or (min X).

The name-binding constructs are here the logical quantifiers and Σx.

7 **Rules**

The inference rules fall into four groups: logical rules, agent rules, fixpoint rules, and structural rules.

Logical rules

Axioms

$c \vdash A{:}\phi,\Gamma$ if $c \vdash \phi$, where ϕ is a literal.

=-rules

$$\frac{x \neq y \wedge c \vdash \Gamma}{c \vdash A{:}x=y,\Gamma}$$

$$\frac{x = y \wedge c \vdash \Gamma}{c \vdash A{:}x \neq y,\Gamma}$$

$\vee$-rule

$$\frac{c \vdash A{:}\phi_1,A{:}\phi_2,\Gamma}{c \vdash A{:}\phi_1 \vee \phi_2,\Gamma}$$

&-rule

$$\frac{c \vdash A{:}\phi_1,\Gamma \quad c \vdash A{:}\phi_2,\Gamma}{c \vdash A{:}\phi_1 \& \phi_2,\Gamma}$$

∀-rule

$$\frac{c \vdash Aw{:}\phi[w/x],\Gamma}{c \vdash A{:}\forall x\phi,\Gamma}$$

where w is new, i.e. does not occur in c, A, ϕ, Γ.

∃-rule

$$\frac{c \vdash Aw{:}\phi[w/x],[A{:}\exists x\phi],\Gamma}{c \vdash A{:}\exists x\phi,\Gamma}$$

where the brackets around [A:∃xϕ] indicate that this expression is optional.

cond-rules

$$\frac{c \vdash A{:}\phi,\Gamma}{c \vdash \mathrm{cond}(x{=}y,A,B){:}\phi,\Gamma} \quad \text{if } c \models x{=}y$$

$$\frac{c \vdash B{:}\phi,\Gamma}{c \vdash \mathrm{cond}(x{=}y,A,B){:}\phi,\Gamma} \quad \text{if } c \models x{\neq}y$$

$$\frac{c \wedge x{=}y \vdash A{:}\phi,\Gamma \qquad c \wedge x{\neq}y \vdash B{:}\phi,\Gamma}{c \vdash \mathrm{cond}(x{=}y,A,B){:}\phi,\Gamma} \quad \text{if } c \text{ does not decide } x{=}y$$

Agent rules

Σ-rules

$$\frac{c \vdash A{:}\phi[x/y],\Gamma}{c \vdash [x]A{:}(\Sigma y)\phi,\Gamma}$$

$$\frac{c \wedge \bigwedge w \neq z \text{ for all old } z \vdash A[w/x]{:}\phi[w/y],\Gamma}{c \vdash (\nu x)[x]A{:}(\Sigma y)\phi,\Gamma}$$

where "old z" means a name z that occurs in c,A,ϕ,Γ.

diamond rule

$$\frac{c \vdash B{:}\phi,\Gamma}{c \vdash A{:}\langle\alpha\rangle\phi,\Gamma}$$

where either α is not τ, A>β.B and $c \models \alpha=\beta$, or α and β are τ and $A>_c\tau.B$.

square rule

$$\frac{c' \vdash B{:}\phi,\Gamma \quad \text{for all } c',B \text{ satisfying the conditions below}}{c \vdash A{:}[\alpha]\phi,\Gamma}$$

Here, if α is not τ, c' is consistent, α and β are both names or both co-names, c' is $c \wedge n(\alpha)=n(\beta)$ and A>β.B or c' is c and A>α.B. If α is τ, c' is consistent and c' is $c \wedge x=y$ where A>{x,y}τ.B or c' is c and A>τ.B.

Fixpoint rules

For the fixpoint rules, we follow [2] and introduce a sequence U1,U2,.. of *special constants* to keep track of fixpoints. Each special contant occurring in a proof is associated with a fixpoint formula as indicated below. This association has been left out of the notation.

Unfold

$$\frac{c \vdash A:app(U,\mathbf{x}),\Gamma}{c \vdash A:(m\ X)\phi,\Gamma}$$

where U is a special constant associated with the fixpoint formula in the conclusion, and **x** is the sequence of names that occur free in ϕ.

Fold

$$\frac{c \vdash A:\phi[\mathbf{w}/\mathbf{y},U/X],\Gamma}{c \vdash A:app(U,\mathbf{w}),\Gamma}$$

where U is associated with $(m\ X)\phi(X)$. Here, if m is max, we are allowed to discharge any assumption above this conclusion of the form $c' \vdash A:app(U,\mathbf{w}),\Gamma',\Gamma$.

Structural rules

Thinning

$$\frac{c \vdash \Gamma}{c \vdash A:\phi,\Gamma}$$

Equivalence

$$\frac{c \vdash A:\phi,\Gamma}{c \vdash A:\phi,\Gamma}$$

if A and A' are syntactically equivalent

Names

$$\frac{c' \vdash \Gamma}{c \vdash \Gamma}$$

if $\exists \mathbf{z} c'$ is equivalent, in the infinite set of names, to $\exists \mathbf{z} c$, where **z** are the variables in c,c' that do not occur in Γ.

8 The algorithm

The algorithm is, as stated, a bottom-up procedure whereby we start with the sequent whose validity is to be decided and apply the rules from conclusion to premisses, looking for a proof. In dealing with the quantifier rules, the same method is used as in [3]. That is to say, in applying the ∀-rule to the conclusion $c \vdash A:\forall x\phi,\Gamma$, we introduce a new name x(i), called a parameter, and attempt to prove the premiss $c \vdash Ax(i):\phi[x(i)/x],\Gamma$. The parameter is annotated with an index i which is incremented after each application of the rule. In applying the ∃-rule to the conclusion $c \vdash A:\exists x\phi,\Gamma$, we introduce a *variable* X(i), annotated with the current index. At various points we attempt to *unify* this variable with a name in such a way as to obtain a proof. To ensure that the unification respects the restriction on the name used in the ∀-rule, X(i) can be unified with a parameter x(j) only if i>j.

The other mechanism taken over from [3] is that of *suspension*. This mechanism is invoked in two situations. First, in trying to prove a sequent $c \vdash X \neq Y,\Gamma$ where at least one of X,Y is an unbound variable (i.e. one that has not been unified with a name). In this case we simply suspend the sequent and go on with the next sequent, if any, that has to be proved.

Second, when trying to prove $c \vdash A:[X]\phi,\Gamma$ or $c \vdash A:[\tau]\phi,\Gamma$, certain premisses may be suspended. Thus e.g. in the case of the first sequent, if X is a name or an unbound variable, any A>Y.A', where the value of at least one of X,Y is a variable, and the values of X and Y are not the same variable, yields a suspended sequent. Suspended sequents are awakened whenever a variable that has justified the suspension becomes bound in a call to the unification algorithm. If no proof of the suspended sequents can be found, the unification will fail.

Looping is avoided in the algorithm through two mechanisms. First, contraction is applied whenever an existential formula is encountered (that is, the formula is duplicated in the premiss sequent), but the duplicate will be used in the continued search for a proof only if new information has appeared on the left hand side of the sequent (in the form of equations or inequations) since the duplication was made. Otherwise that branch of the attempted proof fails. Second, a record is kept of all applications of the fixpoint rules. When the fold rule is applied (from conclusion to premiss) with the same maximal fixpoint formula and an "essentially identical" agent, the branch succeeds. Here "essentially identical" is a somewhat weaker relation than syntactical equivalence. In the case of minimal fixpoints, a more involved comparison is needed, by which it is established that if a minimal fixpoint problem is reduced to one that is at least as general as the original, no proof exists.

References

[1] M. Dam. "Model Checking Mobile Processes". SICS Research Report R94:01. Preliminary version published as "Model Checking Mobile Processes", Lecture Notes in Computer Science 715 (1993) pp. 22-36.

[2] C. Stirling and D. Walker. "Local model checking in the modal mu-calculus." *Theoretical COmputer Science,* 89:161-177, 1991.

[3] Dan Sahlin, Torkel Franzén, Seif Haridi. "An Intuitionistic Predicate Logic Theorem Prover." *J. Logic Computat.,* Vol. 2 No. 5, pp. 619-656 1992.