Enhancing test coverage by back-tracing model-checker counterexamples

A. Fantechi [1,2]

Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
Firenze, Italy

S. Gnesi [1,3]

ISTI-CNR
Pisa, Italy

A. Maggiore [1,4]

AltraVerifica Ltd, UK

Abstract

The automatic detection of unreachable coverage goals and generation of tests for "corner-case" scenarios is crucial to make testing and simulation based verification more effective. In this paper we address the problem of coverability analysis and test case generation in modular and component based systems. We propose a technique that, given an uncovered branch in a component, either establishes that the branch cannot be covered or produces a test case at the system level which covers the branch. The technique is based on the use of counterexamples returned by model checkers, and exploits compositionality to cope with large state spaces typical of real applications.

1 Introduction

Code coverage metrics, such as statement coverage and branch coverage, are largely used in testing and simulation based verification, both for software

[1] This work has been partially supported by the Italian Ministry of University and Research within the COFIN 2001 project "Quack: a platform for the quality of new generation integrated embedded systems"
[2] Email: fantechi@dsi.unifi.it
[3] Email: gnesi@isti.cnr.it
[4] Email: adriana@altraverifica.com

and hardware design, to measure the progress of the verification effort and to identify areas of the design where further tests are needed. Code coverage metrics report on areas of the design which were not exercised during simulation. These will also include portions of code which are unreachable. A common objective of simulation based verification is that of achieving 100% *justified* branch coverage. The process of manually identifying the unreachable portions of code or producing test cases for "corner-case" scenarios is time consuming and error prone. The verification process is made more efficient by automatically and reliably establishing if a coverage goal, like the coverage of a branch in the control flow, is achievable or not. This process is the subject of coverability analysis [14].

A typical situation in which non-coverable goals can be found occurs when the unreachability is the result of constraints imposed on a component by the environment in which it operates. For example, this kind of unreachability can be due to a partial usage of the component's functionality in a particular application. [14] and [15] describe how symbolic model checking and static analysis can be used to calculate coverability at the component level. Coverabiliy at the system level can, in principle, be calculated by applying the same symbolic model checking technique used at the module level, but the size of the state space of real world examples is often beyond the capacity of a general purpose symbolic model checker. In [9] this problem was attacked by means of a-priori abstractions: the state space is reduced by *test-case preserving* abstractions. In [12] SAT-based bounded model checking is applied, together with simulation, to look for reachable states within a bounded depth.

In this paper we propose a method for coverability analysis and test case generation for uncovered branches at the system level which is based on the use of counterexamples returned by model checkers, exploiting compositionality to cope with large state spaces typical of real applications.
Counterexamples are one of the most useful outcomes of formal verification based on model checking [3], and have been extensively used for diagnosis of problems detected in the models of systems. Here a richer notion of counterexample is used, that is, the notion of "counterexample automaton", which expresses *all* finite linear counterexamples of a given formula on a given model.

The paper is organized as follows. Section 2 introduces the proposed technique in general terms. Section 3 introduce the framework (formalism, logic and tools) used in our proposal and describes by means of an example the method. The complexity of the method is then discussed in 4. The counterexample automaton generation algorithm is reported in the appendix.

2 The proposed technique

2.1 The basic principle

The technique we propose to check coverability and derive test cases from uncovered branches of a system is based on the following principles.

- we suppose that the system can be modeled as a finite state Labelled Transition System (LTS) [13], and that the testing process is able, through proper tools, to provide both the coverage measure and information about the uncovered branches.
- starting from each uncovered branch, we build a temporal logic formula expressing the property: "the uncovered branch can never be reached"
- we apply a model checker to the LTS which models the system, to check the given temporal logic formula on the model: if this returns TRUE, the branch cannot be covered;
- otherwise, we ask the model checker for a counterexample, which is a path that exercises the uncovered branch: this path contains information about the input needed to exercise it - that is, the sought test case.

2.2 Compositionality

This apparently simple process is however complicated by the fact that real applications have very large state spaces, and hence model checking becomes soon unfeasible. The technique we propose exploits compositionality to address this problem:

- We assume that the system is composed of a chain of modules (see Fig. 1), and that the uncovered branch we want to address can be localized in the inner module S_0. Note that this assumption is less restrictive than what can appear at a first sight: indeed, a system where components interact with a more complex structure can be sliced in modules according to this assumption, by grouping more components in a single module (this is why in the following we refer to "modules" rather than to "components"); moreover, this structure is typical of the client-server interaction paradigm.
- We apply the process described in sect. 2.1 to the module S_0, obtaining a path which shows which inputs the module needs to exercise the uncovered branch. We assume that such inputs to S_0 come, as outputs, by the previous module S_1.
- From this sequence of outputs of S_1, we then elaborate a temporal logic formula ϕ_1 expressing the property: "it is never possible to produce such an output sequence".
- We apply the model checker on the model of the module S_1 to check the formula ϕ_1: again, this should return FALSE (if not, again, the sequence of outputs of S_1 is unfeasible and hence the uncovered branch in S_0 is not

reachable).

- We ask the model checker for a counterexample. This counterexample is a path of S_1 which gives the output sequence that exercises the uncovered branch in S_0. This path also contains information about the input needed to exercise it.
- The system being structured as a chain of modules, each one producing input for the following one and receiving the output of the previous one, we can repeat the application of the process above to each module encountered. The final counterexample, produced on S_n is actually the sought test case for the interface of the module S_n, which when given as input to S_n causes the intermediate counterexamples to be given as input to the next modules, which finally causes the uncovered branch in S_0 to be exercised.

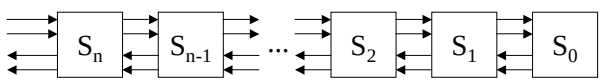

Fig. 1. A chain of modules

2.3 Coping with false counterexamples

It may be the case that the single counterexample trace returned by the method above for a given module does not correspond to any feasible path of the previous module in the chain. This does not mean that no test case exists to cover the uncovered branch. Actually, the model checker has produced a single counterexample trace, which has shown to be not executable under the input sequences that can be provided by the previous module. We need therefore not to extract a single counterexample, but *all* the possible counterexample traces: actually, the number of the possible counterexample traces may be infinite. We are interested therefore to a finite representation of the possible counterexamples, and this can be achieved by considering that the set of the counterexamples is the language generated by the "counterexample automaton". A counterexample automaton for a LTS A and a formula phi is an automaton which recognizes the language of all finite linear counterexamples of phi on A.

As a next step, from the counterexample automaton we need to extract a formula expressing "there exists in the model no path with a sequence of actions which is recognized by the counterexample automaton", but exchanging the role of corresponding input and output actions, and not observing other actions not involved in the interaction between the considered modules. In the following we will also refer to this formula as "counterexample formula".

At this point, we can pass to model check the next module for satisfaction of this counterexample formula, in order to obtain a new counterexample

automaton and then a new counterexample formula for the next module, and so on, repeating this process until we arrive at the borders of the system, that is, the last module, for which any linear counterexample provides a test case that is the desired test case. This case is to be used directly to test the whole module chain, and it will cover the originally uncovered branch.

2.4 The overall test case generation procedure

The overall approach can be described by the following procedure, the details of which will be discussed in the next section by means of a running example, and with reference to a particular modeling formalism and verification environment.

We assume a chain structure such as that represented in Figure 1: G_i the set of common actions between S_i and S_{i+1}. ϕ_i is the temporal logic formula that is checked at the i-th step. ϕ_0 will therefore be the formula expressing: "the uncovered branch can never be reached". The algorithm calls the following operators:

- *MC*, which *model checks* a formula on a LTS, giving a boolean result
- *AC*, which calculates the *counterexample automaton* from a formula and a LTS,
- *FC*, indexed on a set of communication actions, which calculates the *characteristic formula* of a LTS, relative to the given set of actions,
- $A/_G B$, an indexed synchronization operator used to filter out spurious counterexamples.

```
  procedure testcasegen(φ0, S0)
if MC(φ0, S0) = true
then "unfeasible path"
else AC0 := AC(φ0, S0)
   φ1 := FC_G0(AC0)
   for i := 1,n do
      if MC(φi, Si) = true
      then "unfeasible path"
      else
         ACi := AC(φi, Si)/_Gi-1 ACi-1
         if i ≠ n then φi+1 := FC_Gi(ACi)
   "any path of ACi is a test case for the system"
```

3 Feasibility of the approach with JACK

In the following, in order to show the feasibility of the approach, we develop the technique described above basing on the explicit model checker AMC

included in the integrated verification environment JACK [1] [5]. Hence, we inherit from JACK the formalisms in which models are described (Labelled Transition Systems) and the logic in which properties can be described, namely ACTL (Action-based Computation Tree Logic) [7], which is an action-based version of the branching time temporal logic CTL [3].

3.1 Labelled Transition Systems

We use the graphical notation of Labelled Transition Systems (LTSs) and networks of LTSs inherited in JACK from Autograph [16].

An example of a LTS is reported in Figure 2. The initial state of the LTS is represented by a double circle and labels are associated to edges representing transitions of the LTS.

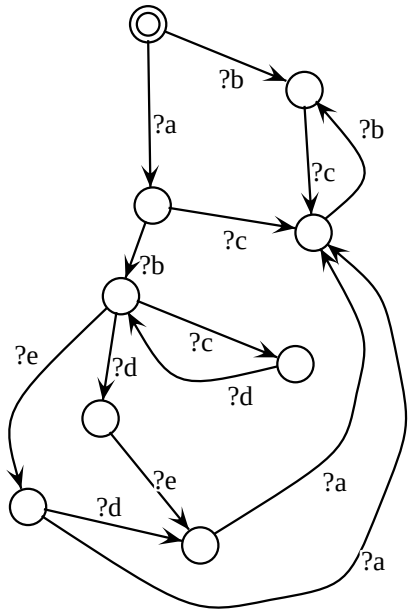

Fig. 2. The graphical specification of an example automaton (Mod0)

To express synchronous communication between two LTSs we use the graphical notation of networks. A process surrounded by a box is said to be a network and the ports at the border are its places of interconnection. If two networks are drawn at the same level, they can synchronize via the actions they execute by linking the corresponding ports. In this case the actions executed at the linked ports are no more observable, and the silent τ action is shown when they are executed.

The graphical representation of an example network is reported in Fig. 3.

[5] Detailed information about the environment are available at http://fmt.isti.cnr.it/fmt-tools.htm

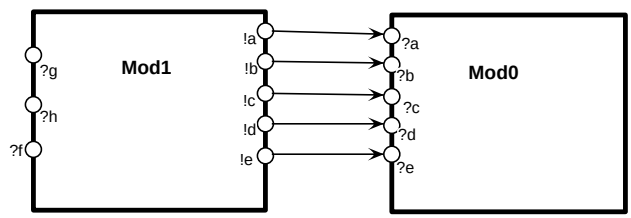

Fig. 3. An example network

3.2 The logic

The particular temporal logic we use is called ACTL (Action-based Computation Tree Logic) [7], which is an action-based version of the branching time temporal logic CTL [3]. ACTL is based on actions rather than states, and hence it is naturally interpreted over LTSs.

We use in the following a subset of the ACTL logic, which includes all the formulae which are useful for the test case generation process. Indeed, what we need are formulae of the kind: "the uncovered branch can never be reached" and "there exists no path with a sequence of actions recognized by the counterexample automaton". Limiting to formulae of these kinds allows the notion of counterexample itself and the counterexample automaton to be defined in a much more accurate and effective manner.

We can observe that all the formulae that interest us are of the kind: $\sim \phi$, where $\sim$ is the *negation* operator, and ϕ is an existential formula.

The formulae ψ we will use are hence defined according the following syntax:

$\psi ::= \sim \phi$

$\phi ::= \ \phi|\phi \ \mid \ \phi \Rightarrow \phi \ \mid \ EX\{act\}\ \phi \ \mid \ EG\phi \ \mid \ EF\phi \ \mid \ << actfor >> \phi$

The syntax of the ϕ formulae is a subset of the positive existential fragment of ACTL, including the propositional disjunction and implication, the existential next operator ("there exists a next state reachable with an action *act* and which satisfies ϕ"), the usual existential *always* and *eventually* operators, and a new modality that absorbs all the input actions which occur before the output actions considered in the given action formula. An action formula is a propositional combination of actions. The informal meaning of $<< actfor >> \phi$ is: "there exist a path composed of input and/or unobservable actions, but for the last action satisfying the action formula $actfor$, which reaches a state where ϕ holds".
We refer to [7] for the formal definition of the previous operators, but for the $<< actfor >> \phi$ modality, which is actually derived from the ACTL Until operator. as

Since we use the negation only at the beginning of a formula, counterexamples are actually "witnesses" [4] of the corresponding positive formula. It can be seen that all the formulae of kind ψ, when not satisfied on a LTS,

admit linear counterexamples; correspondingly, ϕ formulae, when satisfied on a LTS, admit linear witnesses.

3.3 A test case generation example

We show the overall test case generation procedure by means of a simple example, a system composed by two modules, as represented by the network in Figure 3. The modules `Mod0` and `Mod1` are defined respectively by the LTSs in Figures 2 and 4.

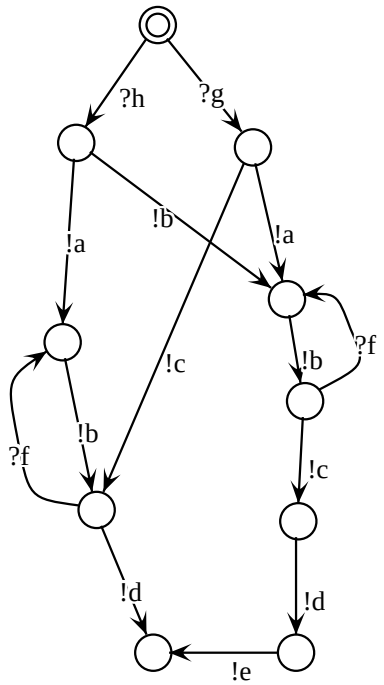

Fig. 4. The graphical specification of the module Mod1

We assume that a testing activity has not covered the branch labelled by the action $?e$, (after $?a$ and $?b$) in Mod0.

We modify the LTS of Mod0 by adding a transition just after the uncovered branch, labelled with the fresh action $!k$ (Figure 5)

The property: "the action k is never possible" is represented by the formula: $\sim EFEX\{!k\}true$

Applying the model checker AMC to verify this formula on Mod0 we obtain, as expected, a negative answer. We then generate the *witness* automaton for the formula $EFEX\{!k\}true$. A witness automaton for a LTS A and a formula phi is an automaton which recognizes the language of all finite linear witnesses of phi on A. The algorithm that extracts the witness automaton from the labelling of the states produced by the model checker, is reported in the Appendix. The automaton AC_0 we obtain is therefore the counterexample automaton of the formula: $\sim EFEX\{!k\}true$ and is represented in Figure 6.

We need now to provide a formula expressing "there exists no path with a

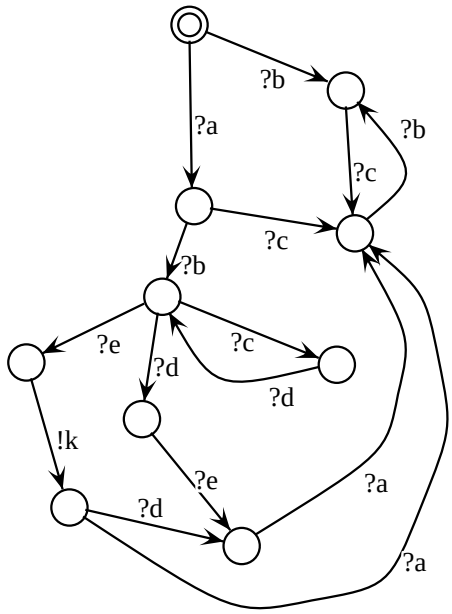

Fig. 5. Adding a transition to mark the uncovered branch in Mod0

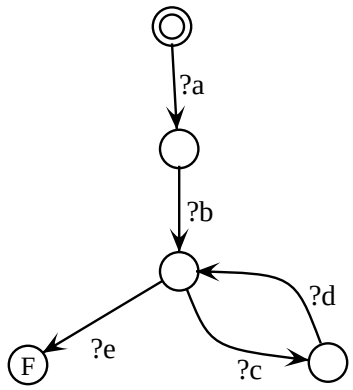

Fig. 6. The counterexample automaton AC_0

sequence of actions recognized by the counterexample automaton". This can be achieved by giving the *characteristic formula* of the automaton [2,17], that is, a formula which describes completely the automaton itself. Actually, we need only the existential part of the characteristic formula, and we adopt the method shown in [8] to give an ACTL characteristic formula, exploiting the notion of *implicit fixed point*, which requires no explicit fixed point operators. A description of the algorithm that calculates the characteristic formula is reported in the Appendix.

Back to our running example, from the automaton AC_0 we derive the formula FC_G, with $G = (a, b, c, d, e)$:

```
<<!a>><<!b>>((<<!e>>true) |
    (EG ( EX{!c} true => (EX{!c}<<!d>><<!e|!c>>true))))
```

In the second step we apply then the model checker to the Mod1 LTS and

to the formula:

```
~<<!a>><<!b>>((<<!e>>true) |
   (EG ( EX{!c} true => (EX{!c}<<!d>><<!e|!c>>true))))
```

and we then obtain the counterexample automaton AC_1 (Figure 7).

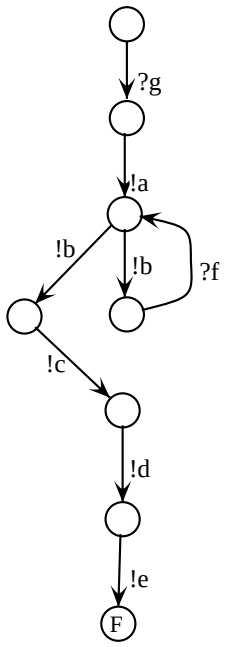

Fig. 7. The counterexample automaton AC_1

The counterexample automaton AC_1 shows a loop with `!b` and `?f` actions, which would correspond to a cycle of `?b` on Mod0, which is not feasible; only a single `?b` action is indeed performed by Mod0 at that point. This means that AC_1 generates false counterexamples which should be avoided.

This phenomenon is due to the fact that, within ACTL, it is not possible to predicate a complex formula on a path without using state formulae, each of which should be individually quantified. Hence it is not possible to express a predicate of the type "there exists a path having a complex behaviour" but only "there exist a path, which, after a simple behaviour, reaches a state from which there exist a path....". This means that computing the Characteristic Formula we introduce spurious traces.

In order to cut this kind of false counterexamples we should consider only the paths of AC_1 that correspond to paths of AC_0. This operation is essentially a synchronization operation between AC_0 and AC_1 on the common actions. We will use for this operation the notation $AC_1/_G AC_0$, where G is the set of common actions, in our case `(a,b,c,d,e)`. A formal description of the operator is reported in the Appendix.

Actually, after this operation, we still need to cut away all the terminating branches not ending in a final state. If this is done in the example, the

operation produces the automaton represented in Figure 8.

Fig. 8. The automaton $AC_1/_G AC_0$

The obtained counterexample reduces to the path $?g; !a; !b; !c; !d; !e.$, from which we extract the input sequence formed only by $?g$, which is the test case that we were looking for.

Notice that producing a single linear counterexample at the first step, instead of the counterexample automaton AC_0, could have produced at the end the path: $?g; !a; !b; !c;$, which is actually the shortest counterexample, but is not feasible in the Mod1 LTS.

This process can be repeated, as long as we have modules connected in a chain; for each intermediate module the counterexample automaton and formula should be calculated and the latter should be checked on the next module. For the last module the counterexample automaton, once false counterexamples have been filtered out, defines a set of test cases, each covering the originally uncovered branch.

4 Complexity of the procedure

The following elements add up to the computational complexity of the procedure:

- Model checking (MC) is linear with the product of state space size times the length of the formula (that is, the maximum nesting of operators);
- The length of characteristic formula (FC) is linear in the size of the automaton. Linear as well is the complexity of the characteristic formula generation;

- Synchronization of automata ($/_G$) has a complexity of at most the product of the sizes of the automata; here it is applied to two successive counterexample automata;
- Counterexample automata (AC) tend to be small, since they generate new test cases: if we assume that the approach is applied only when the "easiest" test cases have already been exercised and only "corner-case" test cases remain to be discovered, the counterexample automata is a small sub-automaton of the considered module (not really a sub-automaton, because some loops may be unfolded depending on the length of the formula).
- The model complexity is anyway attacked compositionally

The order of complexity is therefore $n*m*c$, where n is the number of modules in the chain, m the (average) state space of a module, c the (average) state space of a counterexample automaton. Note that this may be substantially better than $n * m * m$, due to the generally low dimensions of the counterexample automata, especially for "corner-case" counterexamples.

We claim that the above procedure is of minimal complexity achievable basing on explicit state space enumeration, if compared with a similar procedure that could have been defined using the explicit synchronization of two successive modules instead of the characteristic formula plus model checking process.

5 Conclusions and Further Work

We have detailed the proposed approach using a single running example, and particular formalisms and verification tools. Nevertheless, we believe that the approach has a general validity. Work is in progress on implementations of the approach both using explicit model checking (as shown in the paper) and BDD-based symbolic model-checking, but still focusing on LTSs and on action-based temporal logics.

It seems reasonable that the approach works as well with a state based formalism (Kripke Structures) and a state based temporal logic (such as CTL). This needs to be verified: the very definition of counterexample, witness and counterexample automaton is actually highly sensitive to the logic used and to the assumptions on the models.

A result which is related to our work is the definition of more expressive *tree-like* counterexamples for Kripke Structures and CTL; such counterexamples are used as a support to guide a refinement technique [5]. The main difference with respect to our approach is that a tree-like counterexample is in its entirety a proof that the formula is not valid. Our counterexample automaton gives instead the set of all linear counterexamples, each of which can be taken separately as a traditional counterexample. A recent evolution of tree-like counterexamples is represented by *proof-like* counterexamples [11], used to extract proofs for the non satisfiability of a formula over a model. Closer

to our approach is the multiple counterexamples generation of [6,10], which generates all the counterexamples to a given length, expressed as a single counterexample trace annotated with possible values of binary variables.

References

[1] A. Bouali, S. Gnesi, S. Larosa. The integration project for the JACK environment. Bulletin of the EATCS, n. 54, October, 1994, pp. 207-223.

[2] M.C. Browne, E.M. Clarke, O. Grumberg: *Characterizing Finite Kripke Structures in Propositional Temporal Logic*, Theoretical Computer Science, 59 (1,2), 1988, pp. 115-131.

[3] E.M. Clarke, E.A. Emerson, A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. ACM Transaction on Programming Languages and Systems, vol.8, n. 2, 1986, pp. 244-263.

[4] E. Clarke, O. Grumberg, K. McMillan, X. Zhao, Efficient generation of counterexamples and witnesses in symbolic model checking, 32nd design Automation Conference, DAC'95.

[5] E.M. Clarke, S. Jha, Y. Lu, H. Veith. Tree-like Counterexamples in Model Checking. 17th IEEE Symposium on Logic in Computer Science (LICS'2002), pp. 19-29.

[6] F. Copty, A. Irron, O. Weissberg, N. Kropp, G. Kamhi. Efficient Debugging in a Formal Verification Environment, CHARME'01, LNCS 2144, pp. 275-292,Springer-Verlag, 2001.

[7] R. De Nicola, F.W. Vaandrager. Actions versus State Based Logics for Transition Systems. Proc. Ecole de Printemps on Semantics of Concurrency, Lecture Notes in Computer Science vol. 469, Springer, Berlin, 1990, pp. 407-419.

[8] A. Fantechi, S. Gnesi, G. Ristori. Modelling Transition Systems within an Action Based Logic. IEI Technical Report, 1996.

[9] D. Geist, M. Farkas, A. Landver, Y. Lichenstein, S. Ur, Y. Wolfsthal. Coverage-Directed Test Generation Using Symbolic Techniques. First International Conference on Formal Method in Computer-Aided Design, LNCS 1166, Springer-Verlag, 1996.

[10] M. Glusman, G.Kamhi, S. Mador-Heim, R. Fraer, M. Vardi, Multiple-Counterexample Guided Iterative Abstraction Refinement: An Industrial Evaluation. TACAS 2003, LNCS 2619, pp. 176-191, Springer Verlag, 2003.

[11] A. Gurfinkel, M. Chechik. Proof-Like Counter-Examples, TACAS 2003, LNCS 2619, pp. 160-175, Springer Verlag, 2003.

[12] P.H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, J. Long, Smart Simulation Using Collaborative Formal and Simulation Engines, ICCAD'00.

[13] R. Milner. Communication and Concurrency. Prentice-Hall International, Englewood Cliffs, 1989.

[14] G. Ratzaby, S. Ur, Y. Wolfsthal, Coverability Analysis Using Symbolic Model Checking, CHARME 2001, LNCS 2144, Springer-Verlag, 2001.

[15] G. Ratsaby, B.Sterin, S.Ur, Improvements in Coverabiliy Analysis, FME 2002, LNCS 2391, Springer-Verlag, 2002.

[16] V. Roy, R. De Simone. AUTO and Autograph, in Proceedings of the Workshop on Computer Aided Verification, LNCS 531, 65-75, 1990.

[17] B. Steffen, *Characteristic Formulae*, Proceedings 16th ICALP, Lecture Notes in Computer Science, vol. 372, pp.723-732, 1989.

Appendix

Counterexample automaton

We give here an algorithm that, given a formula ϕ and a LTS A, on which a labelling has been computed by an explicit model checker during the successful check of ϕ, computes an automaton WA whose recognized language is the set of finite linear witnesses of the formula ϕ over the LTS A. We assume the formula ϕ belongs to the positive existential logic defined in sect 3. We will call this automaton the "witnesses automaton".

The algorithm proceeds by visiting the portion of the state space of A which is labelled by sub-formulae of ϕ; the visit is guided by the structural analysis of the formula itself, hence it is terminated when the leaves of the formula are reached (notice that for the used logic, the leaves are always the `true` sub-formula). If needed, A is unfolded if a sequence of actions in ϕ matches with a loop in A. The visit is implemented by a depth-first search by recursion, and hence suitable information have to be associated to each state, to say whether the state has already been visited, and with which sub-formula ϕ).

The algorithm is defined in a Pascal-like style, and uses some auxiliary functions and procedures which are defined first, employing shared structures that represent the automata and the information related to them (labelling by the model-checker, bookkeeping information, etc. . .).

The auxiliary procedures work on A and WA as global variables, therefore working also on the final states set F and on the relation R, both formally included in WA. The `build` procedure builds a new transition and a new state in WA, corresponding to a transition and state in A. The `condbuild` procedure calls the previous one, only in the case that the state in A has not yet been visited.

procedure build (**in** s: State, a: Act, s1: State, t: Wstate; **out** t1: Wstate)
begin
  create a new state t1 in WA;
  add the transition t $\xrightarrow{a}$ t1 in WA;
  add the pair (t1,s1) to R.
end

procedure condbuild (**in** s: State, a: Act, s1: State, t: Wstate; **out** t1: Wstate)
begin
  if s1 has already been visited **then**
    t1 := Wstate associated to s1 in R;
  else
    build(s, act, s1, t, t1)
end

The following are the main functions, which also work on A and WA (therefore including F and R) as global variables.

procedure WAgenINIT (**in** s: State, ϕ: TL) – (s is the initial state)
begin
  create a new state t in WA;
  t is the initial state of WA;
  add the pair (t,s) to R;
  WAgen (s, t, ϕ).
end

procedure WAgen (**in** s: State, t: Wstate, ϕ: TL)
begin
 if s, t have not been already visited with ϕ **then**
 begin
    record that s, t have been visited with ϕ;
   case ϕ:

    true: add t to F;

    $\phi_1|\phi_2$: **if** $\phi_1 \in L(s)$ **then** WAgen(s,t,ϕ_1);
              if $\phi_2 \in L(s)$ **then** WAgen(s, t, ϕ_2);

    $\phi_1 \Rightarrow \phi_2$: **if** $\phi_1 \in L(s)$ **then** WAgen(s, t, ϕ_2);

    EF ϕ_1: **if** $\phi_1 \in L(s)$ WAgen(s, t, ϕ_1);
      forall s': State, act such that s $\xrightarrow{act}$ s' $\in$ A and EF $\phi_1 \in L(s')$ **do**
      begin

```
        condbuild(s, act, s', t, t')
        WAgen(s', t', EF φ1);
      end;

EG φ1:     WAgen(s, t, φ1);
      forall s' : State such that s --act--> s' ∈ A do
      begin
        condbuild(s, act, s', t, t')
        WAgen(s', t', EG φ1);
      end;

EX{act} φ1:   forall s' : State such that s --act--> s' ∈ A and φ1 ∈ L(s') do
      if s' has not been visited with φ1, then
      begin
        build(s, act, s', t, t')
        WAgen(s', t', φ1);
      end;
      else
      begin
        taken t' the Wstate associated to s' in the last visit with φ,
        add the transition t --act--> t1 in WA;
      end;

<< χ >> φ1: if    s --act-->, act satisfies χ then
      forall s' : State such that s --act--> s' ∈ A and φ1 ∈ L(s) do
        if s' has not been visited with φ1, then
        begin
          build(s, act, s', t, t')
          WAgen(s', t', φ1);
        end;
        else
        begin
          taken t' the Wstate associated to s' in the last visit with φ,
          add the transition t --act--> t1 in WA;
        end;
    else
     forall s': State, act such that s --act--> s' ∈ A and << χ >> φ1 ∈ L(s')
      do
      begin
        condbuild(s, act, s', t, t');
        WAgen (s', t', << χ >> φ1);
      end;
```

The automaton WA we obtain is deterministic, due to the determinism we assume of the original LTS A. For applying the definition of Characteristic Formula given in sect. 3.3 we actually need a non minimal, nondeterministic equivalent automaton obtained by the following rule:
for any final state s $\in$ F with outgoing transitions, such that r $\xrightarrow{a_1}$ s $\xrightarrow{a_2}$ p, split s in two states s' and s" such that:

- r $\xrightarrow{a_1}$ s', s' $\in$ F, s'$\not\rightarrow$
- r $\xrightarrow{a_1}$ s" $\xrightarrow{a_2}$ p, s" $\notin$ F

Characteristic formula

The algorithm FC_G to produce the characteristic formula can be described informally (that is, without detailing particular cases) as follows:
We visit each state s of the counterexample automaton A, starting from the initial state, building the formula with the following rules, where G is the set of actions on which the two modules of interest synchronize:

- $FC_G(s) = true$ if $s \in F$ (F is the set of final states of A)
- $FC_G(s) = (\bigvee_{a:s \xrightarrow{a} s",a \in G} << a >> FC_G(s")) \mid (\bigvee_{a:s \xrightarrow{a} s",a \notin G} FC_G(s"))$
  $\mid EG(EX\{b\}true \Rightarrow$
  $EX\{b\}((\bigvee_{a:s' \xrightarrow{a} \hat{s},a \in G} << a >> FC_G(\hat{s}))|(\bigvee_{a:s' \xrightarrow{a} \hat{s},a \notin G} FC_G(\hat{s}))))$
  if s is the first state of a loop, s" stands for any next state of s not belonging to the loop, b is the first action in G of the loop, such as $s \xrightarrow{b} s'$, and $\hat{s}$ stands for any of the next states of s'.
- $FC_G(s) = true$ if s a first state of a loop, and has already been visited by the above clause (that is, we are ending a loop)
- $FC_G(s) = (\bigvee_{a:s \xrightarrow{a} s',a \in G} << a >> FC_G(s)) \mid (\bigvee_{a:s \xrightarrow{a} s',a \notin G} FC_G(s))$ otherwise.

Filtering synchronization operator

This operator $AC_1/_G AC_0$ can be defined operationally as follows:

$$\frac{AC_1 \xrightarrow{\alpha} AC_1\prime \quad AC_0 \xrightarrow{\alpha\prime} AC_0\prime}{AC_1/_G AC_0 \xrightarrow{\alpha} AC_1\prime/_G AC_0\prime} \quad if \quad \alpha = !a, \alpha\prime = ?a \ or \ \alpha = ?a, \alpha\prime = !a \ and \ a \in G$$

$$\frac{AC_1 \xrightarrow{*a} AC_1\prime}{AC_1/_G AC_0 \xrightarrow{*a} AC_1\prime/_G AC_0} \quad \text{if } a \notin G, * = !, ?$$