Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

407

J. Sifakis (Ed.)

Automatic Verification Methods for Finite State Systems

International Workshop, Grenoble, France
June 12–14, 1989
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo Hong Kong

Editor

Joseph Sifakis
LGI-IMAG, BP 53X
F-38041 Grenoble Cedex, France

CR Subject Classification (1987): C.2.2, C.3, D.2.4, F.3–4

ISBN 3-540-52148-8 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-52148-8 Springer-Verlag New York Berlin Heidelberg

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210 – Printed on acid-free paper

PREFACE

This volume contains the proceedings of the workshop on Automatic Verification Methods for Finite State Systems held at Grenoble from 12 to 14 June 1989. The workshop was organised on the initiative of Ed Clarke, Amir Pnueli and the Editor. It was sponsored by C-cube, the French National Project on Concurrency. Its technical organisation was supported by the IMAG Institute.

This workshop is the first international meeting entirely devoted to the verification of finite state systems. Its organisation has been motivated by the growing interest in this problem due to the conjunction of two independent facts. First, finite state models are very often used to represent complex concurrent systems or their abstractions in several application areas such as hardware, protocols or systems of real-time control. Second, the emergence over the last decade of specification formalisms with well established underlying verification theories such as process algebras and temporal logics.

The workshop brought together 120 researchers and practitioners interested in the development and the use of methods, tools and theories for automatic verification of finite state systems. The goal of the workshop was the comparison of various verification methods for finite state systems, and tools to assist the application designer. The emphasis was not only on new research results but also on the application of existing results to real verification problems.

The material included was prepared by the lecturers after the meeting took place. A few lecturers failed to provide their manuscript on time and their contribution is not in this volume. The proceedings are organised in 5 parts corresponding to sessions of the workshop. Each part consists of a collection of long papers followed sometimes by a collection of short papers.

One may feel that the classification induced by this organisation is arbitrary, as one paper may concern several parts; however, I believe that such a presentation helps the reader to appreciate the importance and applicability of results for each approach and domain.

Part 1 is dedicated to verification methods and tools for process algebras and systems of communicating processes. Most papers present verification tools for the comparison of transition systems modulo some equivalence relation by using model reduction or axiomatic techniques.

Part 2 is a collection of papers on model checking for both linear and branching time temporal logics.

Part 3 concerns the specification of timed systems.

Parts 4 and 5 contain papers dealing with the application of verification techniques in two domains, respectively, protocol validation and hardware verification.

October 1989 Joseph Sifakis

CONTENTS

Process Algebras and Systems of Communicating Processes
G. Boudol, V. Roy, R. de Simone, D. Vergamini
Process Calculi, from Theory to Practice: Verification Tools *1*

R. Cleaveland, M. Hennessy
Testing Equivalence as a Bisimulation Equivalence *11*

R. Cleaveland, J. Parrow, B. Steffen
The Concurrency Workbench *24*

F. Maraninchi
Argonaute: Graphical Description, Semantics and Verification of Reactive Systems by Using a Process Algebra *38*

R. De Nicola, P. Inverardi, M. Nesi
Using the Axiomatic Presentation of Behavioural Equivalences for Manipulating CCS Specifications *54*

P. Wolper, V. Lovinfosse
Verifying Properties of Large Sets of Processes with Network Invariants *68*

I. Christoff
A Method for Verification of Trace and Test Equivalence *81*

H. Krumm
Projections of the Reachability Graph and Environment Models *89*

H. Tuominen
Proving Properties of Elementary Net Systems with a Special-Purpose Theorem Prover *97*

H. Zuidweg
Verification by Abstraction and Bisimulation *105*

Model Checking

A. Arnold
MEC: A System for Constructing and Analysing Transition Systems 117

H. Barringer, M.D. Fisher, G.D. Gough
Fair SMG and Linear Time Model Checking 133

Z. Shtadler, O. Grumberg
Network Grammars, Communication Behaviors and Automatic Verification 151

C. Stirling, D. Walker
CCS, Liveness, and Local Model Checking in the Linear Time Mu-Calculus 166

B. Jonsson, A.H. Khan, J. Parrow
Implementing a Model Checking Algorithm by Adapting Existing Automated Tools 179

C. Jard, T. Jeron
On-Line Model Checking for Finite Linear Temporal Logic Specifications 189

Timed Specifications

D.L. Dill
Timing Assumptions and Verification of Finite-State Concurrent Systems 197

N. Halbwachs, D. Pilaud, F. Ouabdesselam, A-C. Glory
Specifying, Programming and Verifying Real-Time Systems
Using a Synchronous Declarative Language 213

K.G. Larsen
Modal Specifications 232

J.S. Ostroff
Automated Verification of Timed Transition Models 247

W.G. Wood
Temporal Logic Case Study 257

Protocol Validation

S. Aggarwal, D. Barbara, W. Cunto, M. Garey
The Complexity of Collapsing Reachability Graphs 264

S. Graf, J.L. Richier, C. Rodriguez, J. Voiron
What are the Limits of Model Checking Methods for the Verification of Real Life Protocols? 275

P. Azema, F. Vernadat, J.-C. Lloret
Requirement Analysis for Communication Protocols 286

J. Quemada, S. Pavon, A. Fernandez
State Exploration by Transformation with LOLA 294

M.C. Yuang, A. Kershenbaum
Parallel Protocol Verification: The Two-Phase Algorithm and Complexity Analysis 303

Hardware Verification

A. Bronstein, C.L. Talcott
Formal Verification of Synchronous Circuits based on String-Functional Semantics: The 7 Paillet Circuits in Boyer-Moore 317

J.R. Burch
Combining CTL, Trace Theory and Timing Models 334

J. Staunstrup, S.J. Garland, J.V. Guttag
Localized Verification of Circuit Descriptions 349

O. Coudert, C. Berthet, J.C. Madre
Verification of Synchronous Sequential Machines Based on Symbolic Execution 365

G.C. Gopalakrishnan, N.S. Mani, V. Akella
Parallel Composition of Lockstep Synchronous Processes for Hardware Validation: Divide-and-Conquer Composition 374

Process Calculi, from Theory to Practice: Verification Tools

Gérard Boudol
Valérie Roy *
Robert de Simone
Didier Vergamini †
I.N.R.I.A.
Route des Lucioles
Sophia-Antipolis
06561 Valbonne CEDEX
France

Abstract

We present here two software tools, AUTO and AUTOGRAPH. Both originated directly from the basic theory of process calculi. Both were experimented on well-known problems to enhance their accordance to users expectations.

AUTO is a verification tool for process terms with finite automata representation. It computes minimal normal forms along a variety of user parameterized semantics, including some taking into account partial observation and abstraction. It checks for bisimulation equivalence (on the normal forms), and allows powerful diagnostics methods in case of failure.

AUTOGRAPH is a graphical, non syntactic system for manipulation of process algebraic terms as intuitively appealing drawings. It allows graphical editing by the user, but also visual support for display of information recovered from analysis with AUTO.

1 Introduction

The theory of process calculi as started with CCS [Mil 80] resulted in a number of verification tools designs, mostly in the case of terms with finitary representation (finite automata) [CPS 89,BoC 88,GLZ 89]. Part of these attempts was AUTO[Ver 87b,LMV 87a], which originated as a (strong- and weak-) bisimulation congruence checker on terms of the MEIJE algebra [Bou 85].

Such tools can easily build large transition systems and check two of them for bisimulation, on a scale unmanageable by a human operator [Ver 86,Ver 88]. In addition the complexity of the growth of these systems can be cut down to some extent by using the congruence properties in order to reduce subterms first, before setting them in parallel. This is especially true for the weak congruence. Specific algorithms were studied, which are now fairly established. Such algorithms proceed along the following line: first devise a normal form of some kind by reducing each term individually, then perform the so-called *partitioning algorithm* to equate both terms to be proven bisimilar.

*ENSMP-CMA Sophia-Antipolis

†CERICS Sophia-Antipolis

This approach was pushed further in AUTO [SV 89], under the teachings of practice. Reductions to quotients of automata under various semantical criteria showed to be a promising way of analysis. A syntactical formalism for defining those reductions was then in order. We shall present here the state of the art in AUTO in this domain.

Along with the original definition of the MEIJE algebra in [Bou 85] came the notion of *abstract actions* and abstraction criteria, which are a powerful mechanism for defining levels of atomicity with different granularity, and actually move away from low-level details of basic concrete actions. It is a quite natural generalization of the ideas behind weak bisimulation, giving the user the possibility to decide himself on what is to be considered a relevant "experiment" performed on the system. Similar ideas may be found in [HeMi 85,Par 79]. Although we shall further elaborate on this later on, we can just say here that an *abstract action* is a set (usually regular) of concrete action sequences, to be thought of as "having the same meaning", as long as this sort of experiment is considered on the system.

One point of success is that in general abstracted transition systems are reduced drastically in size. They can be considered as characteristic of a partial vision of the system. This is to be contrasted with a temporal logic approach where statements are already imposed before checking, so that one does not get much out of an answer "no". When defining relevant abstract actions, the user usually provides (sets of) sequences with particular meaning which should appear, as well as others which should not. The presence of undesired actions in the quotient abstracted automaton indicates at once in which conditions they may take place, which is unvaluable information while "debugging" a system. Experiments were conducted in [Lec 89].

Use of AUTO quickly showed that editing of process terms was error-prone, due to misspelling of signal names and other deceptive mistakes that could obscure the communication abilities. This was the price to pay for writing terms in such a low-level formalism.Then a graphical representation of terms was wanted both as more flexible and more immediate than a textual one. Communications could be traced with lines joining ports, instead of using the lengthy notations of renaming and restriction operators, which induced most mistakes. Parallel operators could also be easily generalized to more than two processes for instance. Representation followed the lines of flowgraphs [Mil 79].

The graphical system was named AUTOGRAPH [RS 89]. It was not fully integrated with AUTO so that both can be used to a large extent independently. In particular, AUTOGRAPH's output may easily be turned to any process calculus manipulation system.

In fact the future of AUTOGRAPH resides not so much in graphical edition, as full languages tend to be far more complex than simple process algebras, but rather in graphical support of programs skeletons, including only their process structures, on which to visualize results of manipulation analysis from verification systems. This is nowadays our main direction of effort.

2 A short description of AUTOGRAPH

AUTOGRAPH is a graphical system, fully endowed with multi-window facilities. Functions are applied through a mouse button after selection of a menu in a menu bar. We shall not detail AUTOGRAPH general functionalities here, but rather focus on the nature of edited objects as well as functions specifically dedicated to visualizations of interesting results. Examples of typical AUTOGRAPHic drawings are pictured in the sequel. Let us just mention here that pictures may be printed on paper (and in reports!). AUTOGRAPH generates then a specific *Postscript* translation which makes drawings look much nicer than on the screen (and which separates object types more distinctly too).

AUTOGRAPH knows two main types of editable objects:

Networks

They represent terms and subterms, and are drawn as rectangular boxes. They usually

bear ports on their border, which are tied together with straight or broken lines to indicate communications. A communication is called internal if it does not pervade to the father box. Communications need not be named so that all matters of renamings and restrictions are left to the system. The only pertinent names that are required upon signals communications are the port names of innermost boxes, as well as communication names (eventually on the drawn lines) at the outermost level. These may not be guessed of course.

A box may contain a name in order for its content to be drawn in some other window (windows have titles giving names to their full content). Subterms may be shared, so that several boxes in the same window may bear the same name.

A box may also contain one automaton (at most), in which case the display of this automaton may not exceed the box boundaries.

In AUTOGRAPH one may retrieve information produced from AUTO: for example in an AUTOGRAPH **Net** one may highlight the set of states (distributed among all components) corresponding to a given state of a global system produced by AUTO. Then using this primary feature we could display either equivalence classes of such states, browsing back and forth through its scattered states; or behavior paths, by depicting the distributed state jumps, as well as the performed actions and synchronisations at ports at any level up the graphic process tree. This work is still under progress, but does not seem to make any problem.

Automata

They are represented by round-shaped vertices, which are joined by broken line edges. Both edges and vertices may be named, although it is mandatory for edges only.

An edge may actually be named several times, thereby representing several transitions at once. Identically named vertices refer to the same state, but at most one of them may have outgoing edges (it is then the state behavior "declaration", while the others are introduced to avoid loops in drawings). In fact there exist several such short-hand conventions in AUTOGRAPH allowing to simplify drawings. We shall not enter into details here.

Automata may be contained in boxes; alternatively there can be one residing directly inside the window.

Automata representing system components should be entered by the user, as the model of his problem. But one may also depict an automaton as resulting from analysis under AUTO. We call this "exploration". The automaton is not automatically positioned: instead, the initial state is given, and then one-step transitions of any explored state are progressively provided on demand. The reason for this choice was that automatic placement is often disappointing, while progressive unfolding of the states and transitions may lead to interesting considerations (much like simulations of systems).

3 A short description of AUTO

AUTO is a system consisting of a main toplevel loop, in which one may type *commands*. Commands may be of various sorts (including input/output to and from files). But most of them bind identifiers to results of *functions* applied to objects. Functions may be composed from a list of primary functions, which constitute the heart of AUTO. Other usual commands are those binding identifiers to syntactic objects. In this case one has to invoke the corresponding parser explicitly (*e.g.* `parse x = a:stop` is a command parsing a simple MEIJE term). In the former case one simply types `set y = function(...)`.

AUTO knows 6 main types: *(process) terms*, *signals lists* (for sorts of processes), *automata* (for internal representation of compiled systems), *partitions* (for internal representation of equivalence classes of states), *paths* (for sequences of behaviors), and finally *abstraction criteria*.

3.1 Reductions

Abstraction criteria, along with several other notions such as *contexts* [Lar 87], are the syntactical means for AUTO to characterize process behaviors so as to reduce them further. An abstract action is a set of sequences of actions, and in AUTO a regular such set. A criterion is a collection of specific abstract actions, and in AUTO a finite such set.

Abstract actions lead to state identifications, and thus to smaller quotient systems which may be analyzed more easily. This reduction only partially retains properties, but this is under full control of the user. In particular, when the union of all abstract actions does not add up to the full free monoid of possible concrete actions, then certain (sequences of) behaviors may go unnoticed. This amounts to a fairness assumption: such behaviors would not pertain to the abstract model. Think of infinite $\tau-$loops in the weak bisimulation case for instance.

Short-hand notations for functions are used when the criterion to be applied is simple and well-recognized. This is the case of course for weak bisimulation reduction, where we call *a-experiment* any sequence of (more concrete) actions in $\tau\star : a : \tau\star$. This criterion is generalized to the case where only some actions remain visible, while others are renamed to τ.

We can now present a first set of functions in AUTO, some based on the abstraction mechanisms and some on more classical reduction principles. They all share the property that they produce normal forms for automata, from terms, each along a given semantics. They use congruence properties wherever possible. Importantly, these functions may be composed. For details of application, see AUTO's Handbook [SV 89].

tta
: constructs the full global automaton corresponding to a term.

mini
: constructs a normal form automaton *w.r.t.* strong bisimulation.

obs
: constructs a normal form automaton *w.r.t.* weak bisimulation.

tau-simpl
: constructs a normal form automaton *w.r.t.* elimination of $\tau-$loops and single $\tau-$transitions.

trace
: constructs a normal form automaton *w.r.t.* trace language equivalence.

dterm
: constructs a normal form automaton *w.r.t.* determinisation.

exclusion
: constructs a normal form automaton *w.r.t.* elimination of transitions whose labels, as compound actions, contain atomic signals declared as incompatible in a parameter used by the function. Thus it trims away branches in the underlying graph.

tau-sature
: saturates an automaton using transitive closure of the transitions $\tau\star : a : \tau\star$ and $\tau\star$.

abstract
: abstracts an automaton by a given criterion, given as parameter. Unlike the previous functions, this one does not take benefit of congruence properties.

Other similar functions should progressively add up to this list, endowing the user with a consistent range of well-identified functions to create his own reduction notions. A mechanism of user-defined

functions is also envisaged, to give name to most popular reduction schemes. An example of desirable function is the *context-dependent* reduction, where one trims away behaviors of the process which are not part of the ones allowed by a given context. A context is a set of sequences of actions and thus amounts to an abstract action.

3.2 Comparisons

Of course resulting automata may be compared, through any of the two functions:

eq
: for checking strong bisimulation, and

obseq
: for checking weak bisimulation.

It was foreseen that the result of these functions should be a temporal logic formula in case of failure, but other recent efforts in this domain have proved it to be a difficult matter, especially due to the size of this synthesised formula. A progressive simultaneous exploration of the two terms seems a more promising method, even though it will be less automated.

Here again several further functions could be added, mainly the preorder comparisons, and a function providing the result of *testing* a process by a given observer (with *may/must* options).

3.3 Analysis

None of the preceding functions keeps unnecessary intermediate informations, for (space) efficiency reasons. For example $\tau-$behaviors do not remember which synchronizations produced them. Still, information is conveyed at two specific points, in the naming of states:

- The name of a state resulting from the expansion of a parallel system is the ordered list of states in components.
- The name of a state in a quotient automaton is picked from a representative of this class in the original automaton.

This information is enough for most cases, for it allows one to retrieve states and paths in original automata from reduced ones. So observations in our "partial view" systems may be uplifted to the most concrete automata. Now a further step would be to regain this information on the process itself. This amounts to retrieve which (sequences of) synchronisations led to $\tau-$behaviors, knowing each time the start and target states. It is under way.

Corresponding functions are:

structure
: provides the external naming of a state in a given automaton. Otherwise names are referred by integer internal row.

path
: provides a path in a given automaton leading from a state to another (or from the initial state). This function should be completed so as to allow an abstract action to indicate admissible behaviors for performed (concrete) actions along this path.

Of course the internal names of states as required by the **structure** function above should not be user-provided, but obtained by the system. To this end there are functions computing (sets of) states enjoying some properties:

dead
: returns the deadlock states of an automaton

diverge
: returns potentially diverging states of an automaton, those with *real* $\tau-$loops (or livelocks).

refusals
returns all states which may refuse to perform a signal outside a given list of signals.

A proper mixture of abstraction criteria and these functions may allow an analysis leading to a concrete result, as sketched in the example of section 4. We are not going to expand this type of functionality in AUTO, trying to spot every property of interest in the literature. Instead, collaboration with systems more directly dedicated to the definition and manipulation of such properties [Arn 89] seems more fruitful.

In order to realize this, while sticking to the main body of process calculi, we introduced a function performing the partitioning algorithm for (strong) bisimulation reduction from a given initial partition. It is called **refined-mini**. It may also help the user defining his own semantical reduction criteria at will.

Finally, it should be remarked that the original partition may itself be produced by another partitioning experiment, possibly with a specific abstraction formulation or otherwise. More generally, one may at any moment want to grasp and analyze which states are equivalent w.r.t. a given semantics. This is the purpose of the following AUTO functions:

strong-partition
returns (an internal representation of) the collection of equivalence classes in an automaton *w.r.t.* strong bisimulation.

weak-partition
same thing, with weak bisimulation.

crit-partition
same thing, with bisimulation parameterized by a given abstraction criterion.

row
provides the row of the class to which a given (concrete) state belongs.

class
provides the list of elements in a class, given its row.

As we mentioned before, both paths and equivalence classes of states can be displayed with AUTOGRAPH on a graphical version of process terms.

3.4 Managing the complexity

There is no miracle to what AUTO may do in this domain. Efficient data structures and algorithms may push the limit a little further, so that for the time being systems of 10^4 states and around 10^5 transitions may be dealt with in few minutes. For larger systems the problem actually comes from storage limits, more than time bounds. So the solutions advocated in AUTO consist in **never** building full global systems, but instead only reductions of them relying on congruence properties, further enhanced by the partial elimination of unvisible actions, or by abstraction. Another feature here is the division of usual functionalities into smaller-grain functions, allowing finer reduction strategies for the user. For example it was found that the usual weak reduction algorithm, which corresponds to **mini(tau-sature(tau-simpl(process)))** (assuming that **process** contains but one level of parallel nesting, so that we leave away congruence considerations), was in many "symmetrical" problems replaced with benefit by **mini(tau-sature(mini(tau-simpl(process))))**. This is because the transitive completion of transitions performed by **tau-sature** is actually in practice the most consuming of our algorithms, especially in space. So any reduction before this phase is welcome.

Still, observing the complexity growth is not easy. AUTO provides through a collection of flag options the tracing of various measures: time, sizes of subterms at parallel construction, maximal length of τ–sequences to name a few.