13

Automated Testing of Safety Requirements with the Support of a Deductive Database

Patrizia Asirelli, Antonia Bertolino, Stefania Gnesi
Istituto di Elaborazione della Informazione - CNR
Via S. Maria, 46, I-56126 Pisa

Abstract

We present an approach to the implementation of an automated test oracle. The oracle is built on a Deductive Database Management System, called Gedblog. The oracle is partial, in the sense that it checks the correctness of the test outputs with respect to a set of necessary conditions derived from the functional specifications, but does not know the correct output for each possible test input. We identify a set of requirements, called "safety requirements", that is deemed to be essential for the safe behaviour of the system under test. We express these safety requirements as a set of formulae by using the ACTL logic. We then incorporate these ACTL formulae in Gedblog. Assuming that the program has been previously tested apart, according to some test plan, and that the test outputs have been collected then, by means of the Gedblog mechanisms, we are able to automatically check the test outputs against the safety requirements. We illustrate the approach by means of a simple example.

Keywords

Automated Testing, Safety Requirements, Modal Logic, Deductive Databases

1 INTRODUCTION

Given the high-level specification of a safety-critical system, we can derive a set of programs, possibly at different levels of abstraction, that provide an implementation of such a system. Then, we need to verify that these implementations are consistent with the specification: in fact, this is a necessary requisite of safe and reliable software.

Testing is the dynamic verification of the consistency between the program under test and the specification. Indeed, testing has a fundamental role for both the achievement and the assessment of software reliability and safety (Laprie, 1992).

The testing activity essentially consists of checking the actual behaviour of the software

Work supported by CNR-Committee 12 - Project "Use of deductive databases to support software development" and by OLOS HCM Network - EC Contract No. CHRX-CT94-0577

S. Bologna et al. (eds.), *Achieving Quality in Software*

product against the specified, or *expected*, behaviour. For this purpose, the program is executed on a collection of suitably selected inputs, the "test cases".

The problem of the selection of a subset of test inputs from the (usually infinite) input domain is a crucial one and has received great attention in the past research. The selection may be done systematically, according to one of the several test data selection strategies available, (Beizer, 1990) and (Myers, 1979), or may consist of randomly drawing an appropriate sample from a given input distribution (Musa, 1987), (Thevenod-Fosse, 1991). Whichever approach is followed, after having executed the program on the selected test inputs, the equally critical problem remains of analysing the outputs obtained and of deciding for each test whether it is approved or rejected. Unfortunately, this problem has been neglected by researchers.

In fact, most part of the testing literature is based on the assumption that once the inputs have been selected and the program has been executed, then the task of determining whether or not the test results are correct with respect to the specification can always be done in a straighforward manner. A mechanism that routinely decides whether or not a test output is correct is generically called an *oracle*, and the assumption that an oracle is always available is the *oracle assumption*. However, it would be more realistic to speak rather of the "oracle problem". Indeed, the situation is not much changed since Weyuker (1982) recognized more than a decade ago, that confidence in the oracle assumption is not justified, neither in theory nor in practice. In most cases, an oracle is not available, e.g., because the correct output can't be known exactly in advance or because building such a mechanism would be impractical; thus, the tester is faced with the problem of devising a useful substitute.

An oracle consists of two components (Richardson, 1992): the *oracle information*, that is the specification of the correct behaviour for the program under test, and the *oracle procedure*, that is the process followed by the oracle to check the obtained test outputs against the oracle information.

The oracle information can be implemented into one (or more) independent program version(s) intended to accomplish the same specifications as the program to be tested. In this way, each program version provides the complete oracle information to the other version. A simple comparison program then implements the oracle procedure. This approach is however highly expensive and is used just for critical software.

More often, in practice, the test results are inspected manually, i.e., the oracle information is provided by an expert of the application. To help the human oracle, the program is generally executed on a simplified set of test inputs, for which the correct output is known or can be determined easily. In some cases, the oracle information that is available is not complete. For instance, the person who is examining the test outputs cannot know the correct output of the program under test exactly. He can usually say that a test has failed, when he sees an incorrect output, but clearly the risk that he accepts plausible, yet incorrect, tests always exists.

On the other hand, if it can be still admissible to pretend that a human being can examine every result when the execution of comparatively few tests is considered, in the more realistic case that many tests, even thousands or millions, are needed, it is clear that the oracle must be automated. To allow the realisation of an automated mechanism, the oracle information is reduced to a set of conditions. These are usually only necessary conditions, so that incorrect runs may go undetected.

When the oracle information is not complete, we speak of a *partial* oracle. For a partial

oracle, the information consists only of a set of correctness, or *plausibility*, conditions; if a test output does not satisfy these conditions the oracle rejects the test. But, the partial oracle does not know exactly the correct result: so a test that satifies the oracle conditions will be approved, but, even though plausible, the test result might actually be incorrect.

In other words, an automated, high-level oracle can be used to check whether the results of the testing satisfy at least the "type" (if not the"value") of the expected results, where the "type" is derived from the (formal) specifications of the system. To better explain the idea, let us provide the following, trivial, example. Suppose we have to design, and test, a program that implements a multiplier for relative integers. Then, for instance, we could build an automated oracle that only checks whether the sign of the multiplication is correct. That is, if the two factors have the same sign, the oracle checks that the result is positive, or, otherwise, that it is negative. In the special case that one of the factors is zero, the oracle checks that the result is zero. This oracle however does not provide the exact numerical result for the multiplication.

Several approaches have been recently suggested to develop oracles from specifications expressed in formal languages, e.g., Richardson (1992), Gorlick (1990) and Bernot (1991). Among formal languages, logic is a good candidate for expressing the high-level specification of systems, since it permits to describe system properties. Different types of logics have been proposed for this purpose. In particular, modal and temporal logics, due to their ability to deal with notions such as necessity, possibility, eventuality, etc., have been recognised as a suitable formalism for specifying properties of reactive systems (Manna, 1989). Among them, we recall the action based version of CTL (Emerson, 1986), ACTL (De Nicola, 1990). This logic is suitable to express properties of safety-critical systems defined by the occurrence of actions over time. Moreover, a set of ACTL formulae can be used to express those requirements that a safety-critical system must necessarily satisfy. Specification-based oracles provide different degrees of automation, both for the derivation of the oracle information and for the implementation of the oracle procedure. The cost of such oracles increases with the level of detail of the information, which here consists of formally specified conditions. In our approach, the oracle information consists of a set of plausible conditions, which are expressed using the ACTL formalism (see section 2). The oracle procedure is realised by implementing the ACTL formalism using the logic database and integrity checking features of the Gedblog system (Asirelli, 1994) (see section 3). Gedblog is based on a logic language extended with the capabilities of handling graphical and non graphical information, in a uniform way, and the possibility of defining and verifying integrity constraints (Asirelli, 1985). Our automated oracle consists of an environment built on Gedblog according to a formal high-level specification in ACTL of the safety requirements for the system under test. More precisely, from the functional specification of the system we derive in ACTL the safety requirements. These safety requirements are then incorporated in Gedblog and constitute the basis from which the oracle verdict is derived. We suppose that an implementation of the given safety-critical system is tested apart accordingly to some test plan, anyhow derived, and that the test outputs are collected. Gedblog then, allows us to automatically check these test outputs against the ACTL safety requirements.

Table 1 ACTL operators

Action formulae		
$\chi ::=$	$true$	"any action"
	$false$	"no action"
	α	"α action"
	$\neg\chi$	"not χ"
	$\chi \mid \chi'$	"χ or χ'"
State formulae		
$\phi ::=$	T	"any behaviour"
	F	"no behaviour"
	$\sim \phi$	"not ϕ"
	$\phi\&\phi'$	"ϕ and ϕ'"
	$E\gamma$	"there exists a path in which γ"
	$A\gamma$	"for all paths γ"
Path formulae		
$\gamma ::=$	$[\phi\{\chi\}U\{\chi'\}\phi']$	"ϕ is true for the states of the path until a state that satisfies ϕ' is reached by executing an action satisfying χ'. Before it, only actions satisfying χ or τ can be executed."
	$X\{\tau\}\phi$	"the next state of the path satisfies ϕ and is reached by executing a τ action"
	$X\{\chi\}\phi$	"the next state of the path satisfies ϕ and is reached by executing an action satisfying χ"

2 THE ACTL LOGIC

Process algebras and their semantic models, i.e., Labelled Transition Systems (or, state automata), Milner (1989), are generally recognized as a convenient tool for describing sequential or concurrent safety-critical systems at different levels of abstraction. They rely on a small set of basic operators, which correspond to primitive notions of concurrent systems, and on one or more notions of behavioural equivalence or preorder. The operators are used to build complex systems from more elementary ones. The behavioural equivalences are used to study the relationships between descriptions of the same system at different levels of abstractions (e.g., specification and implementation). The concept of a Labelled Transition System and of computation paths over it is formally defined below.

A *labelled transition system* (or simply *transition system*) TS is a quadruple (S, T, D, s_0), where S is a set of states, T is a set of transition labels, $s_0 \in S$ is the initial state, and $D \subseteq S \times T \times S$. A transition system is finite if D is finite. An element of D is denoted by $s \xrightarrow{\mu} s'$.

A *finite computation* of a transition system is a sequence $\mu_1\mu_2..\mu_n$ of labels such that $s_0 \xrightarrow{\mu_1} .. \xrightarrow{\mu_n} s_n$.

ACTL is a temporal logic of state formulae (denoted by ϕ), in which a path quantifier prefixes an arbitrary path formula (denoted by γ) whose models are Labelled Transition Systems. Also, it includes the logic for the definition of action formulae (denoted by χ). The ACTL operators and their informal semantics over TS's are reported in Table 1, while the formal semantics is described in (De Nicola, 1990).

Several logic operators can be defined starting by the basic ones. Let χ, χ' range over action formulae, E and A be path quantifiers, X and U be the "next" and "until" operators, respectively. We will write:

- $EF\phi$ for $E[true\{true\}U\phi]$ and $AF\phi$ for $A[true\{true\}U\phi]$:

these are called the *eventually* operators;

- $EG\phi$ for $\sim AF \sim \phi$ and $AG\phi$ for $\sim EF \sim \phi$:

these are called the *always* operators.

The ACTL logic allows to simply express *safety* and *liveness* properties in terms of the actions a system can perform. Safety properties claim that nothing bad can happen; liveness properties claim that something good eventually happens (Manna, 1989).

When an ACTL formula is given and a system is described by means of a TS, it is possible to check if the TS is a model for such a formula. In this case we say that the system satisfies the property the formula expresses. On the other hand, it may be also interesting to perform to opposite step: given a formula that represents a system requirement, to look for a model that satisfies such a formula. In (Asirelli, 1995), we have presented a method that, for each given ACTL formula, derives a particular TS that satisfies such formula.

3 THE GEDBLOG SYSTEM

The Gedblog system, (Asirelli, 1994), (Aquilino, 1994), is a uniform environment which supports the fast prototyping of applications that can take benefit from a declarative specification style. For instance, applications which demand for:

- a support for knowledge management (according to the given data-model this knowledge is based on);
- a support for the graphic representation of knowledge;
- a support for the interactions, to make the knowledge manageable at the graphic level.

Gedblog is based on logic databases theory. It is a deductive (logic) database, that can deal with basic knowledge management functionalities (storing, retrieving, quering), and besides it is enriched with several additional features: i) Integrity Constraints and Checks, to define the data model entities must fit in; ii) Transactions, to enter the operational framework; iii) Input/Output graphic model (declarative, based on prototypes), to define graphics and interactions with graphic objects.

Gedblog can manage logical theories that consist of different kinds of clauses:

- *Facts* - "unit" clauses, considered to be the Extensional component of the DB (EDB);
- *Rules* - considered to be the Intensional component of the DB (IDB);
- *Integrity Constraints* (ICs) - their syntax is: $A >> B_1, ..., B_s$. Informally speaking their meaning is that: each time A is true, then B_1 and ... and B_s must be true. ICs are managed following the method of the *modified program* (Asirelli, 1985) and they automatically inhibit the deduction of inconsistent information;
- *Checks* - their syntax is: $A_1, ..., A_m ==> B_1, ..., B_n$. These formulae are checked only on user request and do not modify the knowledge expressed by the theory, as integrity constraints do; their meaning is analogous to ICs.
- *Transactions*, in the form T := Precondition # $B_1, ..., B_n$# Postcondition. The goals in the body can modify the extensional component of the theory (facts). Modifications are recorded only if Postcondition succeeds.

By means of the system-defined predicate *theory*, it is possible to perform inclusion among theories. In this way, given a *starting* theory Th, the associated Gedblog theory can be defined as the set-theoretic *union* of all the theories in the inclusion tree rooted in Th.

Gedblog includes a graphic specification language, integrating the features of Motif and X11 in the Gedblog theories.

4 DESCRIPTION OF THE APPROACH THROUGH A SIMPLE EXAMPLE

In this section, we shall illustrate our approach to derive an oracle from an ACTL specification and to implement it within the Gedblog system. To help exposition, we shall describe our approach while walking-through a simple example.

4.1 The *search* program

The example program we shall refer to is a simple C program, *search*. It is the demonstration program given with the SMARTS tool (Sostware Test Works, 1991), which we used to automatically run the testing session (see 4.2).

Search accepts two arguments: the name of a textfile and a string (up to 20 characters long) and searches the textfile for the string. Then:

- if the program finds the string in the textfile, it returns a message saying "Match found on line *i*";
- if the program does not find the string, it returns a message saying "No match";
- if the program was not launched with exactly two arguments, the program returns either of the two messages: "Not enough command line arguments" or "Too many command line arguments", depending on which is the appropriate case;
- if a textfile with the given name does not exist, the program returns the message "File does not exist".

4.2 The test session

In our approach for the derivation of a partial oracle, we are not concerned with the strategy followed to select the test cases. We assume that the program has been tested according to a certain test plan, and that we have the test inputs and the produced outputs available. Thus, for this specific example, a set of about one hundred test inputs was generated randomly using an automatic test generator tool, TDGEN (Software Test Works, 1991). Then, we supplied these inputs to a test driver, the SMARTS tool (Software Test Works, 1991), which automatically executed the *search* program and registered the test outputs on a file. In order to verify the functioning of the Gedblog oracle, we introduced a bug in the program: we have commented out the source lines that verify whether the arguments provided are more than two and output the error message "Too many command line arguments".

4.3 Safety requirements for the *search* program

We now want to derive a formal description of the expected behaviour for the program *search* and use this to derive our automated oracle. More precisely, we want to excerpt from the informal requirements given in 4.1 a set of conditions that we believe essential for the safe behaviour of the program, i.e., those that we deem are the *safety requirements* for this program. This phase is the most critical, because it is the phase in which the contents of the oracle information is decided. Obviously, the decision relies on the tester's sensibility. Different users might identify different conditions as their own safety requirements. For instance, in our example, we decided that the oracle must necessarily detect errors in the input phase. That is, the oracle will reject those tests that will not output an error message "Not enough command line arguments" or "Too many command line arguments" when less or more, respectively, than exactly two input parameters are passed to the *search* program.

We now describe more explicitly how we derived the safety requirements. We have identified in the informal requirements some key actions. They are: the launch of the search operation; the input of the file name and the input of the string. For the sake of simplicity, we have identified each of them with a keyword: *check*, *read_file* and *read_string*, respectively. Moreover, we have identified the possible output messages of the system and again we have assigned to them a set of overlined keywords: $\overline{error1}$ (for "not enough command line arguments"), $\overline{error2}$ (for "too many command line arguments") , $\overline{file\_not\_found}$, $\overline{match}$, $\overline{nomatch}$.

Hence, we have expressed our safety requirements as follows:

- Only a *check* operation is expected after a sequence of a *read_file* and a *read_string* (or viceversa). Afterwards, the system may return any * of $\overline{file\_not\_found}$ or $\overline{match}$ or $\overline{nomatch}$.
- When a *check* operation precedes a *read_file* or a *read_string* operation then the system returns $\overline{error1}$.
- If after a sequence of a *read_file* and a *read_string* (or viceversa) another *read_file* or a *read_string* is performed then the system returns $\overline{error2}$.

From these requirements expressed by natural language sentences we derived the following set of ACTL formulae.

- $AG([read\_file][read\_string][check] < \overline{match}|\overline{nomatch}|\overline{file\_not\_found} > true)$
- $AG([read\_string][read\_file][check] < \overline{match}|\overline{nomatch}|\overline{file\_not\_found} > true)$
- $[check] < \overline{error1} > true$
- $AG([check][read\_file|read\_string] < \overline{error1} > true)$
- $AG([read\_file][read\_string][read\_file|read\_string] < \overline{error2} > true)$
- $AG([read\_string][read\_file][read\_file|read\_string] < \overline{error2} > true)$

Finally, from this set of formulae we derived in figure 1 a finite TS (Asirelli, 1995) that represents the minimal TS satisfying the conjuction of the above formulae: this TS will be the basis on which the oracle is built. In 1, we have denoted with a double circle the final states.

*Note that our partial oracle is not able to judge whether any of these 3 outputs is correct.

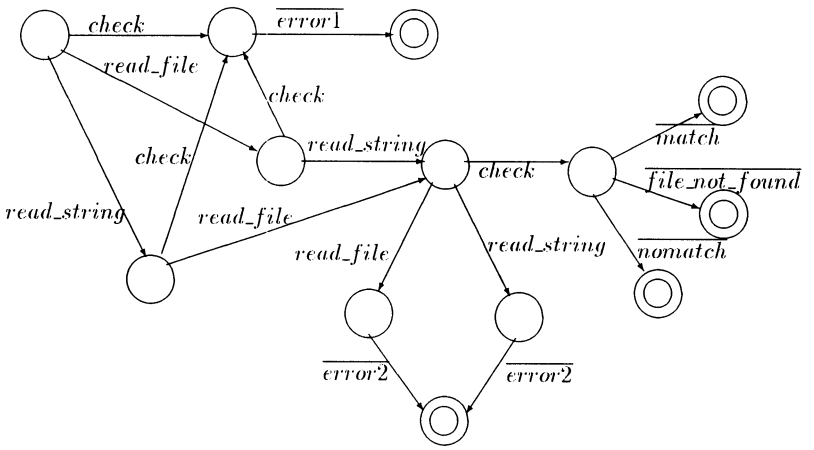

Figure 1 A TS for the Search Program

4.4 Implementation of the oracle

The oracle implementation in Gedblog consists of three theories:

- *oracle_info*: that contains the specification in clausal logic of the program under test. That is, the translation into clausal logic of the ACTL specification of the safety requirements for the *search* program;
- *test_outputs*: that contains the test outputs in form of unit clauses;
- *oracle_proc*: it is defined as the union of the two theories *oracle_ info* and *test_ outputs*, plus a set of definitions that can be considered of two kinds:

  - one is the specification of the syntactic transformation of the *test_outputs* so that they can be handled by the *oracle_ proc*, i.e., it specifies the mapping between the specification and the output of the test;

  - the other is the definition of two predicates: *correct_proof* and *wrong_proof* that realize the oracle itself; i.e., they identify which *test_output* (called "proof") is correct and which is wrong, respectively.

In particular:

- the *oracle_info* theory consists of three sets of facts:

  1) the set of states of the model of the program, e.g.,:

  $state(s0)$.

  $state(s1)$.

  $\vdots$

  2) the set of facts that defines which are the initial and the final states, e.g., :

  $initial\_state(s0)$.

  $final\_state(sfin1)$.

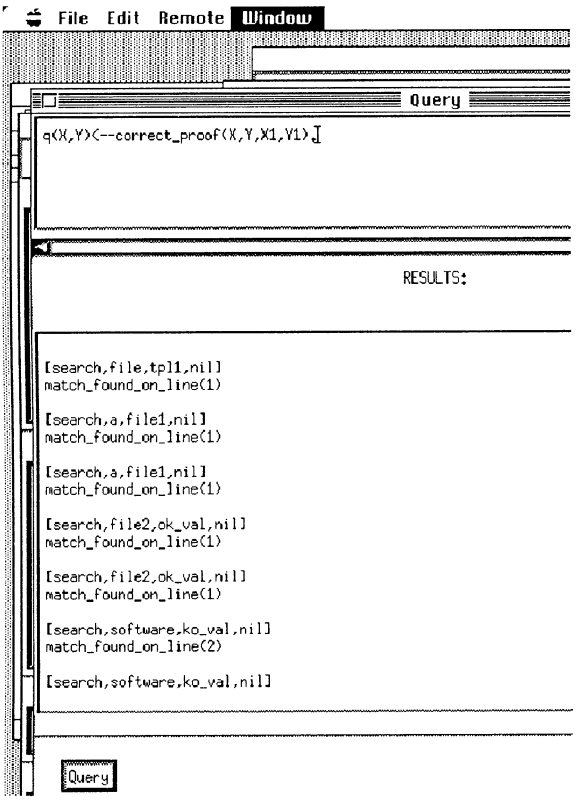

Figure 2 Correct test outputs

⋮

3) the set of transitions among the states, e.g.,: $transition(s0, s1, search)$.
$transition(s0, s2, read\_file)$.
$transition(s0, nil, nil)$.

⋮

The following are the rules to define a correct behaviour of a process as a possible path in the transitions net:
$correct\_behave(X, Y, Out) \longleftarrow path(X, Sfin, Y, Out) \& final\_state(Sfin)$.
$path(Sin, Sout, [], Out) \longleftarrow transition(Sin, Sout, Out)$.
$path(Sin, Sout, [X, Y], Out) \longleftarrow transition(Sin, Sinter, X) \& path(Sinter, Sout, Y, Out)$.

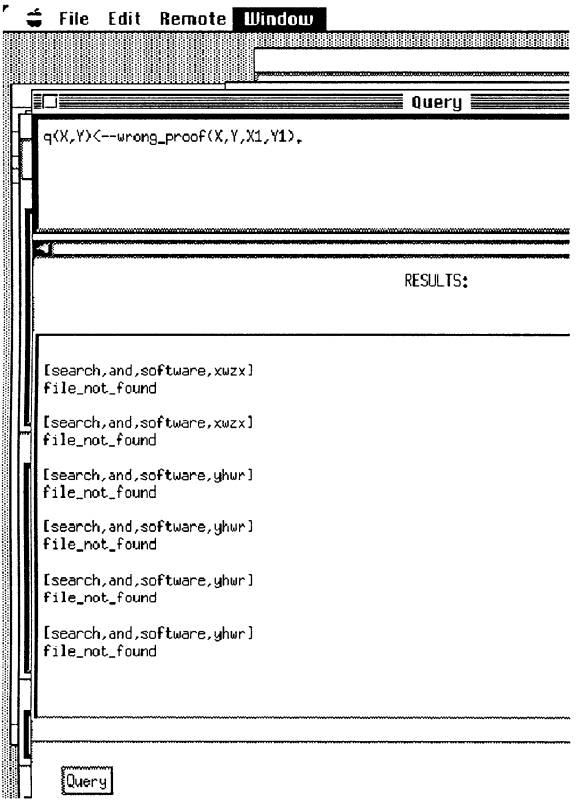

Figure 3 Wrong test outputs

Finally, there are the "checks", i.e., integrity constraints that can be periodically run to check that the model satisfies them.

$final\_state(X)\&transition(X,Y,Z) ==> Y = nil.$

$initial\_state(X)\&transition(Y,X,Z) ==> fail.$

$\vdots$

- The theory *test_outputs* appears as follows :

  $proof([search, nil, nil, nil], not\_enough\_command\_line\_arguments).$

  $proof([search, file1, abcd, file2], too\_many\_command\_line\_arguments).$

  $\vdots$

- Finally, the theory *oracle_proc* includes:

  - the two theories above described, by means of clauses:

  $theory('oracle\_info').$

$theory('test\_outputs')$.

- the definition of the mapping of the input/output parameters between the two theories, e.g.,:

$convert([search, nil, nil, Z], [search, []])$.

$convert([search, Y, nil, Z], [read\_file, [search, []]]) \longleftarrow not(Y = nil)$

$\vdots$

$final\_msg(error1, not\_enough\_command\_line\_arguments)$.

$final\_msg(file\_not\_found, file\_not\_found)$.

$final\_msg(error2, too\_many\_command\_line\_arguments)$.

$\vdots$

the definition of *correct_proof* and *wrong_proof* that implement the oracle:

$correct\_proof(X, Y) \longleftarrow proof(X, Y)\&$

$convert\_proof(X, Y, X1, Y1)\&correct\_behave(s0, X1, Y1)$.

$wrong\_proof(X, Y) \longleftarrow proof(X, Y)\&$

$convert\_proof(X, Y, X1, Y1)\&not(correct\_behave(s0, X1, Y1))$.

which states that the result of a test is correct if, given the syntactic, pre-defined, mapping of the input /output of the test (defined by the *convert_proof* predicate), there is a correct behaviour, in the specifications, starting from the state s0 (the initial state).

We have executed the oracle on the set of produced test outputs. As expected, the oracle could automatically indentify those test outputs that were correct and those not correct with respect to our TS (figure 1). The correct and wrong test outputs were found by running two different queries as shown in figure 2 and figure 3 respectively. In particular, 13 test outputs were rejected.

5 CONCLUSIONS

We have introduced an approach to implement an automated oracle for a subset of the specification requirements for a given system. The oracle has been implemented in Gedblog and is running on a Sun 4 station under Unix.

We have illustrated our approach by means of an example program. The example described is of course too simple to allow us to draw general conclusions.

However, our goal in this paper was to verify the feasibility of the approach. Our preliminary study seems to confirm the validity of our ideas. Now, experimentation on more significant case studies is necessary to investigate if the approach is viable in more realistic development environments.

Several enhancements on this preliminary implementation of an oracle are planned. The oracle implementation here described relies only on the support of the deductive database capabilities. We are now implementing the ACTL Graphic Formalism Modelling System, which will allow us to derive the TS from a set of ACTL formulae in a completely automated way. Hence, we should be able to exploit the full power of Gedblog in the near future. Moreover, we believe that such an approach can be used to deal with more complex applications, in particular concurrent programs, which can be easily expressed in the ACTL formalism.

6 REFERENCES

Aquilino, D., Asirelli, P. and Inverardi, P. (1994) Gedblog: a Multi-Theories Deductive Environment to Specify Graphical Interfaces. In *Proc. GULP-PRODE'94 Joint Conf. on Declarative Programming.* Peniscola, Spain.

Asirelli, P, De Santis, M. and Martelli, M. (1985) Integrity Constraints in Logic Databases. *Journal of Logic Programming*, **3**, 221-232.

Asirelli, P., Di Grande, D., Inverardi, P. and Nicodemi, F. (1994) Graphics by a Logic Database Management System. *Journal of Visual Languages and Computing*, **5**, 365-388.

Asirelli, P., Gnesi, S. and Magnani, S (1995) Syntesis of Temporal Logic Formulas: an Approach to Software Design. IEI-Internal Report., 1995.

Beizer, B. (1990) *Software Testing Techniques.* Second Edition. Van Nostrand Reinhold, New York.

Bernot, G., Gaudel, M.C. and B. Marre (1991) Software Testing based on Formal Specifications: a Theory and a Tool. *Software Engineering Journal*, **6**, 387-405.

De Nicola, R. and Vaandrager, F. W. (1990) Action versus State based Logics for Transition Systems. Proceedings Ecole de Printemps on Semantics of Concurrency. *Lecture Notes in Computer Science*,**469**, Springer-Verlag, 407-419.

Emerson, E. A. and Halpern, J. Y. (1986) Sometimes and Not Never Revisited: on Branching Time versus Linear Time Temporal Logic. *Journal of ACM*, **33 (1)**, 151-178.

Gorlick, M. M. , Kesselman, C. F., Marotta D. A. and Parker, D. S. (1990) Mockingbird: A Logic Methodology for Testing. *Journal of Logic Programming*, **8**, 95-119,

Laprie, J. C. (1992) Dependability: Basic Concepts and Terminology. *Dependable Computing and Fault-Tolerant Systems*, **5**, Springer-Verlag.

Manna, Z. and Pnueli, A. (1989) The Anchored Version of the Temporal Framework, in Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, *Lecture Notes in Computer Science*,**354**, Springer-Verlag, 201-284.

Milner, R. (1989) *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs.

Musa, J. D., Iannino, A. and Okumoto, K. (1987) *Software Reliability Measurement, Prediction, Application.* McGraw-Hill, New York.

Myers, G. J. (1979) *The Art of Software Testing* John Wiley & Sons, New York

Richardson, D. J., Aha, S. L. and O'Malley, T. O. (1992) Specification-based Test Oracles for Reactive Systems. *Proceedings of the 14th Int. Conference on Software Engineering*, 105-118.

Software TestWorks (1991) Test Regression Tools: CAPBAK, SMARTS, EXDIFF, TDGEN. SR Software Research, Inc. San Francisco.

Thevenod-Fosse, P. and Waeselynck, H. (1991) An Investigation of Statistical Software Testing *J. of Software Testing, Verification and Reliability*, **1(2)**, 5-25.

Weyuker, E. J. (1982) On Testing Non-testable Programs. *The Computer Journal*, **25(4)**, 465-460.

7 BIOGRAPHY

Patrizia Asirelli graduated in Computer Science at the University of Pisa, Italy. Since 1978 she has been a researcher of the Italian National Research Council (CNR), Pisa, Italy, in the Programming Languages and Operating Systems group for wich she is responsible since 1991. Her research interests include logic programming, deductive databases and software engineering.

Antonia Bertolino graduated cum laude in Electronic Engineering at the University of Pisa. Since 1986 she has been a researcher of the Italian National Research Council. Her research interests are in software engineering, particularly in testing theory and techniques and in testing automation. She is an Associate Editor of the Journal of Systems and Software.

Stefania Gnesi, graduated cum laude in Computer Science at the University of Pisa, Italy. Since 1984 she has been a researcher in the Programming Languages and Operating Systems group of the Italian National Research Council (CNR), Pisa, Italy. Her current research interests include methods and tools for the high-level specification and formal verification of concurrent systems, and applications of temporal logic.