Mechanical a-posteriori Verification of Results: A Case Study for a Safety Critical AI System

Roy Bartsch, Wolfgang Goerigk

Christian-Albrechts-Universität zu Kiel, Germany*

Abstract

This paper is to show how mechanical theorem proving can be used to verify even complex and heuristic programs like mission critical expert systems. Our approach is mechanical in two ways: The basic idea of runtime result verification is to validate each program result (at runtime) rather than to verify the program itself beforehand. Filtering each result by a sufficient algorithmic correctness predicate guarantees partial correctness of the modified program, if successful checking is proved to imply correctness of the result. We use a mechanical theorem prover to prove the latter fact.

Introduction

The effort of proving the correctness of AI systems seems often not justifiable. Heuristics and programming tricks are necessary to solve complex problems successfully. Mathematical induction then often fails because the algorithms to be verified get too complex and tricky. This applies in particular to many knowledge-based systems. On the other hand, we feel a strong need for verification in particular for safety- or mission-critical software.

In (Goerigk, Gaul, & Zimmermann 1998) we propose a *checker-based* approach to software verification which exploits the idea of *runtime result checking* (Blum, Luby, & Rubinfeld 1989) for verification. It is applicable if partial correctness of the application suffices. Partial correctness can be proved by a-posteriori runtime result verification. In the present paper we describe the application of this approach to the mechanical verification of an expert system, which is used as a *safety critical tool* for certifying railway control components. We use the ACL2 theorem prover (Kaufmann & Moore 1994).

The Relais Master (DTK 1996; Lange, Möller, & Neumann 1996) is an expert system for computing test-plans for relay assembly groups that control railway systems. It is used in the engineering phase of such devices in order to (more automatically) provide support for hardware-in-the-loop tests. The test-plans are generated from circuit descriptions. Later, they are to be automatically executed by a *test-roboter* in order to check the correct electrical behavior of the device.

Although digital circuits get more frequently used today, at least many German railway systems are still controlled by relay blocks. In order to guarantee a sufficiently high level of safety, these devices need regular maintenance, and in particular, a re-certification of every single physical device is required within regular time intervals.

Runtime Verification of Results - Getting the Idea

Let us assume a transformational program `y := π(x)` (or π for short) to be specified by pre- and post-conditions $P(x)$ and $Q(x, y)$ for inputs x and outputs y. Instead of proving the partial correctness (cf. (Hoare 1969; Jones 1990))

$$\{\, P(x) \,\}\ y := \pi\,(x)\ \{\, Q(x, y) \,\}$$

of the program π itself – *i.e.,* if P holds for x, and if π successfully terminates on x with result y, then Q holds for x and y – the idea is to modify the program and add a *checker predicate* which rejects incorrect results. That is to say, we construct a sufficient algorithmic formulation $check_Q$ of the post-condition Q and prove

$$check_Q\,(x, y) \;=\; true \implies Q\,(x, y)\,.$$

We call this property the *checker correctness*. It is now a simple exercise to prove that the modified program π', which additionally checks the post-condition and rejects any incorrect result, is partially correct with respect to P and Q:

$$\begin{array}{l}\{\, P(x) \,\}\\ \quad y := \pi\,(x)\ ;\ \textbf{if}\ \neg\ check_Q\,(x, y)\ \textbf{then abort fi}\\ \{\, Q(x, y) \,\}\end{array}$$

Of course we additionally have to guarantee, that running π does not corrupt $check_Q$. In our case, we run a separate checker program on ASCII representations of x and y, not modifying the expert system at all.

The Relais Master – A Safety Critical AI Tool

The Relais Master[1] automates the time-consuming and error prone manual test-plan construction. After scanning the

*Institut für Informatik und Praktische Mathematik, Olshausenstraße 40, D-24098 Kiel, Germany. Email: rba@is.informatik.uni-kiel.de , wg@informatik.uni-kiel.de

[1]The system has been developed by the German company DTK (Hamburg) in cooperation with the Laboratory for Artificial Intelligence (LKI) at the University of Hamburg, Germany.

circuit, the engineer transforms it into an internal graph representation, and the system then generates a set of measurements between terminal contacts of the device (the test-plan). Each measurement is augmented with information about the required *state* of relay contacts (*open* or *closed*). The test roboter will later pneumatically switch the relays. We have the following requirements for the generated test-plan:

- It should be *complete* for the circuit, *i.e.,* it should contain every test necessary to detect any combination of any number of defects in the relay group. (*soundness*)
- It should be "good" in the sense that it does not require too many relay switching operations, because the maintenance interval is mainly triggered by the estimated number of switches. (*quality*)

The first property is safety-relevant, whereas, fortunately, the second is not, although it is as important for practical usability and makes the use of AI techniques adequate for the problem. But for safety we only need soundness. In fact, this observation is crucial to our *checker-based* approach: Soundness can quite easily be checked for a given test-plan, whereas the test-plan generation is complicated and has to assure quality as well.

The input of the Relais Master is a circuit description which actually is manually constructed from a scanned circuit plan. Figure 1 shows a very small example and the Ascii

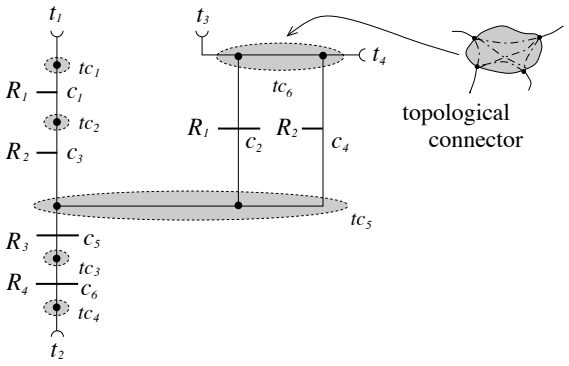

Figure 1: A sample circuit. Relais R_1 has two contacts, c_1 and c_2, and R_2 has contacts c_3 and c_4. If R_1 is not excited (released), contact c_1 is *closed* (*i.e.,* is to conduct), whereas c_2 is *open* (*i.e.,* is to isolate). Topological connectors are abstractions from wiring and soldering, for n ports, at least $n-1$ connections have to conduct.

representation of the internal object-oriented Relais Master data structure is a Lisp-s-expression of the form:

```
(((R1 released ((c1 closed) (c2 open)))
  (R2 released ((c3 closed) (c4 closed)))
  (R3 released ((c5 open)))
  (R4 released ((c6 open)))
  t1 t2 t3 t4)

 ((t1 c1) (t2 c6) (t3 c2 c4 t4) (t4 t3 c2 c4)
  (c1 t1 c3) (c3 c1 c5 c2 c4) (c5 c6 c3 c2 c4)
  (c6 c5 t2) (c2 t3 t4 c4 c3 c5)
  (c4 t3 t4 c2 c3 c5))

 ((con t3 t4 () (R1 R2 R3 R4))
  (con t3 t1 () (R1 R2 R3 R4))
  (iso t2 t1 (R3) (R1 R2 R4))
  (con t3 t2 (R3 R4) (R1 R2))
  (iso t3 t2 (R1 R3 R4) (R2))
  (iso t1 t2 (R1 R3 R4) (R2))
  (con t3 t2 (R1 R2 R3 R4) ())
  (iso t1 t2 (R2 R3 R4) (R1))
  (iso t2 t3 (R1 R4) (R2 R3))))
```

The first sub-list contains the contacts and terminals, the second is the graph structure of the circuit (every single node is followed by the list of direct neighbors), and the third list is the set T of measurements (conduction or isolation between two terminals, *e.g.* t_3 and t_4, with a certain state, *i.e.,* a set of excited and a set of released relays).

In the following, C will refer to the circuit description (the first two sublists), and T will refer to the third sublist, *i.e.,* the test-plan. C represents the input of the Relais Master, and T its output. C and T together form the input of the correctness predicate $check_Q$.

The Correctness Requirement

For simplicity, we just consider relay groups that contain terminals, connectors (wires, soldering points etc.) and relay contacts as *elements*. Diodes, resistors and capacitors are left out. So in our case, the generated test-plan only consists of *conduction* and *isolation* tests. Note however, that relay contacts are part of relays, so can not be switched independently, and the test-roboter can only measure between terminals (outside connectors).

Since later the certification is supposed to be mainly performed automatically, the generated test-plan and hence the Relais Master is *safety critical*. We need the following guarantee:

> *Every* defect of any relay contact, connector or terminal of the physical device will be detected by at least one measurement of the test-plan.

The combination of the generated tests must assure, if successful, that every single element is individually tested to work properly, *i.e.,* to conduct (terminals, connectors and relay contacts) and isolate (relay contacts) if supposed to. In that case, we call the test-plan T *complete for* the circuit C.

The crucial point is, that this is a *partial correctness* requirement for the Relais Master program (`R` for short): If it successfully generates a test-plan T for a given circuit C, then we want T to be *correct*. Let $P(C)$ characterize regular circuit descriptions, and let $Q(C,T)$ define T to be *complete for* C. Then we may formalize the correctness requirement for `R` by

$$\{\, P(C) \,\}\; T \texttt{ := R (} C \texttt{)} \;\{\, Q(C,T) \,\}\,.$$

Partial correctness requirements are typical for tools used in the construction of safety critical applications. We do not

want to prove that the tool never fails. But we want to guarantee that any given result is correct, *i.e.,* that the tool is partially correct w.r.t. the correctness requirement for its results.

The Checker and its Verification

We will now construct (and verify) a predicate $check_Q$ on circuits C and test-plans T which checks T to be complete for C, *i.e.,* which guarantees the postcondition $Q(C,T)$ to hold.

The checker is a Lisp program. Inputs are s-expressions representing C and T (as in the above example). C is a graph with nodes for every element, T is a set of either conduction or isolation tests between two terminals in a given state of the relay contacts.

A successful conduction test between t_1 and t_2 guarantees *one* conducting path (we do not know which), whereas a successful isolation test guarantees *every* path to isolate. In both cases, however, this might well be due to a defect. The checker proceeds in five steps:

1. The circuit representation C is transformed into a handier internal graph representation.
2. Each test in T is replaced by the set of all paths between t_1 and t_2 in C together with its type.
3. Every contact of each path in a test is augmented with the corresponding should-be-state (open or closed).
4. Assuming all tests to succeed, we get true logical propositions about physical conduction of (sets of) paths in the concrete device; simple logical transformations give us true propositions about individually tested single elements.
5. Finally, we check that indeed every circuit element is individually tested. If not, the checker aborts.

The Checker Program

The checker is written in the subset of applicative Common Lisp supported by the Boyer/Moore theorem prover ACL2 (Kaufmann & Moore 1994). ACL2 stands for "A Computational Logic for Applicative Common Lisp". It is a logic, a theorem prover and also an applicative programming language. As a logic, ACL2 is an essentially quantifier-free first order logic of total recursive functions. As a theorem prover, ACL2 is an industrial strength successor of the well-known Boyer/Moore theorem prover `Nqthm` (Boyer & Moore 1979).

Step 1. The first step is to transform the graph structure of the circuit (see the above example) into in handier form adding the topological connectors as nodes. We define a function `make-graph` such that, given a circuit c, `(make-graph` c`)` returns the new graph structure. We will later prove that each edge in c is an edge in `(make-graph` c`)` and vice versa, hence, that the two graph representations are (essentially) eqivalent.

Actually, the example representation above is already the result of this step. The original Relais Master generated representation of C and T is much larger and hard to read.

Step 2. The second step is to replace the terminal contacts t_i and t_j of each conduction of isolation measure m in the test-plan T by a set pl of paths between t_i and t_j. The function `find-paths` returns all paths (lists of nodes) in g between terminal t_i and t_j. We will later prove, that each path between t_i and t_j in g is a path in `(find-paths` t_i g t_j`)` and vice versa.

Step 3. With respect to a given measurement m in T, each relay contact has a corresponding *should-be-state*, *i.e.,* it should conduct (is *closed*) or isolate (is *open*) depending on the ground state of the relay, the ground state of the contact, and whether the relay is to be excited or released in the particular measurement. Each contact c_i in each path expression is augmented by its should-be-state, *i.e.,* is replaced by either c_i^{open} or c_i^{closed}.

In our example, this procedure finally returns the following augmented measurement path list (for simplicity, we omit the topological connector expressions, and we use c_i^{cl} and c_i^{op} for c_i^{closed} and c_i^{open}, respectively):

```
((con (t3 t4))
 (con (t3 c2^op c3^cl c1^cl t1) (t3 c4^cl c3^cl c1^cl t1))
 (iso (t2 c6^op c5^cl c3^cl c1^cl t1))
 (con (t3 c2^op c5^cl c6^cl t2) (t3 c4^cl c5^cl c6^cl t2))
 (iso (t3 c2^op c5^cl c6^cl t2) (t3 c4^op c5^cl c6^cl t2))
 (iso (t1 c1^op c3^cl c5^cl c6^cl t2))
 (con (t3 c2^cl c5^cl c6^cl t2) (t3 c4^op c5^cl c6^cl t2))
 (iso (t1 c1^cl c3^op c5^cl c6^cl t2))
 (iso (t2 c6^cl c5^op c2^cl t3) (t2 c6^cl c5^op c4^cl t3)))
```

This representation is computed in two steps. First, `(switch-contacts` m C`)` computes a list s of annotated contacts, and then, `(make-state-paths` T s`)` produces the annotated list pl of paths from the test-plan T.

Step 4. Assuming that isolation is the logical negation of conduction, we can read the above expression as a logical formula $\varphi_{C,T} = \varphi_{m_1} \wedge \ldots \wedge \varphi_{m_9}$ of logical path list formulas φ_{m_i}, which are disjunctions of conjunctions (path formulas φ_p) of atomic propositions for conduction measures, respectively their negations for isolation measures.

If a path p conducts (*i.e.,* $\varphi_p = true$), then every contact in that path conducts. If a path isolates (*i.e.,* $\neg\varphi_p = true$), then at least one contact isolates. Using these facts and some simple laws of the propositional calculus (such as *e.g.* deMorgan's law) we can transform $\varphi_{C,T}$ into a simpler formula φ, such that $\varphi \implies c_i^{closed}$ (respectively $\varphi \implies \neg c_i^{open}$) is easily checkable for any contact c_i. This transformation is computed by `(transform` $\varphi_{C,T}$`)`, and we will later prove that validity of $\varphi_{C,T}$ implies validity of `(transform` $\varphi_{C,T}$`)`.

Step 5. The final step is just to check that c_i^{closed} and $\neg c_i^{open}$ is a logical consequence of $\varphi_{C,T}$ for every contact c_i of the circuit (including terminals and single connections of topological connectors).

Verification of the Checker

In order to verify the Relais Master program according to our verified runtime result verification approach, it remains to prove $check_Q$ to be *sufficient* to guarantee Q, *i.e.,*

$$check_Q\,(C,T) = true \implies Q\,(C,T)\,.$$

We prove that the checker program returns $true$ only if the measures in T are sufficient to guarantee that every contact c_i in C is individually tested both to conduct, if it should be $closed$, and not to conduct, if it should be $open$. Terminals and single connections of topological connectors can be seen as special $closed$ contacts.

Note, that for safety it is again sufficient to prove *partial correctness* of the checker, *i.e.,* the checker might fail even if Q holds. We have to prove, that the test-plan T is complete for the circuit C, if the checker returns *true* for C and T.

We cannot go into much detail of the proof, but let us comment on the ACL2 theorems proved for each of the steps. Every of the five steps above is implemented by a set of ACL2 functions. The first three steps are syntactical transformations which are proved to be sound and complete.

Step 1. For the first step, we prove equivalence of the two graph representations; that is to say: If `(topology` c`)` extracts the graph part of the checker input (see the example above), and if `(make-graph` c`)` returns the transformed graph representation, then any edge between nodes x and y is an edge in the original graph `(topology` c`)` if and only if it is an edge in the transformed graph `(make-graph` c`)`:

Theorem 1 [equivalence of circuit and graph]

```
(thm
 (iff
  (circuit-edgep x y (topologie c))
  (graph-edgep x y (make-graph c))))
```

Step 2. The second step is to prove that the function `find-paths` is sound and complete, *i.e.,* (a) that every found path is a path in the graph g between terminal t_i and terminal t_j, and (b) that every path in g is found. For that, we prove the two theorems

Theorem 2 [soundness (a)]

```
(thm
 (implies (in-graphp ti g)
          (path-listp
            (find-paths ti g tj)
            g))))
```

Theorem 3 [completeness (b)]

```
(thm
 (implies
    (pathp p g)
    (member-equal p
       (find-paths
          (car p) g (car (last p))))))
```

Step 3. The first of the theorems for step 3 below is to prove (in any of eight cases) that the annotation of each contact with its corresponding *should-be-state* is correct. For each measure m and relay R we prove that any contact c is correctly annotated, *i.e.,* that `(switch-contacts` m C`)` contains c with the correct should-be-state. So for instance, if R is to be `excited` in m, if the ground state of R is `non-excited`, and if the ground state of R's contact c is `open`, then c is to be $closed$ for m. We prove that in this case c^{closed} is a member of `(switch-contacts` m C`)`:

Theorem 4 [one of eight theorems for switch-contacts]

```
(thm
 (implies (and (excitedp m r)
               (not (ground-excitedp r))
               (ground-openp c) ...)
          (member-equal c^closed
            (switch-contacts m C))))
```

We have omitted some additional technical well-formedness conditions, which are necessary to let the prover mechanically prove this theorem. The next theorem is to prove that `make-state-paths` produces a correctly annotated path list T_a, *i.e.,* for any contact c in T with state st from s we find c^{st} in T_a:

Theorem 5 [correctly annotated path list]

```
(thm
 (implies (and (member-member-equal c T)
               (member-equal c^st s) ...)
          (member-member-equal c^st
            (make-state-paths T s))))
```

In the final theorem for this step we prove that despite this change, the structure of T is preserved by `make-state-paths`, in particular, that T_a does not contain any contact not already contained in T:

Theorem 6 [structural equivalence]

```
(thm
 (struct-equivp p (make-state-paths p s)))
```

Step 4. We prove, that the logical interpretation of the annotated path lists as a $true$ formula φ is invariant under the logical transformation `transform`. Given a semantics function `sem` which defines the meaning of φ (in an assignement s which maps propositional variables to truth values), we prove that for any s,

Theorem 7 [correctness of the logical transformations]

```
(thm
 (implies (sem φ s) (sem (transform φ) s)))
```

holds. Since we assume the original formula $\varphi_{C,T}$ to be validated by the successful execution of the test-plan, *i.e.,* to be $true$, this direction is sufficient. We only depend on `(transform` φ`)` to be $true$ as well. Actually it is not hard also to prove semantical equivalence, though.

Step 5. We omit the fifth and final step, which is to prove that the checker $check_Q$ does not answer $true$ unless the list of checked contacts, connectors and terminals is equal to the list of all elements of the circuit, *i.e.,* unless these two lists are equal.

Note that all our (and many other) theorems have been mechanically proved by the ACL2 theorem prover. That is to say, the entire proof is mechanically checked (Bartsch 2000). The checker program is much simpler in many respects than the Relais Master expert system, much smaller, and it is a predicate written in the clean and abstract functional Lisp subset ACL2.

Related Work

We used a-posteriori runtime result verification to mechanically prove the trustworthiness of results of a knowledge-based tool of industrial relevance. The proofs can be found as part of (Bartsch 2000). Although not generally applicable, our approach demonstrates a very practical method useful in order to incorporate verification into the engineering of AI systems, in particular for *safety-critical tools*. Double checking the results is often surprisingly easy compared to the effort necessary to construct solutions. It is a well-known method to increase trust in results of computations. Our approach adopts this method for software verification. Note that, *e.g.* in embedded and real-time safety-critical systems engineering, checker programs are sometimes called *observers* used to ensure state consistency. Our checkers are result checkers for transformational programs, though.

The crucial part of our verified checker program can be seen as a problem specific tautology or model checking (E.M. Clarke & E.A. Emerson 1981; J.P. Queille & J. Sifakis 1981): Successful hardware-in-the-loop test according to the generated set T of measurements will guarantee a circuit dependent set of logical formulas to be true, and our checker program basically checks the particular formula

$$\bigwedge_{l=1}^{p} t_l \wedge \bigwedge_{j=1}^{k} tc_j \wedge \bigwedge_{i=1}^{n} (c_i^{closed} \wedge \neg\, c_i^{open})$$

(for all p terminals, k topological connectors and all n relay contacts) to be a logical consequence of that set of formulas. If so, the set T is *complete* for the given circuit, *i.e.,* the (finite) set T of *test cases* is sufficient for certification of the devices.

Runtime result verification (Goerigk, Gaul, & Zimmermann 1998; Pnueli & Traverso 1999) is strongly related to program checking (Blum, Luby, & Rubinfeld 1989), and our case study is an application in the field of testing safety critical devices. However, our main focus is on verification of the checker program that guarantees correctness of the test case generator (the Relais Master expert system), and for that we use classical inductive theorem proving. In contrast to *e.g.* (Jeron & Morel 1999), where test case generation is based on model checking, our approach does not rely on particular techniques used in the expert system itself. While program verification and model checking might well find their limits for complex (and large) applications, we strongly believe that (verified) runtime result verification scales up to real world applications in certain mission critical domains – not only for safety but also for security.

Acknowledgments We would like to thank our colleagues in the *Verifix* project on Correct Compilation, Gerhard Goos, Friedrich von Henke, Hans Langmaack, Wolf Zimmermann. Many fruitful discussions helped us to realize the impact of partial program correctness and runtime result and checker-based program verification. We thank Uwe Haferstroh, Sven Nordhoff and DTK for supporting the work on our case study. Many thanks to J Strother Moore (University of Texas at Austin) for the background and sometimes even backup necessary to successfully use the ACL2 logic and theorem prover. Finally, we thank the unknown referees for carefully reading an earlier draft of this paper.

References

Bartsch, R. 2000. Mechanisch verifizierte Programmprüfung für die Korrektheit von Prüfplänen in der Bahntechnik. Master's thesis, Institut für Informatik, CAU, Kiel.

Blum, M.; Luby, M.; and Rubinfeld, R. 1989. Program result checking against adaptive programs and in cryptographic settings. In *DIMACS Workshop on Distributed Computing and Crypthography*.

Boyer, R., and Moore, J. 1979. *A Computational Logic*. Academic Press Inc.

1996. RelaisMaster Documentation and Manuals. DTK - Gesellschaft für technische Kommunikation MBH, Palmaille 82, Hamburg, Germany.

E.M. Clarke, and E.A. Emerson. 1981. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen., ed., *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, 52–71. Yorktown Heights, New York: Springer-Verlag.

Goerigk, W.; Gaul, T.; and Zimmermann, W. 1998. Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science. Malente: Springer Verlag.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12:576–583.

Jeron, T., and Morel, P. 1999. Test generation derived from model-checking. *Lecture Notes in Computer Science* 1633:108–??

Jones, C. 1990. *Systematic Software Development Using VDM, 2nd ed*. New York, London: Prentice Hall.

J.P. Queille, and J. Sifakis. 1981. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*.

Kaufmann, M., and Moore, J. 1994. Design Goals of ACL2. Technical Report 101, Computational Logic, Inc.

Lange, H.; Möller, R.; and Neumann, B. 1996. Avoiding Combinatorial Explosion in Automatic Test Generation: Reasoning about Measurements is the Key. In *Proceedings of KI'96 Conference on Artificial Intelligence*. Dresden: Springer Verlag.

Pnueli, A., and Traverso, P., eds. 1999. *Proceedings of the FLoC'99 International Workshop on "Runtime Result Verification"*.