Efficient Regression Verification

*R. H. Hardin**
*R. P. Kurshan**
*K. L. McMillan***
*J. A. Reeds**
*N. J. A. Sloane**

January 4, 1996

ABSTRACT

A significant problem in commercial-size development projects is to ensure that the process of fixing one design problem does not introduce another. In the context of conventional testing this is checked through *regression testing*. If consecutive test suites check N properties, a failure in one may require retesting all the previous suites once a fix has been made. This results in $O(N^2)$ tests in all, to assure that no fix in fact breaks a previously good test. (Usually, one does not dare to defer retesting to the very end– thereby counting on nothing having been broken in the process– but retests all previous properties after each significant fix.)

When formal verification is used in place of conventional testing, the analog of regression testing can be done much more simply– in some cases effectively in constant time. The key to achieving this simplification is to replace re-verification with a highly reliable "hash" check on the model parse tree, reduced relative to the verified property. If the hash value is unchanged after a change in the model, it is essentially certain that the validity of the previously verified property has not changed.

*AT&T Bell Laboratories, Murray Hill, New Jersey 07974
**Cadence Berkeley Labs, Berkeley, California 94701-1144

1. Introduction

Testing hardware or software in the course of its development classically has been accomplished by running it through a number of *scenarios*. Each scenario focuses on a functional aspect of the design, with the purpose of checking whether the function is correctly implemented. Since a single scenario rarely is sufficient to test a given function, a number of related scenarios are tested. Together, these form a *test suite*. Thus each function of a design is tested through a test suite, comprising several scenarios.

Running a single test suite can be an involved process, requiring setup, running the scenarios, and evaluating the results. Usually a single scenario can be tested quickly, but a test suite may contain many scenarios, designed to check what may be a very large number of possible variations of the "basic" scenario.

Suppose a design has N roughly independent functions. In many commercial designs, $10 \leq N \leq 100$. For each function, a test suite is designed, and used to run a *test* of the given implementation. We may suppose that the tests are run consecutively. If at some point the implementation has passed the first $i-1$ tests but fails the i-th test, the source of the problem is located and the implementation is changed so as to fix the problem.

How does this change affect the outcome of the first $i-1$ tests? Commonly, a fix for one functionality has a chance of breaking another functionality. The usual way to guard against this is to use *regression testing*: to rerun the first $i-1$ tests to ensure they still pass. In reasonable designs, fixing one problem generally does not cause another, and if it does, the circle of fixes stabilizes quickly. On the other hand, almost nothing works the first time around. Thus $O(N^2)$ tests is a reasonably good estimate of the number of regression tests an N-function design will require.

When formal verification is used in place of testing, the methodological problem is the same. However, for *regression verification* (the analog of regression testing, when verification is used in place of testing), re-verifying can be extremely– even prohibitively– costly, as a single verification run may take many hours or even days. For $N = 50$, even if each verification run takes only 1 hour, regression verification could take in excess of 100 cpu days, which may lie outside the limit of feasibility for many projects, given the memory- and cpu-intensive nature of verification. (Many time-sharing computers become next to useless when more than a couple of large verification jobs are in progress.)

However, there is some good news: regression verification can be accomplished in a fashion which often is extremely efficient– even significantly more efficient than regression testing. The key lies with hashing relative to *localization reduction*.

Each of the N properties to be verified may (and often, must) be verified in a *reduced* model. Reduction is relative to the property being verified, so each reduced model reflects only those aspects of the total design relevant to the given property which is being checked. Different properties give rise to different reduced models. If it becomes necessary to change the implementation in order to fix a problem relative to one property, and this change in the implementation does not given rise to a change in the reduction relative to another property, then that second property need not be re-verified.

How can we know whether a change in the implementation will give rise to a change in a particular reduction? The original reduction could be saved and compared with a new reduction after the change. If the two reductions are identical, then the result of the verification runs would be identical, so there is no need to re-verify. If a change to fix a problem with one function does not affect another function, then the reduction relative to the second function most often will be unaffected. (Conceptually, the same principle could be applied to conventional testing. However, the result would be ambiguous and unreliable unless a semantically well-founded localization reduction were applied. That is, to be reliable, the testing would need to be conducted in the context of formal verification.)

Recomputing the reduction is fast, taking no more than 1–2 cpu minutes, even for quite large designs. Compared with the hours or days re-verification would require, this time is negligible. The reduction relative to some properties may change, and these could need to be re-verified. Nonetheless, if these are few in number, then the average complexity of regression verification would be sublinear or, effectively, "constant time" (since in practice, we really are interested only in the complexity for, say, $N \leq 100$).

However, the size of the reduced model which one is required to save for regression verification is fairly large: 1–5 megabytes is typical. As these models (saved as parse trees) have a lot of regularity, they yield good results with data compression: Ziv-Lempel compression typically reduces the size of the model by a factor of 10. Hence saving compressed reductions would require only 100–500 kbytes per reduction. Nonetheless, for $10 \leq N \leq 100$, this can add up to an unacceptable space requirement: 1–50 megabytes for regression verification. Even after verification is complete, these reduced models must be retained for maintenance purposes

(even after release, there are inevitable changes, which need to be re-verified).

An alternative to saving the reduced models is to use a reliable hash function on reduced models. If different reduced models hash to different values, with very high probability, then only the hash value need be saved, not the reduced model. During regression verification, the reduced model is recomputed and re-hashed. If the new hash value is the same as the old hash value, the validity of the property in the new model is deemed to hold, and re-verification is omitted.

This scheme has been implemented in the COSPAN verification system [HK90], with dramatic reductions in the cost of regression verification.

Moreover, once one has the hash value of a verified model, that value denotes a verified model in any context (assuming the semantics of the source language does not change). Since hash values are so short, one reasonably could keep a database of all the hash values of any model ever verified. In a later project, if a certain module is re-used, the fact that it already has been verified can be checked through its hash value, and it need not be verified again in its new setting. This is analogous to the practice with the HOL theorem prover, of storing a secure library of proved theorems. For this purpose, in order to achieve greater generality, string constants could be mapped to the ordinal (number) of their appearance in the source, and likewise, variable names could be mapped to generic names: $a[0], a[1], \ldots$, prior to hashing. An extra level of safety could be obtained by having the verification tool apply a digital "hallmark" to each successful verification. The hallmark would confirm that a purportedly verified module in fact was verified.

2. Localization Reduction

The COSPAN verification system [HK90] implements a *localization reduction* algorithm [Ku94, p. 170–172], which reduces a given model relative to a given property which is to be verified. The algorithm is iterative, and is based upon the monotonic nature of automata-theoretic verification, which seeks to test whether the formal language containment

$$\mathcal{L}(P) \subset \mathcal{L}(T) \tag{2.1}$$

holds. Here, P is an automaton model of a given design or implementation (possibly derived automatically from the VHDL or Verilog code used to implement the design) and T is an automaton which defines a property or "task" which P is intended to perform.

An automaton P' is a *reduction* of P if all the (linear-time) behaviors of P are behaviors of P', *i.e.*, if

$$\mathcal{L}(P) \subset \mathcal{L}(P') \tag{2.2}$$

and if P' is smaller (in some sense) than P. If also

$$\mathcal{L}(P') \subset \mathcal{L}(T) \tag{2.3}$$

then naturally (2.1) holds. The point of reduction is that P' is a simpler model than P, and therefore the test (2.3) is computationally simpler than (2.1). In localization reduction, P' is constructed algorithmically from P and T so that (2.2) holds by construction.

The computational complexity of localization reduction is guaranteed to be no worse than checking (2.1) directly, when that check is feasible, and typically is much smaller.

Localization reduction proceeds in 2 stages. In the first stage, P' is constructed as the "part" of P which is in the *dependence graph* of T. The dependence graph of T can be understood intuitively if P is represented in terms of assignments of system variables. Then P' comprises those variables upon which T ultimately depends. More specifically, say a variable x *depends* upon a variable y if the assignment of x depends upon y. Define a directed graph $\mathcal{G}$ whose vertex set is the set of system variables, and whose edges are the pairs (x, y) where y depends on x. Let $\mathcal{G}(T)$ be the subgraph of $\mathcal{G}$ comprising those variables admitting of a directed path to the variables of T. It is intuitive that the variables of $\mathcal{G}(T)$ are the variables upon which T ultimately depends, and that checking (2.1) can be reduced to checking (2.3), when P' comprises only those variables in $\mathcal{G}(T)$.

Formally, if P is defined in terms of component L-processes [Ku94], as a product

$$P = P_1 \otimes \cdots \otimes P_k$$

for a given Boolean algebra L of "system events", and T is an L-automaton, then the *localization reduction* P' of P is defined as

$$P' = P'_{i_1} \otimes \cdots \otimes P'_{i_n}$$

where $P'_{i_1}, \ldots, P'_{i_n}$ are constructed as follows.

Define the *envelope* $\epsilon(X)$ of an L-process or L-automaton X to be the smallest subalgebra $\epsilon(X) \subset L$ such that X in fact is an $\epsilon(X)$-process or $\epsilon(X)$-automaton, respectively. (In the context of system variables, $\epsilon(X)$ is the set of Boolean predicates (formulas) expressed in

terms of assignments of the variables of X.) Say an L-process P_i is *dependent* on a subalgebra $L' \subset L$ if the projection [Ku94] $\Pi_{L'} P_i$ is not a $\mathbb{B}$-process, *i.e.*, if some transition condition in the projected process is not identically 1 (*true*).

Assume that for each $i = 1, \ldots, k$, $\mathcal{L}(P_i) \neq \phi$. Define a directed graph $\mathcal{G}(T)$, the *dependency graph* of T, relative to $P_1, \ldots, P_k$, as follows. First, consider the graph $\mathcal{G}$ whose vertices $V(\mathcal{G}) = \{T, P_1, \ldots, P_k\}$ and whose edges $E(\mathcal{G})$ satisfy

$$(P_i, P_j) \in E(\mathcal{G}) \Leftrightarrow P_j \text{ is dependent on } \epsilon(P_i) .$$

Then $\mathcal{G}(T)$ is the subgraph of $\mathcal{G}$ whose vertices are those vertices of $\mathcal{G}$ which admit of a directed path to T, and whose edges are those edges of $\mathcal{G}$, both of whose endpoints are in $V(\mathcal{G}(T))$. For simplicity, we assume that the P_i's include no fairness constraints. That is, we assume that $\mathcal{L}(P_i)$ is the limit of a prefix-closed language [Ku94]. (Otherwise, the definition of dependency would need to incorporate the automaton acceptance conditions of the P_i's [Ku94].) Then, the following theorem is an easy consequence of the preceding definition.

Theorem: *If* $V(\mathcal{G}(T)) = \{P_{i_1}, \ldots, P_{i_n}\}$ *then*

$$\mathcal{L}(P_{i_1} \otimes \cdots \otimes P_{i_n}) \subset \mathcal{L}(T) \Rightarrow \mathcal{L}(P) \subset \mathcal{L}(T) .$$

This result can be significantly strengthened. Let $L' = \epsilon(P_{i_1} \otimes \cdots \otimes P_{i_n})$ and let P'_{ij} be the state-minimization of $\Pi_{L'} P_{ij}$. The computation of L' corresponds to *resizing* the system variables in the dependence graph of T, by reducing their ranges to the "ranges of relevance" to the dependence graph. For example, if a variable x takes its values from the range $1 \ldots 100$, but the only references to x in assignments of variables in the dependence graph, are switches on the conditions $x = 1$ and $x = 2$, then x is resized to a variable with the range $0, 1, 2$, where 0 means "not 1 or 2". (If a BDD-based language containment algorithm is used, it may be better to use $P'_{ij} = \Pi_{L'} P_{ij}$ instead, since the minimized process actually may have a larger BDD.)

Corollary:

$$\mathcal{L}(P'_{i_1} \otimes \cdots \otimes P'_{i_n}) \subset \mathcal{L}(T) \Rightarrow \mathcal{L}(P) \subset \mathcal{L}(T) .$$

In the worst case, $\{P'_{i_1}, \ldots, P'_{i_n}\} = \{P_1, \ldots, P_k\}$ and the check

$$\mathcal{L}(P'_{i_1} \otimes \cdots \otimes P'_{i_n}) \subset \mathcal{L}(T)$$

reduces to (2.1).

The second stage of the localization reduction algorithm is initiated only if the check (2.3) runs out of memory or allotted time. In this case, P' is successively redefined, according to the algorithm in [Ku94, p. 172]. In this algorithm, successively

$$P' = \bigotimes_{P \in \mathcal{P}_i} P$$

for $\{T\} \subset \mathcal{P}_1 \subset \mathcal{P}_2 \subset \cdots \subset V(\mathcal{G}(T))$. Either $\mathcal{P}_1 = \{T\}$ or $\mathcal{P}_1$ is a user-defined superset of $\{T\}$. If at some iteration, i,

$$\mathcal{L}(P') \subset \mathcal{L}(T)$$

fails, then the error track e– a counter-example to that containment– is used to augment $\mathcal{P}_i$, forming $\mathcal{P}_{i+1}$. Specifically, $\mathcal{P}_{i+1}$ is a smallest superset of $\mathcal{P}_i$ such that e cannot be simulated in

$$\bigotimes_{P \in \mathcal{P}_{i+1}} P \ .$$

If no such $\mathcal{P}_{i+1}$ exists, then in fact e can be extended to an error track in P [Ku94, Sec. 8.4], and thus (2.1) fails.

In terms of system variables, each successive definition of P' defines a section of the full model, containing the variables of T. Each variable at the edge of the section is allowed to take on every value in its range, in every combination with each other variable at the edge of the section (*i.e.*, the variables on the edge are "freed" and assigned nondeterministically, in their respective ranges). If this section has the property defined by T, then (since the variables are under-constrained, and automata define monotone properties) the full system necessarily has the property defined by T. On the other hand, the section also may introduce bogus errors, since in the full model the variables on the edge of the section are not so freely set. P' is expanded until either it produces a real error, or the property defined by T is verified. This may happen when P' is very much smaller than the full model.

Once a $\mathcal{P}_i$ is found relative to which (2.3) succeeds, it is saved in order to recompute P' for regression verification.

3. Hashing

Once (2.3) is verified for some localization P', we apply a hashing algorithm to the parse tree of P' and record the result. The hash value should occupy little space, and there should be a vanishingly small probability that two different versions of P' have the same hashing. Moreover, the hashing should be very fast. To accomplish this, we use a combination of two standard hashing techniques. The first uses a very high degree CRC check, based on the polynomial

$$(1+x)(1+x^{135}+x^{316}) = 1+x+x^{135}+x^{136}+x^{316}+x^{317}\ .$$

This incorporates the polynomial $1+x^{135}+x^{316}$, the highest-degree primitive trinomial of degree less than 320 [ZB69]. This limit was chosen so that this hashed value will fit into 80 hexadecimal digits. The factor $1+x$ ensures that no two parse trees that differ in at most three bits can be confused. (Of course it would be even better if we could guarantee that no two parse trees differing in say 10 bits could be confused, but it is not known how to achieve more than three with a CRC check of such high degree.) If the parse trees are random, the probability that two different versions of P' have the same check string is about $2^{-317} < 10^{-95}$.

To guard against the possibility of some unforeseen mechanism which systematically favors CRC hash collisions in the application at hand, and also to guard against other unforeseen factors, we also use as a second, independent, hash function the SHA-1 algorithm specified in [FIPS PUB 180-1]. This algorithm was designed to produce a "message digest" of 160 bits from a message or data file, in such a manner that it is computationally infeasible to find two different messages which produce the same message digest. Its design uses principles similar to those used by R. L. Rivest of MIT when designing the MD4 message digest algorithm [MD4], and is closely modeled after that algorithm. We used an off-the-shelf implementation of SHA-1 which correctly reproduces the test values listed in [FIPS PUB 180-1].

4. Performance

Both codes together produce only 120 hexadecimal digits, so the space required to store the encoded parse tree of each reduced model is negligible. The space required to store the version of $\mathcal{P}_i$ for which the verification succeeded, typically is .1–2 kbytes of ascii. Even for large examples, the hashing takes only a few seconds, and the entire localization reduction takes only a couple of minutes. The following table gives results from four "typical" examples,

run on a Silicon Graphics computer.

system model	number of system variables	compressed localized parse tree (kbytes)	hash time (seconds)	% of total localization reduction time
ISAFE	156	39	.5	50
DMA	801	156	2.7	36
SDLC	806	121	1.9	32
CDIC	2840	538	19.6	25

All these algorithm are implemented in COSPAN.

References

[FIPS PUB 180-1] U.S. Department of Commerce, Federal Information Processing Standards Publication 180-1, SECURE HASH STANDARD, 1995 April 17.

[HK90] Z. Har'El and R. P. Kurshan. Software for Analytical Development of Communications Protocol. *AT&T Tech. J.* **69** (1990) 45–59.

[Ku94] R. P. Kurshan, *Computer-aided Verification of Coordinating Processes — The Automata-Theoretic Approach*, Princeton Series in Computer Science, 1994.

[MD4] The MD4 Message Digest Algorithm Advances in Cryptology–CRYPTO '90 Proceedings, Springer-Verlag, 1991 pp. 303-311.

[ZB69] N. Zierler, J. Brillhart, On primitive trinomials (mod 2), II, *Inf. Control* **14** (1969) 566–569.