Computer-Aided Verification: How to Trust a Machine with Your Life

Gordon J. Pace

Department of Computer Science and AI,
University of Malta

Abstract. Mathematical predictive analysis of the behaviour of circuits and computer programs is a core problem in computer science. Research in formal verification and semantics of programming languages has been an active field for a number of decades, but it was only through techniques developed over these past twenty years that they have been scaled up to work on non-trivial case-studies. This report gives an overview of a number of computer-aided formal verification areas I have been working on over these past couple of years in such a way to be accessible to computer scientists in other disciplines. Brief mention is made of problems in these areas I am actively working on. It does not purport to be an overview of the whole field of computer-aided formal verification or a detailed technical account of my research.

1 Introduction and a Brief History

Today, software and hardware are used to control all sorts of devices — from washing machines and microwave ovens to braking systems in cars, medical devices and nuclear power stations. Engineering practice, as evolved over the centuries, dictates guidelines to follow when designing bridges and towers to guarantee that they do not fall. However, with software and digital hardware similar techniques fail to work for a number of reasons:

- A problem of scale: the techniques fail to scale up to the level of complexity found in software and hardware;
- Discrete vs continuous: the digital nature of computing systems precludes the effective use of algebraic laws in design which usually work well on problems with continuous variables;
- Discontinuity: thorough testing is nigh on impossible on discontinuous systems[1].

The more software and complex hardware is used in safety-critical systems, the more worrying this lack becomes.

Since the 1960s, techniques have been developed to enable mathematical reasoning about programs. Mathematical models of digital circuits had been around even before. Such models are obviously necessary to be able to prove properties of programs. However, the complexity of proofs using these models is formidable. The formal verification of a five-line algorithm can easily run into pages of dense mathematical formulae. The question 'Is the program correct?' is simply transformed into another: 'Is the proof correct?' This problem was addressed by writing computer proof checkers and proof editors. These tools allowed a proof script to be checked by a machine. Checking a proof for

[1] On a continuous domain problem, we know that if at a certain point the behaviour is correct, that it should also be so in the immediate neighbourhood of the point.

correctness turns out to be a computationally simple problem, and it is only the generation of the proof which is difficult. In 1971, Cook proved the boolean satisfiability problem to be *NP-Complete* [10]. Writing a program to calculate a proof in polynomial time is one way of showing that the satisfiability problem lies in P and thus that $P{=}NP$. This disillusioned those who hoped to add modules to theorem provers to generate proofs automatically[2].

In the late eighties and early nineties, researchers started looking at finite state systems whose states can be exhaustively enumerated. Typical examples of such systems are digital circuits, programs which do not use dynamic features, and systems which use a bound number of resources. The catch is that the number of states is exponential with respect to the length of its symbolic description. However, if we look at the *reachable* states in the system, it turns out that in certain cases we can manage to (i) enumerate all the reachable states, verifying the correctness of the system, or (ii) enumerate sufficiently many reachable states until a 'bad' scenario is unveiled. This led to a surge of interest in *model-checking* algorithms which verify a system automatically by systematically exploring its state space. In practice, applying such techniques blindly works only on toy examples.

Although it is still not known whether a polynomial complexity satisfiability algorithm exists, there do exist algorithms which are worst case exponential, but work reasonably efficiently on most typical cases. Such algorithms are usually symbolic in nature, in that they try to encode sets of different boolean variable assignments using a symbolic representation, thus allowing the description of large sets using limited space. These algorithms have been used in model-checking tools, pushing the limits of these tools to work on larger systems. This is usually called *symbolic model-checking,* in contrast with *enumerative model-checking* techniques already mentioned. Although symbolic model-checking may seem to be superior, it is largely a question of the problem domain as to which type of model-checking is more effective. In cases where large chunks of the state-space are not reachable, enumerative techniques may actually work better[3].

In the meantime, *abstract interpretation* techniques also started being used, where the user gives information on how to abstract the system under examination, thus reducing the size of the state space. Abstraction can also used to reduce infinite state systems to finite state ones. Abstract interpretation algorithms which try to calculate an appropriate abstraction algorithm have also been proposed, and such algorithms can enable the automatic verification of systems of up to 10^{1500} or more states[4].

Despite this seemingly huge number, this limit still falls well below the size of complex circuits currently being developed. However, the cost incurred by Intel when a bug was found on its Pentium chip, and the danger of a law-suit if, or rather when, a system failure directly leads to loss of life (or major financial losses) have driven hardware design companies to use formal verification techniques. In fact, a number of commercial hardware verification software packages have now been on the market for a number of years and most microprocessors and complex chips are partially verified before going on the market.

Software verification is much more complex than hardware verification. Strong reduction and abstraction techniques are required to be able to do anything automatically. However, limited success

[2] In most programming languages (which may use features such as unbounded integers, unbounded length strings and dynamic lists) the problem is even worse. Gödel's incompleteness theorem guarantees that no algorithm to decide the correctness of a program can exist.

[3] For example, in the case of concurrent programs and communication protocols, the state of the system would also contain the program counters along the different threads. Most program counter settings are impossible to reach, making large partitions of the state-space inaccessible. In these cases, enumerative techniques tend to be more effective.

[4] This figure is obviously dependant on the system in question. However, it is interesting to compare this to the typical limits of 10^7 states for enumerative model-checking and 10^{150} for symbolic model-checking using no abstraction.

has been demonstrated on restricted domain problems, such as memory leak analysis and device driver verification. A number of commercial tools which use static analysis and verification techniques to guide programmers have just started appearing on the market.

Despite its short history, model-checking boasts a vast literature. A good starting point to the field is Edmund Clarke's book [8] which gives an overview of model-checking at the time of its writing.

The rest of the report will outline a number of research areas on which I have been working over these past few years, and identifies the problems in which I am still working on at the moment.

2 Model Reduction Techniques

In enumerative model-checking, a major problem is that the composition of tractable systems may turn out to be intractable. Composition operators which may result in state-space explosion include asynchronous composition and partially synchronous composition, where some ports are synchronized, while the others are left to work independently. Unfortunately, these operators appear in various systems such as communication protocols, and thus cannot be avoided.

One solution proposed was to generate the individual elements of a composed system, and minimize the finite state machines as it composes them [11]. Systems are described in terms of a composition expression which states how the basic nodes of the system are composed together. A simple example of such an expression is:

$$\mathit{receiver} \mid\mid \mathit{hide\ sel\ in\ ((sender1} \mid\mid\mid \mathit{sender2)} \mid\mid_{\mathit{sel}} \mathit{arbitrator)}$$

where $||$ is synchronous composition, $|||$ is asynchronous composition, and $||_P$ composes together two systems synchronizing all communication over ports listed in P but communicating asynchronously an all other ports. *hide P in S* makes ports P inaccessible outside S. *sender1, sender2, arbitrator* and *receiver* are finite state machines. The algorithm would minimize these machines, compose them together and minimizes again, as it moves up the expression tree. In practice this algorithm significantly improves the performance of the finite state machine generation.

However, one problem remains. Quite regularly, the individual finite state machines allow for complex behaviour which will be simplified when composed with another machine in the system. In the example we have previously given, one could imagine complex senders which can emit data in any order. On the other hand, the receiver only accepts a number followed by a character. Combining *sender1* and *sender2* will result in an enormous automaton, but most of the states and transitions will be discarded when we later compose with *receiver*. In practice we may not be able to generate the intermediete automaton due to its size, even if the top level automaton may be quite small. This and other similar problems severely limit this approach.

One solution to this problem was proposed by Krimm et al [13] where they propose a solution which allows us to reduce an automaton with the knowledge that it will be later composed with another node (called the *interface*). In the example we gave, *sender1* and *sender2* can be reduced by using *receiver* as an interface. This technique has allowed the verification of substantially larger systems than was possible before.

Given an automaton with n_1 states, and an interface with n_2 states, the algorithm proposed in [13] guarantees that the result will have less than n_1 states, but requires space of the order $O(n_1 \times n_2)$ to generate the reduced automaton. If the receiver in our example was too large, reduction may be impossible. To circumvent the problem, either the user would have to come up with a simplified interface (the algorithm makes sure that the interface is correct, so there is no risk of giving a

wrong interface) or one can try to generate a smaller version of the interface automatically to use. The problem of generating the smaller version of the interface automatically, can be expressed in terms of the language recognised by automata:

> *Given an automaton M, we require a new automaton M' such that (i) M' does not have more states than M and (ii) the language generated by M' is larger than that generate by M: $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{M}')$.*

The problem has two trivial solutions: taking $M' = M$, or defining M' to be the automaton made up of a single state and which can perform any action in M. However, we would like a solution which allows an effective reduction when used as an interface. I have been looking into different ways of generating an effective automaton M'. The applications of this are numerous and a number of case studies which use interfaces for verification are available, and can be extended to the effectiveness of the alternative interface generation.

Another reduction technique, based on composition expressions has been identified in Pace et al [15]. It has been shown to be quite effective when reducing machines with internal actions which are not visible to the outside user. The interaction between and combination of these two reduction algorithms which use composition expressions has not yet been studied.

3 Model-Checking a Hardware Compiler

There are two essentially different ways of describing hardware. One way is using *structural* descriptions, where the designer indicates what components should be used and how they should be connected. Designing hardware at the structural level can be rather tedious and time consuming. Sometimes, one affords to exchange speed or size of a circuit for the ability to design a circuit by describing its behaviour at a higher level of abstraction which can then be automatically *compiled* down to structural hardware. This way of describing circuit is usually called a *synthesisable behavioural description*[5]. Behavioural descriptions are also often used to describe the specification of a circuit.

Claessen et al [5] have used a technique from the programming language community, called *embedded languages* [**?**], to present a framework to merge structural and behavioural hardware descriptions. An embedded description language is realised by means of a library in an already existing programming language, called the *host language*. This library provides the syntax and semantics of the embedded language by exporting function names and implementations.

The basic embedded language used is *Lava* [7, **?**]. Lava is a structural hardware description language embedded in the functional programming language Haskell [16]. From hardware descriptions in Lava, EDIF netlist descriptions can be automatically generated, for example to implement the described circuit on a Field Programmable Gate Array (FPGA).

If one looks at the compiler description in Haskell, the code is short and can be easily understood. Consider a regular expression compiler: Given a regular expression, the compiler will produce a circuit which once reset, will try to match the inputs to the regular expression.

```
compile :: RegularExpression -> Signal -> Signal
```

[5] These are to be distinguished from *behavioural descriptions* (as used in industrial HDLs such as Verilog and VHDL) which are used to describe the functionality of a circuit, but are do not necessarily have a hardware counterpart.

For example, the circuit compiled from the regular expression (using Haskell syntax) is `Wire a :+: Wire b`, will have one reset input, and one output wire. `Wire a` is interpreted as 'wire `a` is high for one time unit' (corresponding to the normal regular expression `a`), and the plus symbol is language union in regular expressions. The resulting circuit would thus output high if and only if it has just been reset and, either wire `a` or wire `b` (or both) carry a high signal. Compilation of language union would simply by expressed as:

```
compile (e :+: f) reset = or2 (match_e, match_f)
  where
    match_e = compile e reset
    match_f = compile f reset
```

Note that each language operator would simply add a finite amount of new circuit components. Now assume that we would like to prove a compiler invariant[6] *inv*. This can be proved using structural induction by showing that expressions such as:

```
(inv (reset1, match1) /\ inv (reset2, match2) /\
 plus_circuit((reset1,match1),(reset2,match2),(reset,match))
 ) |=> inv (reset,match)
```

This is actual Lava code, where `/\` is conjunction and `|=>` implication. The function `plus_circuit` simply ensures that the subcircuits are related according to the circuitry introduced for compiling `:+:`:

```
  plus_circuit((reset1,match1),(reset2,match2),(reset,match)) =
       (reset1 <==> reset)
    /\ (reset2 <==> reset)
    /\ (match  <==> or2(match1, match2)
```

All this can be elegantly expressed in terms of embedded languages. Note that the cases of the structural induction are of a finite nature and can thus be model-checked. This gives an excellent platform to experiment with and verify compiler invariants, and thus hardware compiler correctness.

In unpublished experiments I have carried out with Koen Claessen, we have managed to prove that a regular expression compiler satisfies standard regular expression axioms. I am currently trying to extend this work for compilers of more complex languages such as Esterel [4].

4 Hybrid Systems

In real life, digital systems interact with analogue ones. While programs, and digital circuits can be fully analysed in terms of boolean values, a large class of (classical) engineering systems can only be modelled as a set of differential equations. Typical systems are expressed as a set of equations, defining the rate of change (with respect to time) of the variables which describe the system. Quite frequently, however,one needs to look at the interaction between continuous and discrete

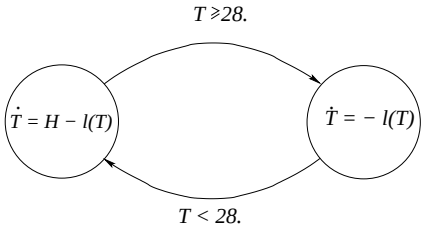
[6] In this context, a compiler invariant is a property relating the inputs and outputs of a circuit produced by the compiler whatever the compiled program.

systems. A typical example is a thermostat controlling a heating element. The equation defining the temperature can be defined in terms of a case equation:

$$\dot{T} = \begin{cases} -l(T) & \text{if } T \geq 28 \\ H - l(T) & \text{if } T < 28 \end{cases}$$

Where T is the temperature, and $\dot{T}$ is the rate of change of the temperature with respect to time. $l(T)$ is the rate of heat loss at temperature T, and H is the heating rate of the thermostat (which turns on when the temperature falls below 28). Note that the system can be in either of two different modes or states, one with the thermostat turned on and the other with the thermostat turned off.

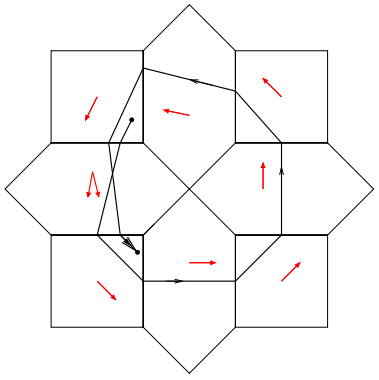
In such systems, discrete variables determine the system states which, in turn, determine the differential equations which describe the behaviour of the system variables. Such a system is called a *hybrid system.* Typically, hybrid systems are described using *timed automata* [1]. The thermostat example can be seen expressed as a (simplified) timed automaton in the following diagram:



As already noted, analysis of such systems is extremely difficult, and is in general undecidable. There is substantial research attempting to identify subproblems which are decidable. One such class of problems are Simple Planar Differentiable Inclusions (SPDIs) as identified by Asarin et al [2]. SPDIs are a subclass of hybrid systems with only two continuous variables. The best way to understand this subclass is to look at their visual representation.

An SPDI can be represented as a finite set of adjacent polygons on a plane. With each polygon we associate a differential inclusion (a pair of vectors which define the range of the rate of change of y with repect to x).

A typical example to illustrate SPDIs is the *swimmer problem* which depicts the motion of a swimmer in a whirpool:



The differential inclusions define the directions in which the swimmer may swim when in a particular region (polygon). When the swimmer reaches the edge of a region, her dynamics change to match the inclusion in the new region. The figure also includes a path which may be taken by the swimmer under the SPDI constraints.

The kind of question one would typically want to ask of such a system is a *reachability* question. If the swimmer were to start in a certain location, can she end up in a dangerous location? In [2], Asarin et al have proved the correctness of an algorithm to decide this question for SPDIs. The algorithm depends upon the identification of the concept of *meta-paths* each of which describes a family of paths within the SPDI (abstracting away the number of times loops are taken). Meta-paths enjoy a number of interesting properties:

1. Given any two polygon edges in the SPDI, we can find a finite number of meta-paths such that any path in the SPDI starting on the first and finishing on the second edge is an element of at least one of the meta-paths.
2. This set of meta-paths can be algorithmically generated.
3. Given a meta-path, we can decide whether there is a feasible path obeying the differential inclusions in the SPDI along that meta-path.

These three properties guarantee the decidability of the reachability problem. The algorithm was implemented in [3]. The main shortcoming of the algorithm is that it searches the graph in an inherently depth-first manner. Even though a short counter-example may exist, the algorithm might have to try very long traces before finding the short counter-example. This also means, that unless an exhaustive search is done, we cannot guarantee that the counter-example found is the shortest one possible (a desirable thing since it would then simplify debugging of the system).

In (the as yet unpublished) [14], we have identified a breadth-first search solution to the problem. The algorithm is still based on the idea of meta-paths, but information is stored in a combination of enumerative and symbolic techniques so as to be able to express the solution in terms of a more standard reachability algorithm:

```
reached := { start_edge };
repeat
   old_reached := reached;
   reached := reached ∪ one-step(reached);
   if (not disjoint(reached, finish_edge))
      then exit with counter-example;
until (reached ≈ old_reached);
report 'No counter-example';
```

The algorithm starts with the `start_edge` and adds other edges as they are reachable in one step (a path from one edge to another in the same region), until either nothing new is added (with certain provisos) or the `finish_edge` is reached giving a counter-example.

This algorithm is still to be implemented and compared to the depth-first version. Furthermore, the techniques used in our new algorithm (in particular the conditions required for the proof) indicate that a class of automata, more general than SPDIs, amenable to this technique can be identified. Work still needs to be done to identify this class and prove the generalised results.

5 Conclusions

In this report I have tried to give a brief outline of the history and relevance of formal verification techniques, to set the scene for some of my recent research and its current and future directions. At the risk of sounding like a collection of paper introductions and conclusions sections, not much detail is given in any of the research topics. Having only recently returned to the University of Malta, I hope that this report defines some research areas in which I am interested, possibly leading to fruitful collaboration with others in the department.

References

1. R. Alur and D.L. Dill, *A theory of timed automata,* Theoretical Computer Science 126:1(183–235), 1994.
2. E. Asarin, G. Schneider and S. Yovine, *On the Decidability of the Reachability Problem for Planar Differential Inclusions,* in Lecture Notes in Computer Science 2034, 2001.
3. E. Asarin, G. Pace, G. Schneider and S. Yovine, *SPeeDI — a Verification Tool for Polygonal Hybrid Systems,* in Computer-Aided Verification (CAV 2002), in Lecture Notes in Computer Science 2404, 2002.
4. G. Berry, *The Constructive Semantics of Pure Esterel,* Unfinished draft, available from `http://www.esterel.org`, 1999.
5. K. Claessen and G. Pace, *An Embedded Language Framework for Hardware Compilation,* in Designing Correct Circuits 2002 (DCC 2002), Grenoble, France, 2002.
6. K. Claessen, M. Sheeran and S. Singh, *The Design and Verification of a Sorter Core,* in Correct Hardware Design and Verification Methods 2001 (CHARME 2001), Lecture Notes in Computer Science 2144, 2001.
7. K. Claessen and M. Sheeran, *A Tutorial on Lava: A Hardware Description and Verification System,* Available from `http://www.cs.chalmers.se/~koen/Lava`, 2000.
8. E.M. Clarke, O. Grumberg and D.A. Peled, *Model Checking,* MIT Press, 2000.
9. E.M. Clarke, O. Grumberg and D.E. Long, *Model Checking and Abstraction,* ACM Transactions on Programming Languages and Systems (TOPLAS) 16:5(1512–1542), 1994.
10. S.A. Cook, *The Complexity of Theorem-Proving Procedures,* in Annual ACM Symposium on Theory of Computing, pages 151–158, New York, 1971.
11. S. Graf, B. Steffen and G. Lüttgen, *Compositional Minimisation of Finite State Systems Using Interface Specifications,* in Formal Aspects of Computing 8:5(607–616), 1996.
12. P. Hudak, *Building Domain-Specific Embedded Languages,* ACM Computing Surveys, 28:4, 1996.
13. J.-P. Krimm and L. Mounier, *Compositional State Space Generation from Lotos Programs,* in Tools and Algorithms for the Construction and Analysis of Systems 1997 (TACAS 1997), in Lecture Notes in Computer Science 1217, 1997.
14. G. Pace and G. Schneider, *Invariance Kernels for the Verification of Polygonal Differential Inclusions,* work in progress, 2003.
15. G. Pace, F. Lang and R. Mateescu, *Calculating τ-Confluence Compositionally,* in Computer-Aided Verification (CAV 2003), in Lecture Notes in Computer Science (volume to appear), 2003.
16. S. Peyton Jones, J. Hughes et al., *Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language,*, Available from `http://haskell.org`, 1999.