Designing Executable Abstractions

Gerard J. Holzmann
Bell Laboratories
600 Mountain Avenue 2C-521
Murray Hill, NJ 07974
gerard@research.bell-labs.com

1. ABSTRACT

It is well-known that in general the problem of deciding whether a program halts (or can deadlock) is undecidable. Model checkers, therefore, cannot be applied to arbitrary programs, but work with well-defined abstractions of programs. The feasibility of a verification often depends on the type of abstraction that is made. Abstraction is indeed the most powerful tool that the user of a model checking tool can apply, yet it is often perceived as a temporary inconvenience.

1.1 Keywords

Model checking, software verification, distributed systems

‘‘Seek simplicity -- and distrust it,’’
Alfred North Whitehead (1861-1947)

2. INTRODUCTION

In the days when C was still just letter in the alphabet, and C++ a typo, it was already well-established that it would be folly to search for a computer program that could decide mechanically if some arbitrary other computer program had a given property. Turing’s formal proof [6] for the unsolvability of the ‘halting problem’ was illustrated in 1965 by Strachey with the following simple construction. Suppose we had a program ***mc(P)*** that could decide in bounded time if some other program ***P*** would terminate; we could then define a program ***Q***, that uses input ***P*** as follows:

```
Q(P):
begin
      if mc(P) == terminates
      then
            L: goto L
      else
            terminate
      fi
end
```

Program ***Q(P)*** terminates if ***P*** does not terminate, and vice versa, and all is well until we decide to apply ***Q*** to itself and ask if ***Q(Q)*** terminates.

Strachey’s construction is eerily similar to Russell’s famous *class paradox* [3], which makes it somewhat dubious to use as a basis for a logical argument. In fairness, this construction does not really prove the unsolvability of the halting problem, but it does prove that it is in general impossible to write a program that can establish arbitrary properties of ***itself***. This, of course, does not mean that no programs could be written that can establish *specific properties* of themselves. A word-count program, for instance, can trivially determine its own length, and a C-compiler can confirm its own syntactic correctness, and even recompile itself.

So, given this, could we in principle also write a computer program that can specifically establish its own logical correctness? It would, for instance, be quite attractive to have a model checker that could be applied to itself. The answer is, of course, negative. (Note, for instance, that such a program could report its own correctness erroneously...)

So what makes us think that we can write a model checker that can establish the logical correctness of ***other*** programs? The answer is that a model checker does not work with arbitrary programs, but with well-defined *abstractions* of programs. For the model checker Spin [4] these abstractions are also ***executable***, to provide support for rapid prototyping, but in general this need not be the case of course. Through the design of its specification language `Promela` (a Process Meta Language), Spin enforces two basic requirements on the abstractions it can accept as input. The models must:

- have *countable many* reachable states, and they must be
- *closed to their environment* (i.e., fully specified).

The reason for the first requirement will be clear. The second requirement says that the behavior of the model may not depend on any hidden assumptions or components, i.e., input sources must be part of the model, at least in abstract form.

Neither property is guaranteed to hold for an arbitrary program, but it is guaranteed to hold for any model that can be specified in `Promela`. A program that reads input from a file-descriptor or an io-stream, for instance, is not closed to its environment. Similarly, a program that can obtain *a priori* unlimited amounts of memory has an uncountable number of potentially reachable states, and is unanalyzable with a model checker unless an abstraction is first made.

The need for abstraction, or ***modeling***, is by practitioners often seen as a hurdle in the verification process, instead of as the model checker's most powerful feature.

Choosing the right level of abstraction can mean the difference between obtaining a tractable model with provable properties, and an intractable model that is only amenable to simulation and manual reasoning. Sometimes it also means that we have to make a choice between proving a (series of) simple properties of a complex model (corresponding to a low-level abstraction), and proving a complex property of a simpler model (corresponding to a higher-level abstraction of the application).

The importance of abstraction also places demands on the design of the specification language for a model checking tool. If the input language is too detailed, it discourages the user from making abstractions, which can initially appear to simplify the task but in the end will obstruct the verification process. Most model checkers therefore are careful in choosing what features will be supported. In the model checker Spin, for instance, a number of desirable language features have intentionally been left out of the specification language, to facilitate the construction of tractable models. Among them are:

- memory management
- floating point data types
- differential equations, etc., etc.

Some systems are still more restrictive, and exclude also:

- local and global variables, message parameters
- dynamic process creation
- asynchronous message passing

Other systems are more permissive, and, at the price of increased complexity, include support for:

- real-time (possibly drifting) clocks
- probabilities on transitions
- unrestricted function calls

All other things being equal, we should expect the most permissive system to be the easiest to build models for, but also the least efficient to verify them. Like other model checkers, Spin attempts to find a balance between ease of use and model checking efficiency.

The preferred way to build a model for any given verification problem at hand is to follow the following steps:

- ***first*** decide which correctness properties are relevant, and require formal proof
- ***next*** study the essence of the solution that is used in the application to secure the behavior of interest (i.e., that determines correctness with respect to the properties selected in the first step)
- ***last*** construct an executable abstraction for the application that has enough expressive power to capture the essence of the solution, and no more.

It is possible, but not really likely, that one could obtain the proper abstraction by literally transcribing the text of an implementation, say in Java or in C++. It is not even likely that the (human) model checker can fully understand the essence of an application by studying its source code. (It is as likely as someone understanding the essence of a Quicksort algorithm from reading C-source code, instead of the paper introducing the rationale behind the algorithm.)

The purpose of the construction of the model is to attempt to **disprove** the correctness of the chosen solution: to find logical flaws in the reasoning that produced the design, rather than finding mere coding errors in its implementation (there are other, perhaps even better, methods to do that). This means that the model establishes a refutable statement about a design. As Lakatos phrased it: ‘‘*the purpose of analysis is not to compel belief but rather to suggest doubt*’’ [2]. This means that elements of a model that cannot contribute to its refutability can and should be deleted in the interest of enhancing the models tractability (and the possible refutation of the remainder).

In the last two decades formal verification tools have evolved from simple reachability analysis tools, run on machines with often less than 1 Mbyte of main memory, into powerful symbolic and on-the-fly logic model checking systems, that can be run on machines with over 1 Gbyte of memory. Computational complexity (or the familiar ‘state space explosion’ problem) was considered to be the dominant issue twenty years ago, when tools could handle no more than about 10,000 reachable states, and it is still perceived to be the dominant issue today, now that model checking tools can handle state-spaces that are many orders of magnitude larger. Another twenty years from now these limits will undoubtedly have improved by another few orders of magnitude (if only because of predictable increases in memory sizes and CPU speeds), but the challenge to build tractable models will remain. The problem in model checking is not computational complexity but our still limited skill in building abstractions.

Users of verification tools often attempt to build models that remain relatively close to the implementation level of an application, making syntactic changes to accommodate the specification language of a model checker, and they often only seriously consider abstraction methods when a concept or feature is encountered that cannot be represented directly in the language of the model checker. At this point, the user is often frustrated, which is of course at that point in the effort makes for a poor motivator in the search for a good abstraction.

Much of the detail included in verification models, especially by new users, is often functionally irrelevant to the verification chore at hand, yet can seriously impede the chances for its successful completion.

3. AVOIDING REDUNDANCY

Some small examples can illustrate that the success of the model checking process relies on the skill of the user to design powerful executable abstractions and to avoid redundancy. The paradigms that are used here are often borrowed from simulation or regular programming practice, but they can be counter-productive when used in model checking.

3.1 Counters

In debugging a regular program, or in the construction of a simulation model, it is often convenient to add counters, for instance, to keep track of the number of steps performed.

```
int cnt = 1;
do
:: can_proceed == true;
        ... /* perform a step */
        cnt = cnt +1;
        printf("step: %d\n", cnt)
od
```

Figure 1. Counter Example

The *Counter* example in Figure 1 illustrates this in a sample Promela program. It has two flaws. The first flaw is the use an unrestricted integer as the default data-type for a new variable. In model checking our concern is to build tractable models from strictly bounded components. A variable is just another component of the model, and it too should have a bounded range, preferably as small as possible. The variable is used here as if it were a natural number with infinite range Note that no check for overflow of the value assigned to cnt is made, on the implicit assumption that no overflow condition could practically occur. This assumption is valid for debugging or simulations, it is false in model checking.

It is not necessarily a problem that the variable cnt can take up to 32 bits of storage in each reachable state that is generated during the model checking process. The real problem is that cnt can store 2^32 (more than 4 billion) distinct values, and therefore has the potential of multiplying the size of the reachable state space for the model by the same amount. The second flaw seals the fate of this model, whatever its further contents may be: the use of the integer to count steps without apparent bound. What otherwise may be a simple iteration that may be verified exhaustively within a few CPU instructions, is now unfolded 2^32 times, and becomes intractable. Phrased differently, removing the counter can reduce the complexity of a verification run by about nine orders of magnitude...

3.2 Sinks, Sources, and Filters

Another avoidable source of complexity is the addition to an abstract model of a process that acts solely as a source, a sink, or a filter, for a sequence of predetermined messages. The flaw is that such additions have no refutation power, and do not contribute in an essential way to the behavior that is being represented. Removing the associated actions from the abstraction can often be done while preserving all options for behavior, and options for the satisfaction or violation of correctness properties elsewhere in the model.

The sink process in Figure 2, for instance, consumes and discards a stream of messages, without ever changing state. In this model, the sink is always ready to remove a message from the channel, and whichever process is sending to this channel can do so unimpeded. If the sink process is the *only* process reading from the channel q, and *no other* messages than those of the three listed types can be send to q, the abstraction made here is not functionally changed if one deletes the sink process, together with all send actions on channel q. In fact, the refutation power of the model will be increased, by the reduced complexity.

```
mtype = { one, two, three };

chan q = [8] of { mtype };

proctype sink() {
        do
        :: q?one /* discard */
        :: q?two
        :: q?three
        od
}
```

Figure 2. Sink

Note carefully that even though process sink has only one control state, channel q is considerably more complicated, and can needlessly increase the complexity of the model checking problem.

To see how much would be saved by removing process sink, consider the number of states that channel q might be in. The channel can hold between zero and eight distinct messages, each of which is one of three possible types:

$$\sum_{i=0}^{8} 3^i = 9841$$

This means that removing the process and the channel can decrease the complexity of the model checking problem by almost four orders of magnitude, without in any way affecting its outcome. The temptation to include the dummy process is often that the real application contains a process or a task that receives these messages and processes them in a way that need not be modeled. There is an understandable uneasiness in the user to completely discard the entire process in this case.

```
proctype source() {
        do
        :: q!one /* generate messages*/
        :: q!two
        :: q!three
        od
}
```

Figure 3. Source

A very similar example can be constructed if we replace the sink process with a source process, as shown in Figure 3. As before, if the source process is the only process sending messages into channel q, and the three message types listed are the only types expected, the process can again be removed without affecting the functionality expressed by the model. All receive statements on this channel are then replaced with a `skip`, or `null`, statement, and the complexity of the model checking problem may again be reduced by four orders of magnitude.

An alternative to lessen the effect of the redundant process on the complexity of the verification would in both these cases be to reduce the channel capacity to one, or to replace the channel `q` with a rendezvous port.

The third variant of the dummy process paradigm is the use of a filter process that merely transfers every message received on its input channel to an output channel, as illustrated in Figure 4.

```
proctype disk driver(chan in, out )
{       mtype msg;
        do
        :: in?msg -> out!msg
        od
}
```

Figure 4. Filter

In this case, the problem is slightly more subtle, because there may be relevant behavior that is contributed by the presence of the filter process. Note that after receiving a message on the input channel, only the output action can take place. If it can be shown that this output action can never be delayed, or that its delay cannot alter the behavior of other processes in the model, the process can again be removed from the model.

4. SIMPLE REFUTATION MODELS

Is it realistic to expect that we can build models that are of practical significance and that remain computationally tractable? We consider two examples of remarkably simple models that have this property. The first model, discussed in Section 4.1 counts just 12 reachable states, and thereby qualifies as perhaps the simplest model of a realistic problem yet published. The second model is not much larger, with 51 reachable states, yet it too has demonstrable practical value.

A naive model for either of these examples could easily defeat the capabilities of the most powerful model checking system. By finding the right abstraction, though, we can demonstrate that the first model contains a design flaw, and we can prove the other to be a reliable template for the implementation of device drivers in an operating systems kernel. The two abstractions discussed here require less model checking power than what is available on the smallest of PCs. To be sure, it is often harder to find a simple model than it is to build a complex one, but the effort to find the simplest possible expression of a design idea can provide considerably greater benefits.

4.1 The Pathfinder Problem

NASA's Pathfinder landed on the surface of Mars on July 4th 1997, releasing a small rover to roam the surface. The mechanical and physical problems that had to be solved to make this mission possible are of course phenomenal. Designing the software to control the craft may in this context seem to have been one of the simpler tasks, but designing any system that involves concurrency is challenging and requires the best of tools. Specifically, in this case it was no easier to design the software than the rest of the space craft. It was indeed only the control software that noticeably failed during the Pathfinder mission. A design fault caused the craft to lose contact with earth at unpredictable moments, causing valuable time to be lost in the transfer of data. The nature of the bug was traced down, and fixed within a few days. It was tracked down through exhaustive system testing with a duplicate of the craft at JPL, in attempts to reproduce the unknown non-deterministic sequence of events that caused the real craft to fail [8].

```
mtype = { free, busy, idle, waiting, running };

show mtype h state = idle;
show mtype l state = idle;
show mtype mutex = free;

active proctype high_priority()
{
end:    do
        :: h_state = waiting;
                atomic { mutex == free ->
                        mutex = busy };
                h_state = running;

                /* produce data */

                atomic { h_state = idle;
                        mutex = free }
        od
}

active proctype low priority()
        provided (h_state == idle)
{
end:    do
        :: l_state = waiting;
                atomic { mutex == free ->
                        mutex = busy};
                l_state = running;

                /* consume data */

                atomic { l state = idle;
                        mutex = free }
        od
}
```

Figure 5. Pathfinder Model

The flaw was a conflict between a mutual exclusion rule and a priority rule used in the real-time task scheduling algorithm. The essence of the problem can be modeled in an executable abstraction in `Promela` in a few lines of code, as shown in Figure 5. Two priority levels are modeled here as `active proctypes`. Both processes need access to a critical region for transferring data from one process to the other, which is protected by a mutual exclusion lock. If by chance the high priority process starts running while the low priority process temporarily holds the lock, neither process can proceed: the high priority process is locked out by the mutex rule, and the low priority process is locked out by the priority rule, which is modeled by a Promela `provided` clause.

The model shown here captures the essence of this problem in as few lines as possible. The resulting state space counts no more than twelve (12) reachable system states. which should be compared to the many billions of possible states of the real memory-module in the Pathfinder controller that were searched in exhaustive tests of the non-abstracted system. A complete reachability graph for this system is readily built, even by a human model checker without computerized assistance. It is a minor challenge to identify the two possible deadlock trajectories in that graph.

It should be added here that if an abstraction of the mutual exclusion and priority scheduling rules had been constructed *before* the launch of the Pathfinder mission, before the design flaw had manifested itself, the model would likely have contained more detail. In the real system, for instance, a third intermediate priority level was responsible for keeping the low priority process from releasing the lock, which prevented the high priority process from completing its task. Similarly, there were also other tasks in the system that manipulated the mutual exclusion locks. The additional priority levels and tasks would naturally be made part of the verification models, to either locate or rule out all suspicious behaviors.

There is little doubt, though, that the flaw in the Pathfinder software could have been detected prior to launch, with a relatively modest model checking effort.

4.2 A Disk Scheduling Problem

The next example illustrates how an abstract model can be constructed, again in just a few lines of high level code, to prove that a particular implementation method has the properties that it is intended to have. The problem is a typical operating system's design problem: scheduling access of multiple 'client' processes to a single disk IO server. Only one client request can be served at a time; if multiple requests arrive, they have to be queued and served in first-in first-out order.

In the original model for this problem, designed by Pieter Villiers as part of a study on verifiable micro-kernel designs [7]. Separate process types were defined for modeling

- the client processes,
- the disk scheduler,
- the disk controller, and
- the disk device driver.

To initiate a disk IO operation, the client process submits a request to the disk scheduler, where it may get queued. When it is ready to be serviced, the scheduler will send a start command to the disk controller, which will initiate the IO operation with the device driver. Completion of the operation is signalled by a hardware interrupt, which is intercepted by the scheduler. The internal details of the device driver (e.g., mapping disk blocks to cylinders, sending commands to move the disk heads, etc.) are quite irrelevant to the problem of verifying that the queueing of client requests in the disk scheduler is designed correctly. In the first model, therefore, the abstraction for the device driver looked like shown in Figure 6.

```
proctype Contr (chan req, signal)
{       do
        :: req?IO ->
                /* perform IO operation */
                signal!interrupt
        od
}
```

Figure 6. Device Driver Interface Model

This has the signature, within the model, of a ***filter*** process, just like the behavior of the client process (submit a request and wait for the result.) Similarly, the disk controller and the client processes behaved like filters in this context, all of which merely states that the focus of the correctness issue considered here is on the scheduler. The temptation is to include the extra processes in the abstract model anyway, simply because the application contains them. For the verification itself, however, their presence serves no real purpose, and an effort should be made to remove them by applying abstraction rules. Doing so leads to the remarkably simple, yet complete, model shown in Figure 7.

```
#define Client(x)client busy[x] == false -> \
                        client busy[x] = true; \
                        currentproc = x+1; \
                        goto progress

#defineDiskIdle activeproc == 0
#define DiskBusy !(DiskIdle)

#define NCLIENTS 3

active proctype disk scheduler()
{       chan request q =
                [NCLIENTS] of { byte };
        bool Interrupt = false;
        bool client busy[NCLIENTS] = false;
        byte activeproc, currentproc;

        do
        :: Client(0);
        :: Client(1);
        :: Client(2);
progress:       if
                :: DiskIdle ->
do_diskio:      activeproc = currentproc;
                        assert(Interrupt == false);
                        Interrupt = true
                :: DiskBusy ->
                        request_q!currentproc
                fi
        :: Interrupt ->
                Interrupt = false;
                client_busy[activeproc-1] = true;
                if
                :: request q?currentproc;
                        goto do diskio
                :: len(request q) == 0 ->
                        activeproc = 0
                fi
        od
}
```

Figure 7. Disk Scheduler Model

With the client process modeled as a filter, the client is unable to submit a new request until the last one was completed first. In the simplified model this busy status is recorded in a boolean array `client_busy[NCLIENTS]` instead. Two correctness properties are integrated into the model: a local assertion states that no new interrupt can be generated before the last one was handled. It's validity follows from the fact that no new disk IO operation can ever be initiated before the last one was completed.

The second correctness property is expressed in a 'progress label' which appears at the point in the code where a new client request is submitted. Spin can prove that 'non-progress' cycles are impossible for this design This means that there cannot be any infinite execution without infinite progress for at least some client. Note that there are only finitely many clients, and clients cannot resubmit a request for service before their previous request was completed.

What is the complexity of this model? With two clients, the state space contains no more than 51 distinct system states. With three clients, as shown, this number increases to 211 system states, neither of which is much of a challenge for any existing model checking tool.

But, are three clients sufficient to prove the required properties for any number of clients? Note that increasing the number of clients increases the number of system states to be inspected due, primarily, to the increased number of permutations of distinct client requests in the request queue. All clients behave the same, and it should therefore suffice to prove each client-specific property for one arbitrarily chosen client.

The context is comparable to the one encountered in Wolper's proof for the ***data independence*** theorem [9]. In Wolper's case one would like to prove that two arbitrary data items cannot arrive out of order in a data stream. Three distinct types of data, marked, for instance, ***a***, ***b***, and ***c***, suffice to prove this property for *any* number of data items. Specifically, if it can be proven that the data items ***b*** and ***c*** cannot arrive out of order when embedded in the infinite stream ***a* b a* c a****, then it follows that no two data items, whatever their markings, can ever arrive out of order.

In this case we must show that a client-specific property holds no matter at what point in the disk scheduler's operation the client's request may arrive. There are three boolean conditions that together completely determine the state, and the actions, of the disk scheduler when a new request arrives. They are:

- `len(request_q) ==0`
- `DiskIdle`
- `Interrupt`

This gives us maximally eight states to consider. One client process clearly cannot cover all of these, because in that case the truth assignment to the three conditions is always: true, true, false when a new request is submitted. With two clients we reach more combinations, but we still could not have both `Interrupt` be true while `len(request_q)` is nonzero. Three clients is the minimum number needed to cover all eight cases. Adding more clients can therefore increase the complexity of the verification, but not its scope.

The model shown here is, because of its simplicity, interesting in its own regard, but it is more than just a model checking toy. Villiers reports that the original model was successfully used as a guideline for the implementation of several device driver modules for a commercial operating system [7].

5. CONCLUSION

The two applications discussed here require the search of state spaces of, respectively, 12 and 211 reachable system states to be exhaustively verified. But the need to produce a simple verification model is not nearly as strict as what is suggested by these numbers. A model checker like Spin, for instance, can analyze models at a rate of 10,000 to 100,000 states per second, depending on the size of the state descriptors, and the speed of the CPU. The power of a model checker therefore should in most cases suffice to tackle even the most challenging design problems. If no tractable model can be found within these bounds, would it be fair to conclude that the design itself is insufficiently understood to be verified, let alone be implemented?

6. REFERENCES

[1] Holzmann, G.J. An analysis of bitstate hashing. *Proc. 15th Int. Conf. on Protocol Specification, Testing, and Verification, PSTV95.* Warsaw, Poland. Chapman & Hall, London (1995), 301-314. Journal version to appear in: *Formal Methods in Systems Design*, (1998).

[2] Lakatos, I. *Proofs and Refutations: the logic of mathematical discovery*. Cambridge University Press, 1976.

[3] Russell, B. *The Principles of Mathematics*. Cambridge University Press. 1903 Vol. 1, par. 78 and Ch. X, 1903.

[4] The source to the model checker Spin is available from URL: http://netlib.bell-labs.com/netlib/spin/.

[5] Strachey, C. An impossible program. *Computer Journal*, 7, 4 (January 1965), p. 313.

[6] Turing, A.M. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Mathematical Soc.*, Ser. 2-42, (1936), 230-265, see p. 247.

[7] Villiers, P.J.A. *Validation of a micro-kernel: a case study*. Ph.D. Thesis, University of Stellenbosch, S. Africa, (Draft of December 1997).

[8] Wilner, D. Vx-Files: What really happened on Mars. (keynote address.) *Proc. IEEE Real-Time Systems Symposium*, Dec. 2-5, 1997, San Francisco, CA.

[9] Wolper, P. Specifying interesting properties of programs in propositional temporal logic. *Proc. 13th ACM Symposium on Principles of Programming Languages.* St. Petersburg Beach, Fl., January 1986. 148-193.