Trends in Software Verification

Gerard J. Holzmann

JPL Laboratory for Reliable Software
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91006
gerard.j.holzmann@jpl.nasa.gov

Abstract. With the steady increase in computational power of general purpose computers, our ability to analyze routine software artifacts is also steadily increasing. As a result, we are witnessing a shift in emphasis from the verification of abstract hand-built models of code, towards the direct verification of implementation level code. This change in emphasis poses a new set of challenges in software verification. We explore some of them in this paper.

1. Introduction

In the last few years, we have seen a push towards the direct application of formal verification techniques to implementation level code, instead of to manually constructed high-level models of code. Although the direct application of, for instance, model checking techniques to implementation level code can significantly increase the computational requirements for a verification, the promise of this new approach is that it can eliminate the need for expert model builders and can place the power of automated verification techniques where it belongs: in the hands of programmers.

There are two general approaches to the software verification problem in this form.

- Mapping the implementation level description of the software artifact mechanically to the description language of an existing verification tool. The application is rewritten to match the requirements of a given verification tool.
- Developing a verification tool that can work directly on implementation level descriptions. The verification tool is rewritten to match the requirements of a given implementation language.

Examples of projects pursuing the first method include the first Java Pathfinder tool [5], the Bandera toolset [4], and the FeaVer toolset[1] [8], which all target the SPIN model checker[2] [7,10] as the main verification engine. Examples of projects pursuing the second method include the second version of the Java Pathfinder tool [2], Microsoft's Bebop toolset [1], and the Blast tool [6].

Of the six projects mentioned, three target the Java programming language ([2,4,5]), and the remaining three target the C programming language.

The two methods have different advantages and disadvantages. The first makes it possible to leverage the power of an existing tool, and to trust the validity of the

1. http://cm.bell-labs.com/cm/cs/what/feaver
2. http://spinroot.com/whatispin.html

verification process. The second method, on the other hand, makes it possible to leverage the efforts that have already been spent in the creation of the software artifacts and to trust their accuracy, rather than the accuracy of a newly developed translator. In other words: the first method tries to secure that the application is verified correctly, while the second method tries to secure that the correct application is verified. The most significant challenges that each method poses are as follows.

- The first method requires the construction of a model extractor that can convert implementation level program descriptions into detailed verification models that can be submitted to a model checker. To perform the conversion accurately, we need to be able to interpret the semantic content of the implementation level code (e.g., written in C) and convert it into equivalent representations in the verification model.
- The second method requires the construction of a verifier that can pass accurate judgements on the validity of a system execution. The construction of a comprehensive verification system for any formally defined language can be a significant challenge. Doing so for an implementation level language, that was not designed with verifiability in mind, can be even more challenging.

It would seem that both methods face significant hurdles, and are difficult to combine. As it turns out, though, many of the difficulties that are encountered by these two approaches can be overcome with a third technique. This technique is based on the use of embedded code statements inside a traditional model checker.

2. Embedding Code vs Translating Code

A model checker is programmed to systematically explore the reachable state space of a (model) system. As far as it is concerned, the world consists only of states, and state transformers. It renders its verdicts with the help of sets of boolean propositions on states and state sets. Within the model checker, a system state is defined as the set of control-flow points, and value assignments to data objects, where the data objects are restricted to the ones that are definable within the specification language. State transformers, similarly, are defined by the set of executable statements that are predefined in the specification language. So all the model checker does is to provide the user with a carefully designed language for the specification of systems of states and state transformers. There is a pleasing similarity here with a mathematical theory that is defined by a small set of axioms (the initial system state), a small set of rules of inference (state transformers), and a potentially much larger set of provable theorems (the reachable states).

A programming language, just like the specification language for a model checker, allows us to specify systems of states and state transformers. The main difference with a model checking language is that no provision is generally made to keep the system finite or to secure that the properties of the system remain decidable. We will postpone a discussion of the issue of decidability for now and consider just the notion that the purpose of a software application is merely to define systems of states and state transformers. The first strategy for model checking software systems that we mentioned above required us to translate the possibly unwieldy specification from a mainstream programming language into the more structured specification language of a model checker: replacing one system of states and state transformers with another. This is necessarily a hard problem since it will require us to faithfully map semantic concepts from one language into another.

Having recognized that, at least at some level of abstraction, both the programming language and the model checking language perform the same type of function, we may wonder if it would not be possible to use the programming language directly to define a system of states and state transformers and to let the model checker add only its checking engine. We can do so by *embedding* descriptions from the source programming language directly into the target model that will be verified by the model checker.

Doing so, we can combine the benefits of both approaches outlined above, while avoiding all the work that would be needed to solve the hard part of the problem in both domains. For the first approach this means that we can avoid having to develop a method that would allow us to provide an accurate interpretation of source C code, such that it can be mapped into the target language of the model checker. For the second approach it means that we can avoid having to develop an efficient model checking system for a new language from scratch.

SPIN is designed to generate a verification program in C, to perform the model checking task for a high-level system model. To do so, SPIN interprets the state descriptors and state transformers as the user specified them in PROMELA (the SPIN input language), and converts them into C code, thereby fixing their semantic interpretation. Rather than having a new translator convert native C code into PROMELA, and have SPIN convert the PROMELA code back into C, we can try to bypass the translation steps and use the original C code to define elements state transformers within the verifier directly. Ultimately, it is now the C compiler that determines the semantics of the C code, just like it does when we compile the application level code directly for execution.

To support these ideas, SPIN Version 4 introduced a small set of new language primitives. The most important of these are: c_code, c_expr, and c_state.

c_code	The c_code primitive allows us to include an arbitrary fragment of C code as a formal state transformer in a SPIN model.
c_expr	The c_expr primitive can be used to evaluate an arbitrary C expression and to interpret the return value as a Boolean condition (non-zero meaning *true* and zero meaning *false*).
c_state	The c_state primitive, finally, can be used to embed an arbitrary global or a local C data object into the state descriptor that is maintained by the model checker.

With the help of these three primitives it now becomes possible to build an accurate model of a large class of routine C applications with relatively little effort.

3. Separating Data and Control

It is of course not sufficient to simply encapsulate an entire C program and pass it to the model checker to execute: the model checker needs to be able to control the execution of the program. Consider, for instance, the execution of a concurrent system, with multiple threads of execution being able to access and modify a pool of shared data objects. There could well be race conditions in the code that depend on the particular access pattern that is followed: the specific interleaving of statement executions. Unless the model checker is in charge of these interleavings and can schedule the statement executions one by one, we may miss these errors. So by necessity we need to devise a system that can separate control and data. Control in a C program is defined with the help of control flow constructs such as the semi-colon (for sequential execution) the if-then-else statement (for conditional branching), the for- or while-loop (for iterations), and goto statements and labels (for unconditional branching). The control structure of a program can be visualized in a control-flow graph, where nodes represent control-flow states, and edges represent statement executions (i.e., the basic state transformers).

The SPIN extension exploits the fact that we can fairly easily separate the control aspects of a program from the data aspects. We can translate the control aspects, and leave the data aspects untouched, embedding them as-is into a verification model, so that their effect as state transformers is fully and accurately represented.

```
#include <stdio.h>

int
main(void)
{       int lower, upper, step;
        float fahr, celsius;

        lower = 0;
        upper = 300;
        step = 20;

        fahr = lower;
        while (fahr <= upper) {
                celsius = (5.0/9.0) * (fahr - 32.0);
                printf("%4.0f %6.1f\n", fahr, celsius);
                fahr = fahr + step;
        }
}
```

Figure 1. Example C program.

MODEX

We designed a model extractor,[3] called MODEX, to convert simple C programs mechanically into SPIN models, following the principles given above. The model extractor derives the control flow graph of a program, using standard parsing techniques, it expresses the control-flow constructs of the source program into the corresponding control-flow constructs of SPIN's input language (a relatively straightforward procedure), and it embeds data declarations and basic statements into the model with the help of the new embedding primitives from SPIN. As a simple example, the model extractor can mechanically convert the C program shown in Figure 1 into the SPIN model that is shown in Figure 2.

The details of the model extraction process are not too important for this paper, but note that through the use of embedded declarations and embedded code the model checker can now access and manipulate floating point variables, even though SPIN itself does not support the associated data type. The two floating point variables `fahr`

3. http://cm.bell-labs.com/cm/cs/what/modex

and `celsius` are embedded here into the state vector as local objects of the `main` process. The model extractor automatically arranges for the variable references to be prefixed with pointers into the appropriate part of the verifiers state descriptor, in such a way that any reference to, for instance, `fahr` becomes `Pmain->fahr`.

In a similar way we can generate models that use pointers, even function pointers, though there is no direct support for any of these language features at the SPIN level.

```
c_state "float fahr" "Local main"
c_state "float celsius" "Local main"

active proctype main()
{   int lower;
    int upper;
    int step;

    c_code { Pmain->lower=0; };
    c_code { Pmain->upper=300; };
    c_code { Pmain->step=20; };
    c_code { Pmain->fahr=Pmain->lower; };

    do
    :: c_expr { (Pmain->fahr <= Pmain->upper) };
       c_code { Pmain->celsius =
                    ((5.0/9.0)*(Pmain->fahr-32.0)); };
       c_code { Printf("%4.0f %6.1f\n",
                    Pmain->fahr, Pmain->celsius); };
       c_code { Pmain->fahr = (Pmain->fahr+Pmain->step); };
    :: else -> break
    od
}
```

Figure 2. SPIN Model Corresponding to Figure 1.

There are limits to how much can be automated with this approach. Consider, for instance, how function calls, like `printf` in the example, are handled. Without special provision, MODEX considers a function call to be an atomic event, and the code that is generated will not return control to the model checker until the function is completely executed. This is the right policy for the `printf` call. To allow the model checker to look inside a function, though, we need to give additional instructions to the model extractor. This means that we still need to rely on human judgement to determine which functions need instrumenting, and which can be left alone.

To apply the model checking algorithm, the model checker must be able to set the application into any one of its reachable states. This means that the state descriptor that is maintained by the model checker must always contain a complete description of the (relevant part of the) state of the system. If any part is missing from this description, then that part of the system state will not get updated accurately when the verifier places the system into a new state.

A potential problem now exists if the application can maintain part of its system state external to the application. This can happen, for instance, if the application stores or reads data from the file system, if it communicates through live network connections with other systems, and even if it can dynamically allocate memory for new data objects. In the latter case, the memory allocator, maintaining heap memory, is an external resource where some of the relevant system state information is maintained.

All these issues can be resolved, but currently require some degree of user intervention into the model extraction process. A more detailed treatment of these issues can be found in [9,10].

4. Decidability

A SPIN verification model must satisfy two conditions to secure the decidability of the verification problem. First, the model must be self-contained. There can be no hidden assumptions, and no undefined components that contribute in any way to the behavior that is being verified. Second, the model must be bounded. This means that when an execution of the model is simulated, only a finite number of distinct system states can be reached. The number can be large, but it must be finite.

If verification models are specified in SPIN's native specification language PROMELA, then both fitness requirements are automatically satisfied. It is impossible to define a non-finite state SPIN model in PROMELA. All data objects are bounded, the capacity of all message channels is bounded, and there is a strict limit on the number of asynchronous process threads that can be created in a system execution. This secures the decidability of all correctness questions that can be expressed with SPIN, which if formally the class of ω-regular properties, and which includes the set of properties that can be defined in standard linear temporal logic [12].

But the same is not necessarily true for SPIN models that contain embedded C code. If the model is self-contained and bounded, decidability is retained. Reflect for a moment on how the model checker would recognize a runaway C program: one that lands itself in an infinite loop. First note that the model checker maintains a state descriptor in memory, recording all information that holds state information for the application. When the program starts executing an infinite loop, the model checker will detect that previously visited states are repeated in the execution. It can analyze the cycle for the potential violation of liveness properties, and complete its work normally. The cycle is merely a traversal of a strong component in the reachability graph of the system, which the verifier can recognize as it builds that graph.

If the application is not finite-state, it must be able to increase the size of the state descriptor without limit. If this happens, the verifier will sooner or later run out of its limited resources to track the execution, making complete verification impossible. In truth, the application itself, when run standalone, would encounter the same problem, and when it reaches the point where it exhausts the available system resources it too would have to abandon its execution. In real-life, at least to date, the deliberate design of a program that is fundamentally infinite state is not sensible. If it occurs, it is usually the result of a design error, and not a planned feature of a program.

The Halting Problem

But, how do we square this observation with the unsolvability of the *halting problem*, which is one of the best known results in theoretical computer science [14]. In rendering the proof for the unsolvability of the halting problem one normally does not distinguish infinite state programs from finite state ones. As an example, let us consider a popular variant of such a proof, as it was given by Christopher Strachey in 1965 [13], which is also used in [11].

Strachey's proof is by contradiction. Suppose we had a procedure, call it `mc`, that could determine for any given program p whether or not it would terminate if executed. The procedure mc(p,i) can then be used to return *true* if it determines that program p necessarily terminates on input i, and *false* it fails to terminate.[4] Naturally, we must assume that mc itself will always terminate in a finite amount of time, so it cannot simply run the program it is inspecting to determine the answer to its question. How precisely it does operate is undefined.

```
strachey(p,i)            /* program p, input i   */
{
L:      if (mc(p,i))     /* true if p halts on i */
                goto L;  /* make strachey() loop */
        else
                exit(0); /* else halt            */
}
```

Figure 3. Strachey's Construction.

Given the procedure mc we can now write the program shown in Figure 3. The program strachey(p,i) is designed to halt when the program p(i) does not, and vice versa.

All is well, untill we ask whether the program strachey(strachey,strachey) will terminates or loops. Clearly, it cannot do either. If it halts, then it must loop, and vice versa.

It is curious that this version of the proof has never been seriously challenged. First, note that the proof argument seems to be independent of the issue of finiteness, and would appear to apply equally to finite state and infinite state programs.

Strachey tacitly assumes in his argument that all programs either halt or loop. In practice, though, there is a third possibility: a program can fail. When a program attempts to divide by zero, or runs out of memory, it is forced to terminate by its environment: it fails. Program failure cannot simply be grouped into the category of program termination, because if this were the case we could apply Strachey's argument to the class of finite state programs.

> Given an upper-bound N bits on the amount of memory that a program can consume, we can derive an upper-bound on the number of reachable states it could generate when executed (trivially 2^N). If we declare that exceeding the upper-bound of N bits of memory constitutes program termination as considered in Strachey's argument, then we can easily decide the outcome of mc(p,i) in finite time: we have to consider maximally 2^N steps of the program. Within this number of steps the program must either terminate or loop.

We can use SPIN to solve the halting problem for finite state programs, using the model extraction procedure we have outlined before. To do so, we first write a UNIX® shell script that returns *true* if SPIN determines that a given model has at least one reachable endstate, and *false* if it does not.

4. In Strachey's version of the proof, the required arguments to procedure mc() are omitted.

```
#!/bin/sh
### filename: halts

echo -n "testing $1: "

spin -a $1                  # generate model
cc -DSAFETY -o pan pan.c    # compile it
./pan | grep "errors: 0"    # run it and grep stats
if $?                       # test exit status of grep
then
        echo "halts"
else
        echo "loops"
fi
```

We can try this on the Fahrenheit conversion model from Figure 2, to check if the scripts gives us the right answer.

```
$ ./halts fahrenheit.pml
halts
```

If we change the loop in this example into a non-terminating one, the script will accurately report that the model will now loop. So far so good. We can now invoke this script in a SPIN `c_expr` statement, in the devious manner envisioned by Strachey.

```
init {  /* filename: strachey */
        do
        :: c_expr { system("halts strachey") } /* loop */
        :: else -> break                       /* halt */
        od;
        false   /* block the execution */
}
```

Returning to Strachey's proof argument: what happens if we now execute

```
$ ./halts strachey
.....
```

After some reflection, aided by performing the actual experiment, we can see that the `halts` script ends up going into an infinite descent. Each time the model checker gets to the point where it needs to establish the executability of the `c_expr` statement, it needs to invoke the `halts` script once more and it must restart itself. This very construction then is *not* finite state. In reality, the infinite recursion cannot go on forever, since our machines are always finite. The process will stop when the maximum number of processes is exceeded, or a maximum recursion depth on nested system calls is exceeded, leading to a crash of the program. Because the `strachey` program is infinite state, it is firmly outside the scope of systems that can be verified with finitary methods. Note carefully that the infinite recursion is not caused by any particular choice we have made in the implementation of the `halts` script. Even if this script only needed to read the source text of the program before rendering a verdict on its termination properties, the same infinite descent would occur.

The executions of Strachey's impossible program, then neither leads to termination nor does it lead to looping: it leads to a failure. Strachey's program itself then belongs to the class of faulty programs (and there are many ways to construct those).

Note that if SPIN can be used to verify the termination properties of systems with up to `N` reachable states, it will itself need considerably more than `N` reachable states to perform this verification. Therefore, SPIN also could not be used to verify itself in another Strachey-like construction. There is much more that can be said on this topic though, cf. [10].

5. Conclusion

A practically useful software tool is usable by any normally skilled programmer, requiring no more tool-specific training than an ordinary language compiler. Since their inception, roughly twenty years ago, formal software verification systems have relied on the construction of a mathematical or computational model of an application, by a domain expert, which is then analyzed either manually or mechanically. Even the fully automated tools that operate in this domain come short of reaching the goal of practically useful software tools as long as they rely on human experts to construct the input models.

The emphasis of much of the work in the area of formal verification has therefore recently been placed by some groups on the automatic generation of logic models from implementation level code, and by others on the adaption of the verification tools themselves to work directly on implementation level code. We have shown that these two seemingly distinct approaches can effectively be combined, by allowing the embedding of implementation level code into higher-level logic models that can then be verified with existing model checking techniques. The technique we have described relies on the fact that we can separate the control aspects of a program from the data manipulation. The control aspects of a program can in most cases trivially be adapted to the syntax requirements of the logic model checker, while the data aspects (which are much harder to convert) can be embedded.

Limitations: There remain clear limitations to this approach. If *most* control aspects can easily be handled in this way, this does not mean that *all* will fit the default pattern. The use of function pointers in C programs, for instance, needs special care, as does the use of dynamic memory allocation, and access by a program to external sources of information. It may be possible to develop a methodology, though, by which cases such as these can be handled more or less routinely in the construction of a *test-harness* for the application to be verified. A beginning with such a development can be found in the user guide to the Bell Labs FeaVer system [9].

It is also clear that the model checker cannot defend itself fully against outright errors within code that is embedded inside the logic models that it analyzes. Consider, for instance, what happens if such code contains a divide-by-zero error, or dereferences a nil-pointer. A model extractor can be somewhat proactive, and instrument the embedded code with additional checks. Our MODEX tool, for instance, inserts an assertion before any pointer dereference operation, to make sure it is non-zero. Not all errors can be anticipated, and some can cause the model checker to crash, just like the application being verified. There is still benefit to the use of the model checker, even in these cases, since the model checker will be far more likely to find the cases where application code may crash, as part of its search process. A crashed model checking run, like a real execution, leaves a detailed trace of the steps in the program that led to the failure, making it possible to diagnose and repair the code.

Decidability issues: The fact that we can do model checking on at least some categories of implementation level code may at first seem to conflict with long established decidability results, but can easily be seen to be bound by all familiar limits. Other approaches to the software verification problem, such as static analysis and approaches based on theorem proving methods, naturally share this fate. As we hope to have shown, though, the existence of these limits need not prevent us from building systems that are both practically useful, *and* reliable.

2Acknowledgements

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

1. T. Ball, R. Majumdar, T. Millstein, S.K. Rajamani, Automatic Predicate Abstraction of C Programs, Proc. PLDI 2001, f2SIGPLAN Notices, Vol. 36, No. 5, pp. 203-213.
2. G. Brat, K. Havelund, S. Park, W. Visser, Java PathFinder - A 2nd generation of a Java model checker, *Proc. Workshop on Advances in Verification*, Chicago, Ill., July 2000.
3. E.M. Clarke, O. Grumberg, and D. Peled, *Model checking*, MIT Press, January 2000.
4. J.C. Corbett, M.B. Dwyer, et al., Bandera: Extracting finite-state models from Java source code, *Proc. 22nd Int. Conf. on Software Engineering*, June 2000, pp. 439-448.
5. K. Havelund, and T. Pressburger, Model Checking Java Programs Using Java PathFinder, *Int. Journal on Software Tools for Technology Transfer*, Vol. 2, No. 4, April 2000, pp. 366-381.
6. T.A. Henzinger, R. Jhala, et al., Software Verification with Blast, Proc. 10th SPIN Workshop on Model Checking Software, LNCS 2648, Springer-Verlag, 2003.
7. G.J. Holzmann, The Model Checker SPIN, *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, May 1997, pp. 279-295.
8. G.J. Holzmann, and M.H. Smith, An automated verification method for distributed systems software based on model extraction, *IEEE Trans. on Software Engineering*, Vol. 28, No. 4, April 2002, pp. 364-377.
9. G.J. Holzmann, and M.H. Smith, *FeaVer 1.0 User Guide*, Bell Labs, Dec. 2002, 64 pgs. Online document http://cm.bell-labs.com/cm/cs/what/modex/.
10. G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, ISBN 0-32122-862-6, August 2003.
11. M.L. Minsky, *Computation: Finite and Infinite Machines*, Prentice Hall, Englewood Cliffs, N.J., 1967.
12. A. Pnueli, The temporal logic of programs. *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977, Providence, R.I., pp. 46-57.
13. C. Strachey, An impossible program, *Computer Journal*, Vol. 7, No. 4, Jan. 1965, p. 313.
14. A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Mathematical Soc.*, Ser. 2-42, 1936, pp. 230-265.