Programming Research Group

GAME-BASED SOFTWARE MODEL CHECKING: CASE STUDIES AND METHODOLOGICAL CONSIDERATIONS

Dan R. Ghica
Oxford University Computing Laboratory
Oxford, U.K. OX1 3QD

PRG-RR-03-11

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

We introduce a new software modeling and model-checking tool, based on game semantics. The distinctive feature of the tool is its ability to model and verify *program fragments*, i.e. programs with undefined, non-local, identifiers. Through case studies, we show how this ability is useful in modeling certain classes of programs, such as protocols or modules. This feature is shown to be essential in the verification of abstract data-type invariants and equational properties. We explain how the *fully abstract* models generated by the tool can economically represent very large internal state spaces.

1 Introduction

Semantics-directed verification, also known as *model-checking*, has produced in recent years renewed enthusiasm in the area of software verification. One of the most appealing features of software model checking (SMC) is that it is more programmer-friendly than the main alternative technique of proof-checking. SMC is more suitable to automation, it requires less sophistication from the user, and it is counter-example driven, i.e. *it finds bugs.*

There are, however, important challenges and limitations for SMC. For fundamental reasons of computability, there cannot be a general static verification method for arbitrary programs or specifications. Even decidable programs and specifications may be in practice unverifiable because of the high complexity of the algorithms. These algorithmic challenges are tackled using a variety of techniques such as abstract interpretation and program slicing; intense research is ongoing in this area.

In addition to the critical challenge of managing complexity, SMC has to deal with other, more subtle problems [Kur97]:

- *Local versus global verification.* Current SMC techniques cannot handle program fragments, that is programs with non-local identifiers, especially procedures. Consequently, only whole programs can be checked. But since industrial programs are large, and SMC is computationally demanding, the verification of large scale programs using this method is not feasible. In industrial jargon this problem is sometimes referred to as a *scalability problem* for SMC. Also, the requirement for whole programs make it impossible to verify modules or libraries.

- *Source-level versus model-level verification.* Many SMC techniques are adaptations from hardware verification. The program needs to be mapped into a finite-state (FS) system, which is then verified. However, the mapping itself is not perfect and may introduce errors.

- *Unity of programming and specification languages.* SMC techniques derived from hardware verification apply to specification languages based on temporal logics. The natural properties to check in these languages are safety (that an undesirable event does not occur) and liveness (that a desirable event occurs). In the verification of software, however, other formalisms such as assertions, invariants and equational properties are more natural. Expressing the latter in terms of the former is possible but sometimes artificial and awkward for the programmer.

The reason for the difficulties above is an acknowledged *semantic gap* between software and the FS models SMC requires.

Obtaining sound and complete semantic models of software is a notoriously difficult problem; expressing such models in an algorithmic fashion which lends itself to mechanical analysis is an even harder problem. A fortuitous solution to it, however, was provided by the recent application of game-theoretic techniques to the study of programming languages. Although only introduced in computer science in the early 90s, for the study of resource-sensitive logics [Bla92, AJ92], the game-theoretic approach provided answers to the principal, decades-old, open problem of programming language theory, the development of a *fully abstract* (i.e. sound and complete) models for (prototypical) functional languages [AMJ94, HO00, Nic94]. Further results followed, concerning procedural languages [AM96], languages with control [Lai97], concurrency [Lai01] and more.

The game-theoretic models are inherently algorithmic, because they are built using the basic, simple and uninterpreted notions of *move* and *strategy*, which is a set of sequences of moves governed by specific laws of a combinatorial nature. The models used in the works cited above involve complex technical constructions, rather impenetrable to the non-specialist. However, most of the complexity in the model is required just to handle arbitrary higher-order functions. If the programming language is restricted to only its imperative features and first-order procedures, which are the basis of much of the common programming idiom, then the game theoretic model can be greatly simplified [GM00, Ghi01]. Using game-theoretic techniques we can therefore obtain precisely the kind of models needed for SMC: sound, complete and algorithmic.

A specification and verification logical framework for compositional software model checking, based on the game-semantic model, was presented in [Ghi02]. However, since the complexity of the regular-language algorithms required to carry out verification according to the game-based methods is exponential, it was unclear whether the technique was feasible. This is in fact a common situation in SMC: the complexity of the algorithms involved is high (PSPACE, exponential or even semi-decidable) but it (luckily) turns out that the worst-case scenario only happens in pathological cases. Many programs can be in fact verified. But the only way to make such pragmatic assessments is to implement the tool and try it out; we have implemented such a prototype tool, and the initial results are positive.

Contribution: We present a SMC tool based on novel game-theoretic techniques. Compared to existing tools, our method has the distinct feature that it can model fragments of programs, i.e. programs with undefined variables and procedures, thus facilitating compositional model checking. We show how the tool can be applied to model implementations of algorithms, protocols and abstract data types. The resulting models are represented using generic FS machines, and their properties can be verified using (third-party) model-checking tools. We also provide direct model-checking support for code-level specifications: assertions, equational reasoning and invariants for abstract data types. Along the way we identify concrete verification techniques to give a basis for a verification methodology using this new technique.

2 The programming language and its semantics

Our verification tool is targeted at a procedural imperative programming language. It has the usual imperative features (iteration, branching, assignment, sequential composition), first-order (non-recursive) call-by-name procedures and block (stack-allocated) variables and arrays. We will present the programming examples using a C-like syntax.[1] This programming language may seem simple, but the problem of finding an adequate model of the language has been open for decades. Landmark work in the area is collected in the book [OT97]. We prefer the call-by-name procedure mechanism for technical reasons which are outside the scope of this paper. A similar verification tool can be developed for call-by-value languages, along similar lines [Ghi01].

---

[1] The syntax of the programming language used by the tool is slightly more expressive than that of C, but we restrict ourselves here to examples where the differences of syntax are unimportant. We prefer the C-like syntax in this paper for the sake of its presumed familiarity to most readers.

A detailed description of the language and the game-semantic model is given in [AM96]. The technical details of the regular-language representation are given in [GM00, Ghi02]. We will not reproduce these details here, but we encourage the reader to refer to the literature.

We only give an overview of the semantic model and we illustrate it with examples which should provide at least some basic intuitions and an entry point for the non-specialist. Also, a basic understanding of the model is necessary in order to interpret and use the models generated by the tool.

With every type of the programming language we associate an *alphabet of moves*, where a move can be thought of as a basic abstract action:

- integers contain moves q and n; intuitively, they correspond to the action of *asking* an integer expression to evaluate and, respectively, the expression providing the result;
- commands contain moves *run* and *done*; they correspond to *asking* a command to execute and, respectively, to the command acknowledging the completion of its execution;
- variables contain moves *nwrite, ok, q, n*; they correspond to: writing the value n to a variable, having the variable acknowledge the writing, asking the variable for its value and, respectively, having the variable provide its value.

The actions of a procedure will be tagged with:

- the name of the procedure
- an integer n indicating that the actions belong to the n-th argument of the procedure.

For example, consider the procedure **`void`** `p(`**`int`**`)`, which is a command-like procedure taking an integer expression as argument. Its alphabet of moves is:

$$\{p.run, p.done, 1.p.q, 1.p.1, 1.p.2, 1.p.3, \ldots, 1.p.n\}.$$

The interpretation of a program fragment will be a language over the union of all the alphabets of moves induced by all the non-local variables of the program fragment, along with the type of the program fragment itself. Every word in the language can be thought of as an *abstract trace* indicating the actions of the non-local variables appearing in the program fragment.

For example, one possible such abstract trace, for the program fragment consisting just of the command `v = f(x);` is:

$$run \cdot f.q \cdot 1.f.q \cdot x.q \cdot x.7 \cdot 1.f.7 \cdot f.9 \cdot v.9write \cdot v.ok \cdot done,$$

to be interpreted as: start command, evaluate `f`, evaluate `f`'s first argument, evaluate `x`, get the answer 7 from `x`, get 7 from `f`'s first argument, get 9 as a result from `f`, write 9 to `v`, acknowledge the write from `v`, stop execution.

The semantic model of a program fragment is generated by applying rules inductively on the syntax of the program, in compositional fashion.

For example, the semantics for the sequential composition `C1; C2;` of commands `C1` and `C2` is defined as the concatenation of the abstract traces interpreting the two commands: $run \cdot trace(\texttt{C1}) \cdot trace(\texttt{C2}) \cdot done$. This is the simplest composition rule.

The *key* feature of the FS representation generated by the game model is that it is a trace of *observable* actions of, which are of the non-local objects only. The actions of the, locally defined, assignable (block) variables or variable-identifiers denoting procedures, are *unobservable*, hence abstracted. For example, the FS model of program fragment {**`int`** `i; i = 0;` **`return`** `i;`}

is just $q \cdot 0$. Observably, this code fragment is equivalent to the constant `0`; the local variable `i`, since it is truly *local*, is not represented in the model, since it is invisible to the outside world. This high level of abstraction will be seen to be an effective way of greatly reducing the state-space of models.

It is also important to point out that every abstract trace in the model represents a *successfully completed* computation of the program fragment. For this reason, a looping program such as `while (true) { }` is modeled by the *empty* set of traces. This entails that all non-terminating programs are equated.

3 The modeling tool

We have implemented a tool that automatically extracts a FS representation of the game theoretic model of a program fragment. The input to the tool is code, with some type annotations which will be discussed below. The extraction of the model is fully automatic. The output of our tool is just a concrete model of a FS machine, not tied to any particular model-checking tool or formalism.

The properties of the model can be investigated using various standard FS-based model-checking tools. In addition, we also show how this tool can be used directly to model-check code-level specifications such as assertions, invariants and equations. This style of specifications complements well more standard model-checking formalisms such as temporal logics.

3.1 Type annotations and inference for data abstraction

Programs manipulate data that ranges over very large (e.g. integers) or infinite (e.g. strings) sets. In order for model-checking to be feasible we need to abstract away, possibly resulting in a loss of precision of our analysis, from concrete data sets to much smaller abstract data sets. The techniques involved in doing this have been much studied; see [Cou96] for some entry points to the literature.

The basic data type of our programming language are integers. We consider two common data abstraction schemes to be useful in our context:

Congruence arithmetic modulo n : replace the (large) set of integers with a (much smaller) set of integers less than n, and implement arithmetic operations modulo that value. This is a very convenient technique but it needs to be used carefully since it is not always a safe approximation. However, it is useful in spotting many programming errors.

Abstract integers $< n$: use a (small) finite set of integers together with the abstract values of $-\infty$ and $+\infty$. This approximation is known to be safe. However, it is more sophisticated and it introduces non-determinism in the model, since some operations involving infinities are not defined (e.g. $-\infty + \infty$ is non-determined).

The choice of the data abstraction scheme and the size of the finite set of integers is determined both by the program we model and the properties we want to check. In our tool we try to provide maximum flexibility with minimum overhead.

The user indicates the details of the abstraction scheme by using annotations:

- `int%`k indicates congruence arithmetic modulo k
- `int$`k indicates abstract integers up to k.

All occurrences of **int** in the verified program need to be annotated, and the user is free to chose different annotations for different variables. This is quite important for practical reasons: say we want to check an array sorting program. We may be content to restrict the integers stored in the array using `int%2`, i.e. 3 distinct values, but the indices used in the sorting program must have at least as many distinct values as the size of the array in order for the verification to make sense.

In addition to integer identifiers, a program will also use constants and integer operations. Congruence arithmetic modulo k and the arithmetic of abstract integers are very different though. The expression `1+1` can have as result 2 (in `int%3`), 0 (in `int%2`) or $+\infty$ (in `int$2`). However, we do not want to overburden the programmer with annotating every arithmetic operation and constant with an abstraction scheme. Instead, we use a type system with the following coercions:

- **int** $\vdash$ **int**%k
- **int** $\vdash$ **int**\$$k$
- **int**%k $\vdash$ **int**%k' if $k < k'$
- **int**\$$k$ $\vdash$ **int**\$$k'$ if $k < k'$.

The type inference mechanism will find the appropriate abstraction scheme for constants and arithmetic automatically, or produce an error message if the abstraction schemes clash, e.g. as in the annotated fragment `int%3 n; int$3 m; n = m;`.

In addition, the programmer must supply typing information for all non-local variables occurring in the program fragment.

3.2 Implementation notes

The parser and the type-inference parts of the code are written in CAML. The FS manipulation at the back end is done using, for the most part, the AT&T FSM library [atta]. For display and manipulation of the output we use the AT&T GraphViz package [attb].

4 Case studies

In this section we show how the tool may be used to verify three distinct classes of programs: algorithm implementations, protocol implementations and abstract data type (ADT) implementations. The verification of algorithm implementations does not depend crucially on the compositionality of the models, but they provide convincing evidence that our tool can handle theoretically large state spaces. The implementation of protocols and ADTs, however, will be seen to crucially make use of the fact that the model is compositional and can deal with undefined identifiers.

4.1 A warm-up example

The game-based FS models are slightly unusual compared to traditional FS models of software. A key observation is that the *internal state* of the program is abstracted away, and only the actions of *nonlocal* entities are observable, insofar as they contribute to terminating computations.

The warm-up example is a toy ADT: a switch that can be flicked on. Its implementation is:

```
void client(void(), int%2()) |-
  int%2 v = 0;
  void set() { v = 1; }
  int  on()  { return v; }
```

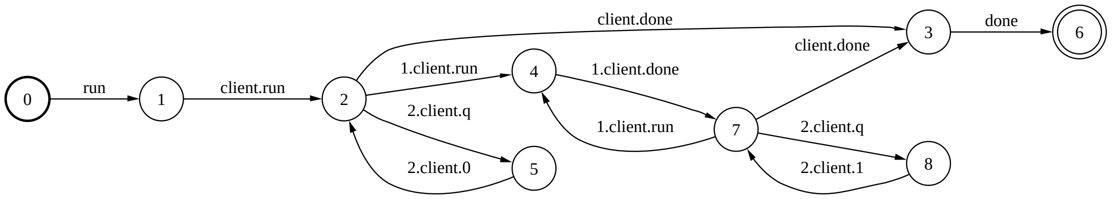

Figure 1: A simple switch

```

  client( set(), on() )
: void.
```

The code consists of local integer (`%2`) variable `v`, storing the state of the switch, together with functions `set()`, to flick the switch on, and `on()`, to get the state of the switch. The initial state of the switch is *off*. The switch implementation is then passed to some `client` code, which accepts a **`void`** function and a **`int`**`%2` procedure as arguments. The `client` function is left undefined. Using the game-theoretic model, the tool generates a "most general" model for the non-local `client` procedure, which acts as a "most general" context. The idea of providing such a generic closure of an open program is not new [CGJ98] but the sound and complete semantic solution afforded by the game model is unique.

The output of the modeling tool is given in Figure 1.

Notice that no references to `v`, `set`, or `on` appear in the model! The model is only that of the possible behaviours of the `client`: whenever the client is executed, if it evaluates its second argument it will receive the value 0 as a result; if it evaluates the first argument, one or more times, then the second argument will always evaluate to 1. The model does not, however, assume that `client` uses its arguments, or how many times or in what order.

Notice that an alternative implementation of the switch:

```
void client(void(), int%2()) |-
  int%2 v = 1;
  void set() { v = 0; }
  int  on()  { return v == 0; }

  client( set(), on() )
: void.
```

has *exactly* the same model as the previous implementation, as the two are observationally equivalent.

4.2 Algorithm implementations

In this section we will discuss the modeling of a sorting program, a notoriously difficult problem. We will focus on *bubble-sort*, not for its algorithmic virtues but because it is one of the most

```
int a0[n] |-
  int a[n];
  int i = 0;
  while (i < n ) { a[i] = a0[i]; i = i + 1; }
  int flag = 1;
  while (flag) {
    int i = 0;
    flag = 0;
    while (i < n - 1) {
      if (a[i] > a[i + 1]) {
        flag = 1;
        int temp = a[i];
        a[i] = a[i + 1];
        a[i + 1] = temp;
        }
      i = i + 1;
      }
    }
  int i = 0;
  while (i < n ) { a0[i] = a[i]; i = i + 1; }
: void.
```

Figure 2: *Bubble-sort* implementation

straightforward non-recursive sorting algorithms. The implementation we will test is the one in Figure 2.

We show the code without the data abstraction annotations. We will discuss the resulting model when several annotations are used. The code also includes metavariable n, which will need to be replaced by a concrete constant in order to use the verification tool.

The implementation is standard for *bubble-sort*. The program first copies the input array `a0[]` into a local array `a[]`, which is then sorted and copied back into `a0[]`. Observe that the array being effectively sorted, `a[]`, is not visible from the outside of the program because it is locally defined. Our model will only trace the actions (reads and writes) of the non-local array `a0[]`.

We will first generate the model for $n = 2$, i.e. an array of only 2 elements, in order to generate a small enough model which we can display and discuss. The type of stored date will be **`int`**`%2`, i.e. 3 distinct values. The types of index `i` is **`int`**`%4`, i.e. one more than the size of the array, and the size of boolean flag `flag` is **`int`**`%2`. The resulting model is as in Figure 3 on the following page. The notation `a0@`k is interpreted as “element k of array `a0`.”

The dynamics of the program are well illustrated by the finite model: the left-side half of the model shows all 9 possible combinations of values being read from non-local array `a0`, while the right-side half writes out the same values back into the array, in the proper order. The local state, array `a`, is internalized and the assignments to it are not reflected in the model of the program.

A slightly different model results if we change the abstraction scheme of the elements of the array to **`int`**`$1`, i.e. the set {`0, -Infty, +Infty, Nan` }, as in Figure 4 on the next page.

The dynamics of the program generated by this abstraction scheme reflect the fact that abstract integers introduce non-determinism in the program. For example, consider State 14, which

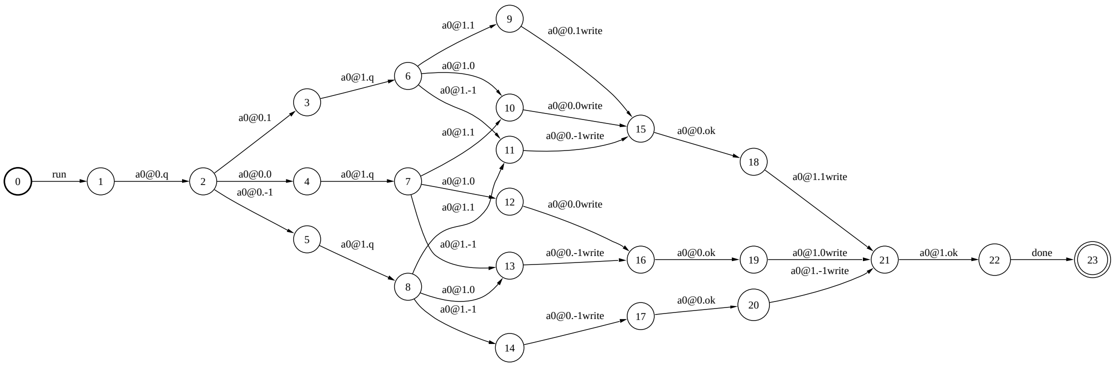

Figure 3: *Bubble-sort* model

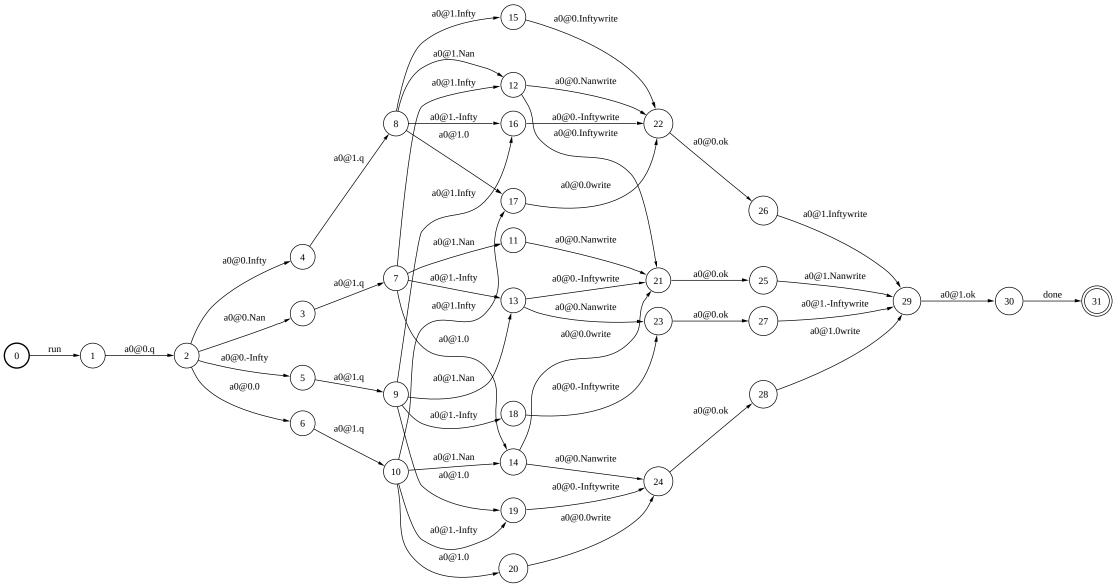

Figure 4: *Bubble-sort* model: abstract integers

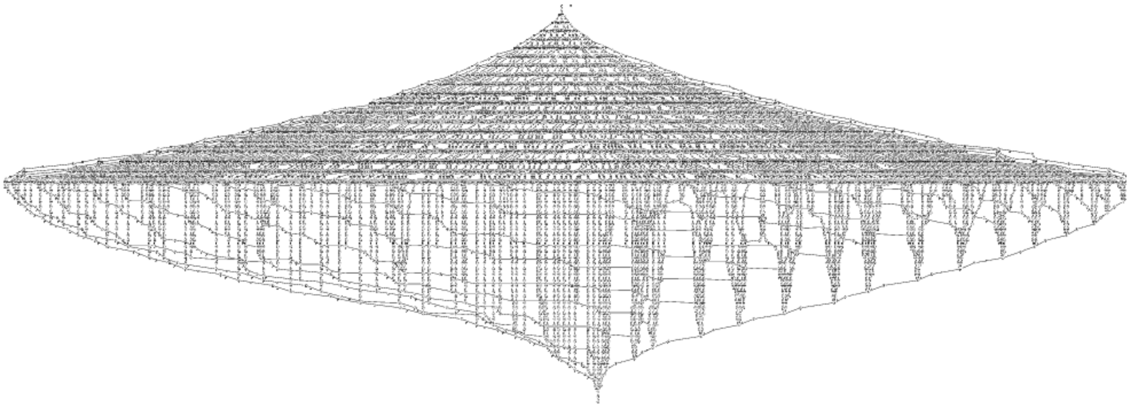

Figure 5: *Bubble-sort* model: 20 elements

is reached in the model if values `0, Nan` are read. The undetermined value (`Nan`) cannot be deterministically compared with 0, therefore there are two distinct values the matrix `a0` can assume: `[0, Nan]` or `[Nan, 0]`; but this is acceptable.

Increases in the array lead to (asymptotically exponential) increases in the time and space of the verification algorithm. On our development machine (SunBlade 100, 2GB RAM), the duration of the generation of the model as a function of n was:

- $n = 2$: 5 minutes
- $n = 5$: 10 minutes
- $n = 10$: 15 minutes
- $n = 20$: 4 hours
- $n = 25$: 10 hours
- $n = 30$: the computation failed.

Figure 5 gives a snapshot of the model for $n = 20$.

Obviously, a model the size of the one in Figure 5 is not meant to be verified manually. But since the output is a generic FS machine, it can be subjected to standard FS-based model checking tools.

Something quite remarkable about the model in Figure 5 is its very compact size. An array of 20 (3-valued) elements can represent 3^{20} distinct states, i.e. approximately 3.5 billion states. This is a vast memory space, beyond the range of tools much more sophisticated than ours. Our tool cannot only handle such a program, but it produces a *complete* model of it.

The key observation is the following: the fact that the state of the array is *internalized* and only a purely behavioural, observationally fully abstract model is presented leads to significant savings in required memory space. In fact, the model in Figure 5 has only circa 5,500 states. So, even though the algorithms we use are generic, the fact that we use a model at a maximum level of abstraction which internalizes the details of stateful behaviour leads to major improvements in efficiency. It is interesting to contrast this kind of abstraction, which comes for free with our fully abstract model, with other, more syntactic, abstraction techniques such as slicing [HDZ00].

In fact, since all the internal state changes are abstracted away, the models above are *extensional* models of sorting. All (correct) sorting program will have models isomorphic to the ones in the figure, subject to the same abstraction schemes. It follows that a straightforward verification method for a sorting program is simply comparing its model with a given, certified correct, reference sorting program serving as an (executable) specification.

Verification Technique 1 *Use a certifiedly correct implementation of an algorithm as a specification. Another algorithm implementation is* equivalent *if and only if the two implementations have isomorphic models. Two equivalent implementation are substitutable in any context. Therefore, if a program has a model equal to that of correct program it can be considered correct itself.*

4.2.1 Modeling out-of-bounds errors

One of the main problems of program manipulating arrays is the possibility of out-of-bounds errors. We handle this error by the following convention: we assume that there exist a system-defined function called `err_oub()` which handles such errors. We need not define this function, only assume its existence. Whenever the program attempts to access elements out of the bounds of the array the `err_oub()` function will be called. Keeping in mind that the traces in our system represent completed computations, we still need to provide a meaningful semantics of out-of-bounds elements, other than abnormal termination. All abnormally terminating behaviour, just like non-termination, is equated in our model therefore such an interpretation, albeit sensible, would lead to errors which are difficult to spot.

The semantics for out-of-bounds we use is the following: call the error-handling function `err_oub()` then access the 0th element of the array. This is convenient because all arrays have at least one element. **N.B.**: the user must realize that whenever the function `err_oub()` is invoked an array-out-of-bounds error has occurred, and all traces of the program involving it are erroneous.

Consider the following potential error in the implementation of *bubble-sort*: the index `i` in the inner loop ranges to n instead of $n-1$ (Figure 2 on page 7, line 9), causing out-of-bounds errors. The resulting model, using the same abstraction scheme as in Figure 3 on page 8, is given in Figure 6 on the following page; all transitions involve out-of-bounds errors.

Verification Technique 2 *If the model of a program (fragment) contains any actions of the* `err_oub` *function then it has array out-of-bounds errors. Any trace of the model involving such actions involve run-time errors.*

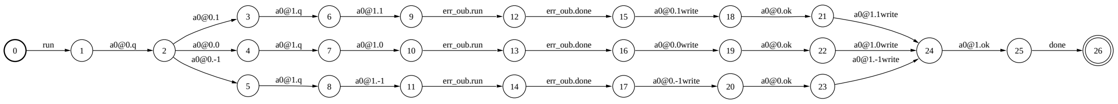

Figure 6: *Bubble-sort* with out of bounds errors

```
int%2 msg(); int%2 send(int%2);
int%2 ack (); void error (); void success () |-
   int%2 v = 0;
   int%2 err = 0;
   while (v > -1 || err == 1) {
      int%2 ok = 0;
      v = msg;
      send (v);
      ok = ack ();
      if (ok) { success (); err = 0; }
      else {
         send (v);  ok = ack ();
         if (ok) { success (); err = 0; }
         else { error (); err = 1; }
         }
      }
:void.
```

Figure 7: Retransmission protocol

4.3 Protocol implementations

Model-checking techniques have been applied very successfully to the verification of network protocols. Network protocols are ideally suited for model-checking because they tend to involve little processing of data, have complex control structures and are compact, stand-alone programs. However, *software implementations* of network protocols are not as easy to verify because of the typical software model-checking issues, discussed in the introduction.

In this section we will see how using the game-based tool, and especially the fact that it can model program fragments with undefined, nonlocal, features, we can generate models of protocol implementations. The example we use is a simple retransmission protocol. As before, we choose a simpler (but still non-trivial) example only for the sake of presentation.

The protocol is the following:

1. get some data from the program, send it and receive an acknowledgement;
2. if the acknowledgement is not positive re-transmit and receive another acknowledgement;
3. if it is still negative report the error to the program;
4. continue until the program sends the key value -1 which signals the end of the protocol;
5. the last value acknowledged in the protocol must be -1.

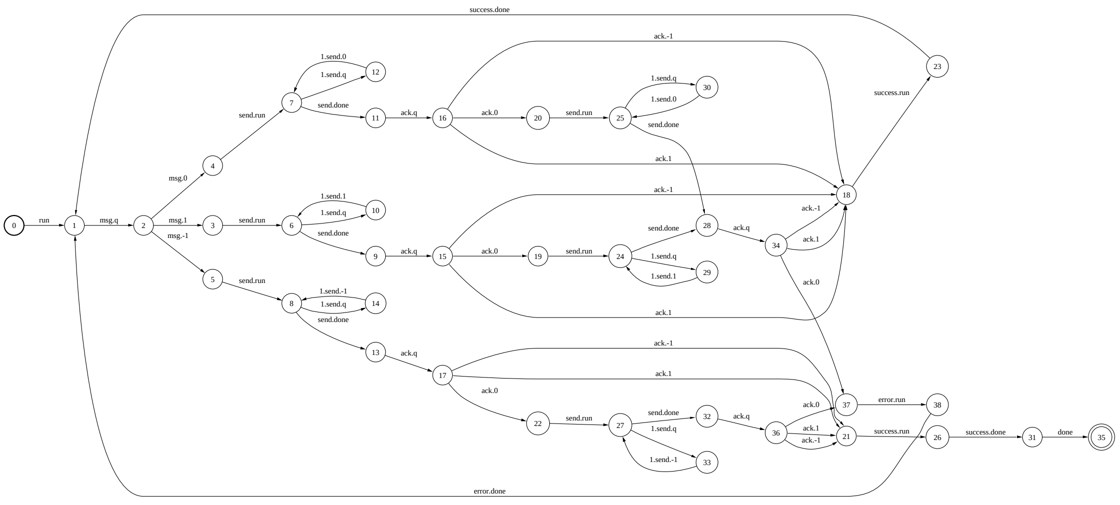

Figure 8: Retransmission protocol model

The implementation is given in Figure 7 on the page before.

The implementation we will model abstracts away from the details of how the protocol communicates with the network (undefined functions `send()` and `ack()`) and with the rest of the program (undefined functions `msg()`, `success()` and `error()`). We only want to model the part of the program which implements the protocol logic. Since our model is compositional, the model we generate will remain valid whatever the concrete implementations of the functions mentioned above is.

The model generated is shown in Figure 8. The action of the non-local undefined functions can be seen in the transitions of the automaton, which can be now analyzed using standard formalisms and tools (for example temporal logics). For example one can easily check, using temporal logics, that the protocol is not guaranteed to terminate, for example if the `ack()` function only returns the value 0.

Verification Technique 3 *The model of a protocol implementation is a FS machine which has the same temporal properties as the protocol itself. Once the model is generated, these properties can be verified in the standard way using (third-party) temporal model-checking tools.*

4.4 ADT implementations

Perhaps the most striking feature of the tool presented here is the ability to model abstract data type (ADT) implementations. We have already hinted at this in the “warm-up” example of the simple switch. In Figure 9 on the next page we show the implementation of a realistic ADT, a fixed-size stack of maximum size m, implemented using an array (`buffer`).

We model the stack module using an undefined function called `CONTEXT` which takes as arguments the interface of the stack. We also assume the existence of two user-defined functions, `empty` and `overflow`, handling `pop` on the empty stack, respectively `push` on a full stack. As before, the implementation details of the error-handling functions can be abstracted without any problems.

The undefined `CONTEXT` stands for a model of all possible usages of the stack. In Figure 10 on page 15 we show the model generated using unary stored data and constant $m = 2$. We have also renamed actions `1.CONTEXT` to `PUSH` and `2.CONTEXT` to `POP`, to reflect the actual actions which we wish to illustrate. Using unary stored data, i.e. only value 0, means that we are only concerned with the *dynamic* properties of the stack, its growth and shrinkage at run-time.

Without going into temporal logic formalizations, we can easily see the truth of propositions such as:

- a `PUSH` followed by a `POP` brings a non-full stack back to the same state;
- $m + 1$ consecutive `PUSH` actions generate at least on `overflow` error;
- `pop` in the initial state generate `empty` errors, etc.

Other key properties, such as the absence of array out-of-bounds run-time errors can be easily observed.

Meaningful stored data and larger array sizes make it unreasonable to examine the model manually, e.g. Figure 11 on page 16 shows the in-context behaviour of a 2-element stack storing ternary values. A model for a 10-element stack of ternary data has over 550,000 states.

Verification Technique 4 *To model an ADT, pass the interface of the ADT to an* undefined *function, which will model a "most general" context for the ADT. Once the model is generated, its properties can be checked using generic (third-party) model-checking tools.*

---

5 Code-level specifications

The emphasis of this work is on *model generation* for program fragments. This is where the main challenge of software model-checking lies. We have said, several times, that the FS model produced by the tool can be investigated using standard FS model checking formalisms and tools.

However, the model-generation tool presented here can function as a model-checking tool in its own right. It does not provide direct support for temporal properties, the formalism typically associated with model-checking, but it provides support for validation of assertions, specifications and equations. These are perhaps properties that programmers are even more interested in checking, because they are not properties *of the model*, but *of the program*.

The support provided for assertions is simple but effective: an assertion can be construed as a boolean function taking as argument a boolean function. We can, in general, define assertions as:

```
void empty ();
void overflow ();
int n;
void CONTEXT (void(), int())
|-
      int buffer[m];
      int crt = 0;
      int isempty () { crt == 0; }
      int isfull () { crt == m; }
      void push (int x)
       {
          int temp = x;
          if ( isfull () )
            overflow ();
          else {
            buffer[crt] = temp;
            crt = crt + 1;
          }
       }
      void pop ()
       {
          if ( isempty() ) {
             empty ();
             return 0;
             }
          else {
             crt = crt - 1;
             return buffer[crt - 1];
             }
       }
      CONTEXT(push (n), pop)
: void.
```

Figure 9: A stack module

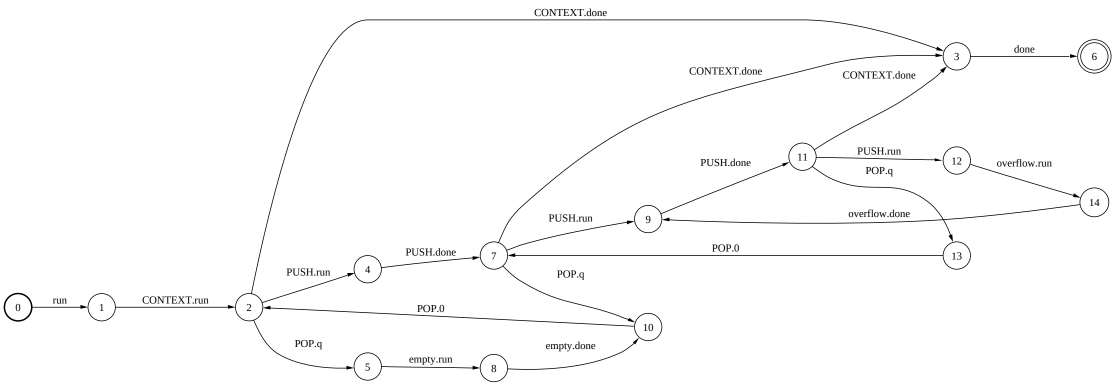

Figure 10: The stack module used in context

```
void assert( int a() ) {
  if ( a() ) while (1); else ;
}
```

To understand how this works, we recall that the model generated by the tool represents successfully completed traces. So, if in a program we call the function `assert` of some expression the program will successfully complete only for those values of the expression which fail the assertion: in other words, the resulting model is a *counter-example* of the assertion!

Consider the simple program in Figure 12 on page 17 which swaps the values of two variables without using a temporary variable. We added the assertion, implemented as above, that upon execution `x` will hold the initial value of `y` and vice-versa.

If we use a data-abstraction scheme based on congruence arithmetic, the model of the program below is empty, meaning that the assertion is indeed true. But if we use abstract finite integers (e.g. `$2`) the resulting model is non-empty, as in Figure 13 on page 17. Not unexpectedly, if the `init` initialization function returns infinite values or `Nan` the assertion fails, because such values do not compare. However, even if we remove these irrelevant failures from the model we still have a non-empty model, as in Figure 14 on page 18.

The conclusion is that for all inputs in which `y` is not assigned initial value 0 the program actually fails — in the presence of overflow. Indeed, if we look back at our program it is obvious that in the presence of overflow the program cannot work because the addition and subtraction operations involved may overflow.

The program only works with congruence or "wrap-around" arithmetic, this is why in C or Java this program is correct. Also, this also shows that the importance of the data-abstraction scheme that is chosen.

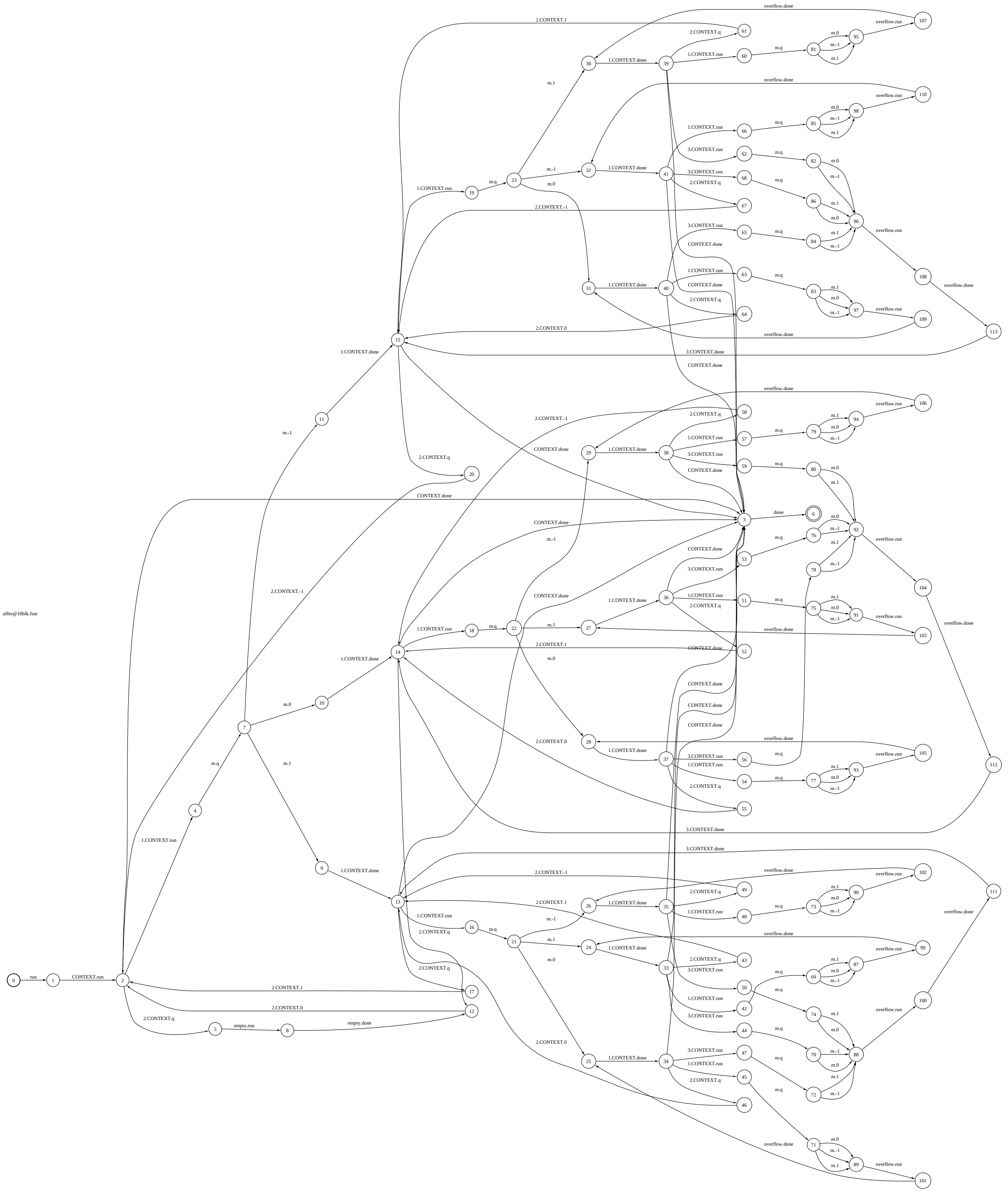

Figure 11: A larger stack model

```
int init() |-
  int x = init();
  int y = init();
  int x0 = x;
  int y0 = y;

  x = x - y;
  y = x + y;
  x = x - y;
  assert( x == y0 && y == x0 )
: void.
```

Figure 12: Using assertions

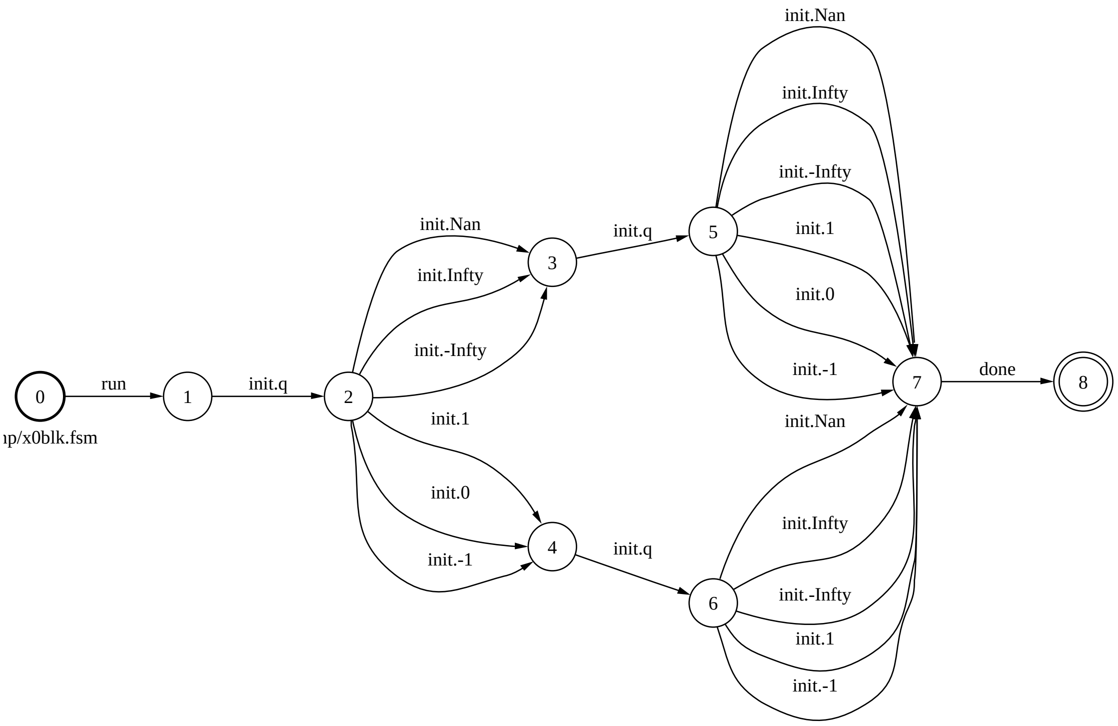

Figure 13: Failures of the *swap* program

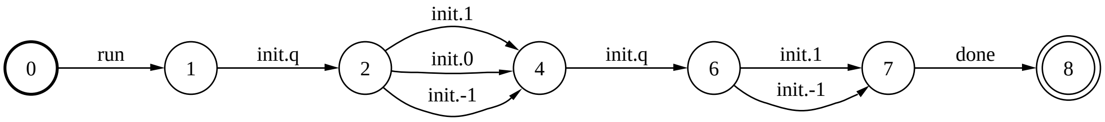

Figure 14: Relevant failures of the *swap* program

Verification Technique 5 *Assertions can be defined directly in the programming language, as seen above. The model of a program (fragment) with such assertions or specifications, if non-empty, represents a* counterexample trace*, a trace for which the assertion fails.*

Finally, using the same assertion mechanism we can check equational properties or invariants of ADTs. For example, going back to the stack example we may want to check that indeed it is the case that the last value pushed onto the stack is the value at the top of the stack. We do this by using the following context, replacing the one used on line 36 of Figure 9 on page 14:

```
int check() {
  int x = m;
  push (x);
  return pop () == x;
  }
CONTEXT(
   push (m),
   pop,
   assert( check() )) : void.
```

This means that in any context, at any time, if we evaluate the function `check` implementing the equational property that interests us the assertion should succeed. Otherwise, it means that the equational property is not actually valid.

The resulting model is not empty, but the non-emptiness of the `CONTEXT` model does not entail, as in the previous case, that there are errors: many non-empty traces are simply traces in which the context does not evaluate the assertion. We need to extract a diagnostic from the equational failures model.

This can be easily achieved using some manipulation of the FS machine representation of the failures model. We know that any traces of the `assert` function represent error traces. Thus, since the assertion is passed as the 3rd argument to the `CONTEXT`, it follows that any traces involving actions `3.CONTEXT.*` are error traces. The first step in extracting the diagnostic model is to *intersect* the FS model with the FS representation of the regular language

$$\mathcal{L} \cdot \texttt{3.CONTEXT.run} \cdot \mathcal{L} \cap \mathcal{L} \cdot \texttt{3.CONTEXT.done} \cdot \mathcal{L},$$

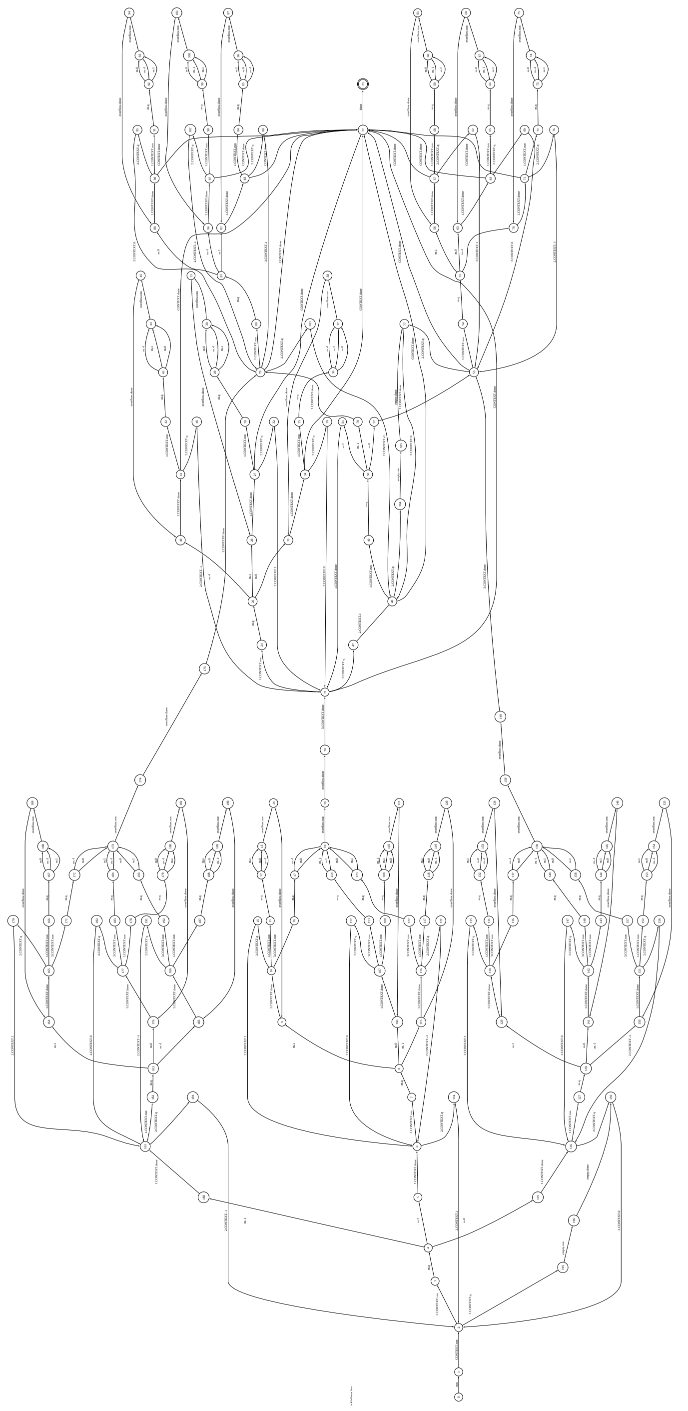

Figure 15: Equational failures of the stack module

```
run
CONTEXT.run
1.CONTEXT.run
m.q
m.1
1.CONTEXT.done
1.CONTEXT.run
m.q
m.1
1.CONTEXT.done
1.CONTEXT.run
m.q
m.1
1.CONTEXT.done
3.CONTEXT.run
m.q
m.0
overflow.run
overflow.done
3.CONTEXT.done
CONTEXT.done
done
```

Figure 16: Shortest trace of the relevant stack failure model

where $\mathcal{L}$ is the language of all possible traces. The result of this is a *relevant failure model* which includes all and only possible traces with errors. Even so, the model is quite large to use as a basis for practical diagnosis (Figure 15 on the preceding page). For practical diagnosis we extract the *shortest trace* in the relevant failure model (Figure 16).

We recollect that `1.CONTEXT` represents a `push` action. So a possible error is caused by pushing 3 times the value 1 onto the 2-element stack. Indeed, if the stack is already full, pushing a new element will be ignored, according to our implementation, and the overflow function handler is called. The problem is easily remedied by including a check of fullness in the equation, using the `isfull` function of the stack module.

The extraction of the shortest trace of the relevant failure model is automated, and part of the tool.

Verification Technique 6 *To verify equational properties or invariants of an ADT write a function implementing the assertion of the equation, respectively the invariant, and pass it as an argument to an undefined context. Extract the* shortest path *in the* relevant failures model *as an effective diagnostic of the error.*

---

6 Conclusion and further work

The proof-of-concept implementation of the game-based modeling tool has yielded very encouraging results. Even using only off-the-shelf, generic implementations of FS machine algorithms for the back-end of our tool we had a tool powerful enough to model complex programs with very large (internal) state spaces. The models generated by the tool are abstract and generic, compatible with other model-checking tools. The single most important advantage of our approach is our ability to verify program fragments and guarantee that the verified properties will remain true when the program fragment is used in any context.

We show how to model three distinct and very important classes of programs: algorithms, protocols and ADTs. We also show a simple and effective way to use the modeling tool as a model-checking tool by introducing code-level assertions, equational properties and invariants.

In conclusion, even though the implementation of our tool is not sophisticated, its simplicity and generality, as well as its user-friendliness make it compare well against other, established and much more sophisticated, software model-checking tools.

There are several important opportunities for further developing both the tool and pursuing related research. Planned developments for the tool, already supported by existing theory, include support for the model-checking friendly specification language of [Ghi02], and translating traces back into source code (to better identify sources of failures). A separate version of the tool supporting call-by-value programming languages would be useful because it would model a substantial fragment of the language C.

Theoretical work, of an applied nature, that would use the remarkable data-independence results of [Laz99] would be helpful because it gives a guarantee that certain properties verified on instances of a program generalize. Also, adapting game models of control [Lai97] and concurrency [Lai01] to our framework seems possible, albeit not immediate.

Game-based models of less restricted programming language fragments, allowing higher-order procedures [Ong02] or some forms of recursion require at the back end machinery more powerful than FS systems, such as pushdown automata. The development and implementation of model-checking algorithms based on such models is in early stages but looks very promising.

Acknowledgements

This research is part of the *Algorithmic game semantics* project at University of Oxford, led by Samson Abramsky and Luke Ong. The author also thanks Bob Tennent for some important suggestions.

References

[AJ92] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. In *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, New Delhi, 1992. Springer-Verlag. Also Imperial College Report DoC 92/24.

[AM96] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. volume 2, chapter 20, pages 297–329. 1996. Published also as Chapter 20 of [OT97].

[AMJ94] S. Abramsky, P. Malacaria, and R. Jagadeesan. *Full Abstraction for PCF*, volume 789 of *Lecture Notes in Computer Science*, pages 1–59. Springer-Verlag, April 1994.

[atta] AT&T FSM Library^{TM} – general-purpose finite-state machine software tools. `http://www.research.att.com/sw/tools/fsm/`.

[attb] AT&T Graphviz - open source graph drawing software. `http://www.research.att.com/sw/tools/graphviz/`.

[Bla92] A. Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56:183–220, 1992. Special Volume dedicated to the memory of John Myhill.

[CGJ98] C. Colby, P. Godefroid, and L. Jagadeesan. Automatically closing open reactive programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 345–357, Montreal, Canada, June 1998.

[Cou96] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

[Ghi01] D. R. Ghica. Regular language semantics for a call-by-value programming language. In *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science, pages 85–98, Aarhus, Denmark, May 2001. Elsevier.

[Ghi02] D. R. Ghica. *A Games-based Foundation for Compositional Software Model Checking.* PhD thesis, Queen's University School of Computing, Kingston, Ontario, Canada, November 2002. Also available as Oxford University Computing Laboratory Research Report RR-02-13.

[GM00] D. R. Ghica and G. McCusker. Reasoning about Idealized ALGOL using regular languages. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming ICALP 2000*, volume 1853 of *LNCS*, pages 103–116. Springer-Verlag, 2000.

[HDZ00] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, December 2000.

[HO00] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163(8), December 2000.

[Kur97] R. P. Kurshan. Formal verification in a commercial setting. In *Proceedings of the Design Automation Conference*, pages 258–262, Anaheim, California, June 1997.

[Lai97] J. Laird. Full abstraction for functional languages with control. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 58–67, Warsaw, Poland, 29 June–2 July 1997. IEEE Computer Society Press.

[Lai01] J. Laird. A games semantics for idealized CSP. In *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics*, Electronic notes in Theoretical Computer Science, pages 157–176, Aarhus, Denmark, May 2001. Elsevier.

[Laz99] R. S. Lazic. *A Semantic Study of Data Independence with Applications to Model Checking.* PhD thesis, University of Oxford, 1999.

[Nic94] H. Nickau. Hereditarily sequential functionals. *Lecture Notes in Computer Science*, 813, 1994.

[Ong02] C.-H. L. Ong. Observational equivalence of third-order Idealized Algol is decidable. In *Proceedings of IEEE Symposium on Logic in Computer Science, 2002*, pages 245–256, July 2002.

[OT97] P. W. O'Hearn and R. D. Tennent, editors. ALGOL-*like Languages*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1997. Two volumes.