Model Driven Code Checking

Gerard J. Holzmann, Rajeev Joshi, Alex Groce

Laboratory for Reliable Software
Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109, USA

Abstract. Model checkers were originally developed to support the formal verification of high-level design models of distributed system designs. Over the years, they have become unmatched in precision and performance in this domain. Research in model checking has meanwhile moved towards methods that allow us to reason also about *implementation* level artifacts (e.g., software code) directly, instead of hand-crafted representations of those artifacts. This does not mean that there is no longer a place for the use of high-level models, but it does mean that such models are used in a different way today. In the approach that we describe here, high-level models are used to represent the environment for which the code is to be verified, but not the application itself. The code of the application is now executed *as is* by the model checker, while using powerful forms of abstraction on-the-fly to build the abstract state space that guides the verification process. This model-driven code checking method allows us to verify implementation level code efficiently for high-level safety and liveness properties. In this paper, we give an overview of the methodology that supports this new paradigm of code verification.

Introduction

An example of a traditional approach to model checking is shown in Figure 1. The figure shows a fragment of a SPIN model to verify some standard file system routines. A process called *main*, written in SPIN's input language PROMELA, initializes some data structures (not shown here) and then enters a loop, randomly attempting to create either files or directories with calls to inline functions that model the relevant behavior of each function. An inline function `choose` specified elsewhere, is used to non-deterministically select a candidate for creation as either a file or a directory.

Many of the smaller details of this model are omitted, and are not of primary interest to us here. In this model, for instance, we represent file names, which are strings in real-life, with abstract numbers. A verification model of this type is typically a high-level abstraction of implementation level code. The abstraction serves clarity and supports the capability to verify system designs free from implementation-level detail. But once the design is verified it always leaves some doubt if the implementation of the design actually conforms to the high-level model. This uncertainty has inspired much of the recent work on the direct verification of concrete implementation level code with the same model checking tools. Still, while doing so, we do not want to give up the ability to define (and benefit from) powerful abstractions. The objective of this paper is to describe a method that was built around the SPIN model checker that provides this critical capability.

Model-Driven Verification

SPIN version 4.0 introduced a small number of primitives that support the embedding of native C code fragments into a PROMELA verification models. The embedded code merely defines state transitions, so the model checking algorithms themselves are not affected by this extension. The model in Figure 2 illustrates how we can use the embedded C code fragments to refine our example from Figure 1. It is now possible to embed the actual file system code that implements the functionality to be verified into the verification model, allowing the model checker to execute this code directly. The initialization routine is now used to initialize a global array with sample pathnames to a fixed range of names, from which the model checker will randomly choose entries. Note that pathnames are real strings now, since we are no longer restricted to the use of only the high-level data types that PROMELA supports.

```
bool exist[N];
bool parent[N];

inline mkdir_spec(d) {
       if
       :: exist[d] -> errno = EEXIST
       :: else ->
              if
              :: !exist[parent[d]] -> errno = ENOENT
              ::  else -> exist[d] = true
              fi
       fi
}

active proctype main() {
       init();
       do
       :: choose(d)-> mkdir_spec(d)
       :: choose(f)-> creat_spec(f)
       od
}
```

Fig. 1. The Traditional Approach to Model Checking

In this example we have used two of the five primitives for supporting the verification of models with embedded C code. The first, called *c_decl*, allows us to declare C functions or data objects that we want to refer to elsewhere in the model. The second, called *c_code*, is used to embed an arbitrary fragment of C code as a state transition into the model. The execution of a C code fragment is always enabled, and it is considered to be a single atomic action.

Again, the details of the revised model in Figure 2 are unimportant, but the potential for model checking applications that the support for embedded code provides is significant. In some sense, the role of the verification model, i.e., the part that is written in PROMELA, changes from that of defining *behavior* to that of acting as non-deterministic *drivers* for externally defined behavior. The behavior itself can now be defined by the implementation. We still use high-level models in this framework, but now these models are used to define the environment for the code to be verified, and not the code itself.

```
c_decl {
          char *path[N];
extern int mkdir(const char *path);
          extern int creat(const char *path);
}

active proctype main() {
      c_code { init(); };
      do
      :: choose(d) -> c_code { mkdir(path[d]); }
      :: choose(f) -> c_code { creat(path[f]); }
      od
}
```

Fig. 2. A Revised Model with Embedded C Code.

```
c_decl {
       extern void canonize(char *dst, const char *src);
       extern Bool is_canonical(const char *str);
       char D[10];
};

byte S[10];
inline pick(i) {
       if
       :: S[i] = 'a' :: S[i] = 'b' … S[i] = 'z'
       :: S[i] = '/' :: S[i] = '.'
       fi
}

active proctype main() {
       int i;
       do
       :: i < 9 -> pick(i); i++
       :: S[i] = '\0'; break /* allow for short paths as well */
       od;
       c_code { canonize(D, now.S); };
       assert c_expr { is_canonical(S) }
}
```

Fig. 3. Checking the Canonize Function with SPIN

Getting Started with Model-Driven Code

As a small application of this model-driven verification methodology, consider the verification of a procedure that converts UNIX pathnames into a canonical form, removing simple redundancies. This function could, for instance, be declared with the C prototype:

```
void canonize(char *dst, const char *src);
```

Given a pathname of the form `"/etc//tmp/..///issue.net"`, this procedure should then produce the cleaned up version of the path `"/etc/issue.net"`. To test the result, we could write a simple Boolean test function:

```
Bool is_canonical(const char *str);
```

This checker can be called after each call to `canonize` as a minimal check that the result conforms to our expectations. A SPIN model to perform a verification of the *canonize* function is shown in Figure 3.

In Figure 3, in addition to the *c_decl* and *c_code* primitives that we discussed before, we have used a *c_expr* primitive. A *c_expr* primitive allows us to execute an arbitrary fragment of C code to obtain a Boolean value. This value can be used to define a conditionally executable statement, but it can also be used (as in this case) as a mere expression, which is used here as the argument to a PROMELA assertion. The model checker will assume that the evaluation of a *c_expr* primitive can be done without any side-effect on the system state. It is considered a modeling error if this condition is not met. The model checker performs some modest checks to make sure that the condition is met. The general problem of checking arbitrary C code accurately for all possible side-effects is of course quite hard, and no attempt to solve it fully is made here.

The test driver is used to randomly generate pathnames, which are then canonized and checked. We can of course rely on the SPIN model checking engine to avoid repeatedly testing the same pathnames. It will also be clear that within this framework we can check any temporal property, since we have the full power of the model checker available to us. This way of using SPIN then puts us closer to a standard method for

randomized software testing, but without sacrificing any of the verification options that we traditionally associate with the use of model checkers.

We will explain in more detail shortly how abstraction functions can be used within this framework, but for the moment the following observation will suffice. Note that the model specified in Figure 3 does contain some unnecessarily redundancies. The canonize function, for instance, needs to check only for the presence of the special characters '/', and '.'. Without loss of generality, we can therefore restrict the non-deterministic choice of pathnames to just three characters: e.g., 'a', '/', and '.'. Since no state other than the PROMELA byte array `S` is relevant to the verification process, the model checker will have no trouble keeping track of the *state* of the system in this case, even though part of the relevant behavior is specified in C code.

More complex applications often maintain a significant amount of additional state information that is not automatically visible to or known to the model checker. This additional state information normally consists of the values of data objects that are declared inside the application code. In these cases, we have to do some more work to make the model checking process work correctly. To do so, we use a predefined mechanism for pointing the model checker at the additional state information with so-called *c_track* primitives.

Tracking External State

To illustrate the verification of models with externally defined state information, we discuss a slightly more detailed C application. Our example application is used to build and maintain a binary tree with integer values stored at the leafs. The contents of the tree can trivially be represented in an array of integers, for instance by listing the elements in the tree in breadth-first order. Figure 4 illustrates this encoding for a small example tree.

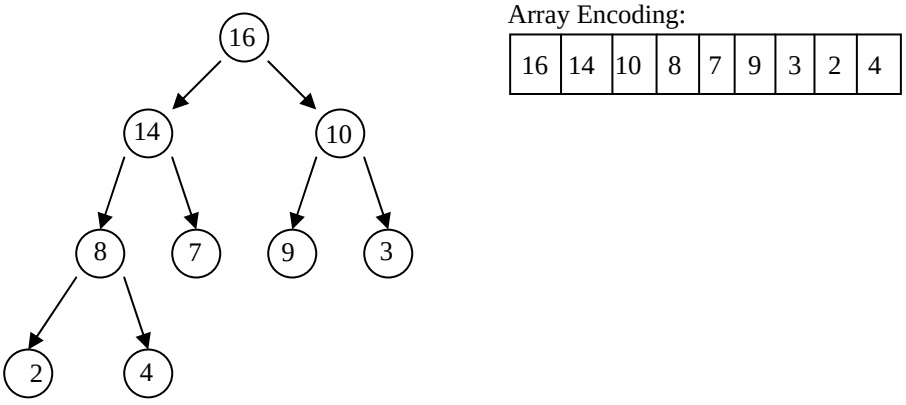

Fig. 4. Embedded C code for a Binary Tree

If we want to verify the C code that is used to construct and maintain trees like this, it will be apparent that the contents of the array of integers now becomes an essential part of the system state, even though it is defined externally to the SPIN model. All the model checker needs to know, though, is where in memory the additional data is stored and how large it is. It does not need to know anything more about how the data is accessed. We can transfer this information to the model checker with the help of PROMELA *c_track* primitives, as illustrated in Figure 5. The *c_track* statements are used in their simplest form here, where they simply give the address of the external data in a first argument and the size of that data in a second argument. Later we will see that we can use an optional third argument to support the use of powerful abstraction functions in the verification process, but we will first discuss the more basic usage.

When the model in Figure 5 is verified, the model checker uses *c_expr* conditionals to perform a non-deterministic selection of all possible tree manipulations, and it checks the integrity of the tree with an

```
c_decl {
       extern void  setup(void);
       extern int   insert_el(int);     /* update function */
       extern int   extract_max(void);  /* update function */
       extern Bool  check_tree(void);

       int pos;     /* position in array */
       int B[100];  /* the array */
};

c_track “&pos” “sizeof(int)”;
c_track “B”    “sizeof(int) * 100”;

inline pick(v, A, B) {
       atomic {
              v = A;
              do
              :: v < B -> v++
              :: break     /* pick any value in the range A..B */
              od
       }
}

active proctype main() {
       c_code { setup(); };      /* initialize tree; pos = 0 */
       do
       :: if
          :: c_expr { pos < 100 } ->    /* if array not full */
              pick(val, 1, 101);        /* insert new element */
              c_code { insert_el(val); }
          :: c_expr { pos > 0 } ->      /* if not empty */
              c_code { int v = extract_max(); }/* find max */
          fi ;
          assert c_expr { check_tree() }
       od
}
```

Fig. 5. SPIN Model for Verifying the Binary Tree.

externally defined C function called *check_tree()*. All externally declared C data and code is compiled and linked to the model checking code before the model checking process starts. After each execution step that involves the execution of embedded C code, the model checker will capture the new state of all tracked data, and store it in the state vector. Similarly, when the model checker backtracks to a previous state, it will restore the state of the external data to the proper values, so that the integrity of the model checking process is preserved.

In the example from Figure 5, the C function *check_tree()* can call SPIN’s builtin routine *uerror(“error”)* in its code to trigger the creation of a counter-example trail when the built-in test fails. The model checker will then write a standard counter-example trail-file that can be replayed later.

There is one important change in the replay of counter-examples for models with embedded C code though. Since SPIN itself does not attempt to interpret anything contained inside c_decl, c_code, or c_expr statements (it is simply passed on to the C compiler) it would not be able to reproduce the effect of the execution of those statements on the system state during a replay either. So instead of replaying the counter-examples with the model checker generator *spin*, we replay them with the generated and compiled model checking code contained in *pan*.

More importantly, though, the framework we have described for checking *safety* properties of C code implementations with the model checker extends also to the verification of *liveness* properties (including all properties that can be specified in linear temporal logic). Note carefully that the use of embedded C code to define state transitions is invisible to the model checking algorithm itself. To the model checker, any state

transition performed through a *c_code* or *c_expr* primitive is no different than a state transition that results from the execution of a native PROMELA statement.
As a final note on the binary tree example, we can observe that the check that is performed on the sanity of the tree data structure need not appear in the model itself, as shown in Figure 5, but can also appear in a standard PROMELA *never* claim, as shown in Figure 6.

```
never {
       do
       :: assert c_expr { check_tree() }
       od
}
```

Fig. 6. Placing the Integrity Check in a Never Claim

The Use of Abstraction

The full reachable state space of an implementation level application is often much larger than that of a high-level model, and more than what can be explored exhaustively by a model checker. This means that the use of abstraction is as important in this domain as it is in the more classic approaches to model checking. Fortunately, there is a simple, and powerful, mechanism to define abstractions that can optimize the state space exploration process, even with the use of embedded C code and externally defined data.

To understand how this works, recall that SPIN performs a standard depth-first search of the reachable state space. The search either advances forward, by placing a new concrete state on its internal search stack, or it reverses by restoring the system to an earlier state of the current execution path, popped from the same internal search stack.

At each new state, the model checker generates successor states that can extend the execution path. For each such successor state, the model checker first checks if it matches a previously visited state that is stored in the state space. If the newly generated state is not found there, it is added both to the state space and to the search stack, and the search proceeds with that state. If the successor state was previously visited, the search does not proceed with this state but continues with the exploration of alternative successor states. Once all successor states have been explored, the search reverses to a previous state in the search path, restores the state of the system to the earlier state stored on the search stack, and continues from there. This process continues until one of four things happen.

1. All states have been visited and the search completes normally.
2. A counter-example to a correctness property is found.
3. The maximum time allowed for a search expires.
4. The maximum amount of memory available to store states is exhausted.

In either of the last two cases, the search has to be abandoned, while delivering only a partial result. If the partial search nonetheless succeeded in exploring a larger fraction of feasible system behavior than can be uncovered with standard software testing, the result can still be valuable. If it does not, which in our experience is rare, the exercise will have no redeeming value. Since the use of embedded C code preserves all standard SPIN model checking features, the fraction of the reachable state space that is explored can often be increased significantly by using state compression or bitstate hashing techniques [H04].

For the current discussion, the main thing to note is that the model checker stores states in *two* different data structures, for *two* distinct reasons. The two data structures are the search stack and the state space. The first data structure (the stack) is used to make it possible to restore the state of the system to a previous point in the execution when the search backtracks. The second data structure (the state space) is used to avoid the repeated exploration of the same states. To serve their purpose, the states that are stored on the search stack must always remain in their concrete form, i.e., they cannot be abstracted (unless the abstraction function is reversible that is). The states stored in the state space, on the other hand, need not be concrete at all: they can be represented in any way that is useful to the verification process.

The model checker has freedom in choosing how states are represented in the state space. It can, for instance, use irreversible compression functions and it can use approximation functions (as is done in

bitstate hashing). The main criterion is that we can recognize when a previously seen state is revisited later in the search. This, then, is the right place to use abstraction functions. The user can define an abstraction function to manage how states are stored in the state space, but not for states stored on the search stack.

Before a state is compared against the state space, the state can be abstracted, and compared and stored in abstract form. If the abstract representation of a state matches a previous state, the model checker will conclude that the search has either returned to the same concrete state, or to a state that is equivalent to it under the abstraction that is used. In this way, the model checker can explore an *abstract* representation of the state space, while performing an accurate *concrete* execution of the application. Any counter-example that is found in this search will still have a concrete representation, and can be reproduced as a full concrete counter-example (since the states for the error-trail are available on the search stack).

```
void dfs(State s) {
     push s onto Stack T              # concrete states
     add abstract(s) to StateSpace S  # abstracted states
     for each successor state s' of s
           if abstract(s') is not in StateSpace S
           {     dfs(s')
           }
     pop s from Stack T
}
```

Fig. 7. Pseudo Code for a Depth-First Search, Using a *Concrete* Stack *T* and an *Abstract* StateSpace *S*.

This process is illustrated in the pseudo code shown in Figure 7, using the standard depth-first search as an example. The process is analogous for the *nested*-depth first search algorithm, which indeed means that with this mechanism we can still verify safety *and* liveness properties for implementation level code, also with user-defined abstractions.

In the next subsection we discuss how the user-level abstractions of this type can be defined.

Implementing Abstraction: Matched and Unmatched Data

The use of abstractions in models with embedded C code is currently restricted to tracked data. (I.e., it cannot be used to define an additional layer of abstraction on native PROMELA variables.) To support this, *c_track* primitives can be used in two different ways, using the optional third argument that we hinted at earlier. The optional third argument to a *c_track* primitive consists of the string *Matched* or *UnMatched*. Tracked data that is declared to be *Unmatched* is stored only on the search stack, but not in the state space. Tracked data that is declared to be *Matched* is stored *both* on the search stack and in the state space. If no third argument is given, a *c_track* statement is always interpreted to be *Matched*. The following example can explain how this mechanism allows us to support used-defined abstractions in a fairly straightforward way.

Suppose our application manipulates two integers, named *x* and *y*, and we want to abstract their combined value into a single integer using the function *(x+y)*. We can do this by tracking the concrete values of both variables as *UnMatched* data, while computing the abstraction *(x+y)* after every state transition and storing the result as *Matched* data. This is illustrated in Figure 8.

```
c_decl {
     extern int x, y;    /* defined somewhere in the application */
     int sum;            /* to store abstraction of x and y */
}

c_track “&y”   “sizeof(int)”  “UnMatched”;   /* stored on the stack only */
c_track “&y”   “sizeof(int)”  “UnMatched”;   /* stored on the stack only */
c_track “&sum” “sizeof(int)”  “Matched”;     /* also stored in statespace */

and elsewhere in the model:

c_code { compute(&x, &y); abstract_xy(x,y,&sum); };

with : abstract_xy(int v1, int v2, int *abs) { *abs = v1+v2; }
```

Fig. 8. The Use of Abstractions

The execution of the embedded C code always works with the concrete values for variables *x* and *y*, retrieving earlier values from the search stack when the search engine backtracks. But, the forward moves of the model checker are controlled by abstract values (the value of *sum*) as stored in the state space. The user's obligation in the definition of an abstraction now consists of four relatively simple steps:

1. Defining the abstraction function proper (in this case the function *abstract_xy()*).
2. Adding a *Matched* tracking statement for the abstract state data thus computed.
3. Marking the tracking of all abstracted data (the input to the abstraction function) as *Unmatched*.
4. And finally, adding a call to the abstraction function at the end of every *c_code* block that potentially modifies the tracked, and now *Unmatched*, state information.

Missing C_track Statements

It can sometimes be difficult to determine accurately where all the state information is hiding in a large C application, so that the right number of *c_track* statements can be defined. The question what will happen if a c_track statement is forgotten is not entirely academic. What happens is that the model checker will not be able to accurately restore the system state on backward moves during the search and can start producing counter-examples that cannot be reproduced on replay. If all is well, any trail file, e.g. *model.trail*, with embedded C code that is produced by *pan*, can be reproduced by the same executable with the command:

```
$ ./pan –r model.trail
```

If not all state data is tracked properly, the replay command will fail, with *pan* reporting “transition unexecutable” for some of the execution steps. One solution, though not an attractive one, is to always track *all* externally defined data structures of an application as *Unmatched* data. This can significantly increase the memory use for the search stack, though, and it can slow down the model checking process unnecessarily. In most cases, a careful inspection of relevant data will suffice to find the additional data that must be tracked. One optional way to simplify this process is to use a code instrumentation tool [GJ08], to add checks into the embedded C code for all locations that modify memory (both tracked and untracked). By ruling out locations that are already tracked or that hold no state information, the missing *c_track* statements can usually be located.

In this process, it can be helpful to eliminate from consideration all data that the model checker itself uses. That data includes all information that stored on the model checker's search stack. To simplify such checks, SPIN defines a special purpose global variable called *stack_begin*, which is set to the address of a stack variable when the depth-first search starts. A reference to that variable can be included into the model as follows:

```
c_decl { extern void *stack_begin; }
```

We can now determine if a memory reference is to a stack variable by checking if its address lies between the value of the predefined *stack_begin* and the address of an arbitrary stack variable in the C function that performs the check (which naturally will be near the opposite end of the stack).

Sound Abstractions

Not all user-defined abstractions will necessarily preserve the logical soundness of a model checking run. Generally, we will have to prove this for any given abstraction function. The procedure to do so is fairly standard though. Let ~ be the equivalence relation that is induced by abstraction A, i.e., $s \sim t \Leftrightarrow A(s) = A(t)$. Further, let P be an atomic proposition and let P(s) represent the truth value of P when evaluated in state s. The following two conditions can be shown to be sufficient for the abstraction function A to be logically sound [HJ04].

1. $\forall w,y,z:\ w \sim y \wedge y \rightarrow z \Rightarrow (\exists x:\ w \rightarrow\ x \wedge x \sim z)$
2. $\forall x,y : P(x) \wedge x \sim y \Rightarrow P(y)$.

The first condition says that relation ~ is a bisimulation. States w and y are bisimilar if, whenever there is a transition from y to z, there is a successor x of w such that x and y are also bisimilar. The second condition says that all atomic propositions P are preserved by ~. For details we refer to the treatment given in [HJ04].

Unsound Abstractions

For large applications, even the best abstractions may not be able to reduce the reachable state space sufficiently for a model checking run to be completed within available resource limits (time and memory). In such cases, there can be a role for the use of abstractions that are formally unsound, but that can restrict the verification effort to an area of special interest to the human verifier. Clearly, such abstractions cannot be used to prove the absence of errors, but they can often be used successfully to expose their presence.

An interesting example of an unsound abstraction is the use of a function that tracks basic code and path coverage. The following example illustrates this approach. Our objective in this case is to achieve good path coverage over the application level code, without trying to achieve fully exhaustive coverage of all possible executions. The following statements first declare an array to store a bit vector of values, and an integer variable to index that array.

```
c_decl {
   Bool BV[100]; /* used as a bit vector path */
   int P;        /* used as an index into the array */
};

c_track “BV” “sizeof(Bool) * 100” “Matched”;
c_track “&P” “sizeof(int)” “Matched”;
```

We now instrument the application code by adding an update of the array *BV* at all places in the code that we want to reach, marking, for instance, different branches of key conditional choices in the code. The following code without instrumentation:

```
if (buf == NULL) {
       S1;
} else {
       S2;
}
```

is now changed into:

```
if (buf == NULL) {
       BV[P++] = 0;
       S1;
```

```
} else {
        BV[P++] = 1;
        S2;
}
```

This type of instrumentation can also be automated with the help of a code instrumentation tool, as discussed more fully in [GJ08].

Because the value of vector *BV* is part of the *Matched* state, whenever the contents of *BV* changes, so does the system state, independent of what really happens inside the statements *S1* and *S2*. As an extreme case of this unsound method of abstraction, one can forego matching any other part of the system state. In experiments we have performed, we have noted that the model checker can still achieve very reasonable coverage of the problem domain, while storing only very small amounts of data. Note, though, that if the application has relevant state that must be restored when the model checker backtracks the search, all such data must still be tracked with *Unmatched* tracking statements, to secure the accuracy of the verification process itself. These *Unmatched* tracking statements do not increase the size of the state space, but they will of course increase the amount of memory that is needed to store the concrete search stack.

Application

The generic framework for model-driven verification of application level code that we have described in this paper, as the recent evolution of the SPIN model checker itself, was motivated by a direct need to apply the technology to substantial software applications that are of critical importance to NASA missions.

One such example is the development of a series of modules that support file storage on non-volatile memory for an upcoming mission to Mars, called Mars Science Laboratory [MSL]. The difficulties experienced on previous inter-planetary missions with commercially available flash file systems prompted the development of a new design that could be verified to a significantly higher degree, with the use of model checking techniques. In this case, since the application was designed and implemented by us, we could make sure that it was structured in such a way that the application of the model-driven techniques outlined in this paper is simplified. All key data holding state information, for instance, was placed in a single contiguous area of memory that could be tracked with a small number of *c_track* statements. Where initially some of our verification work was done with high-level pure PROMELA verification models, we later switched to the exclusive use of models which include all module code in embedded C code. The environment in our case models user-behavior, where our “users” are the other software modules that fly on the spacecraft and that need access to the file system.

At the time of writing, we have performed model-driven verifications of modules totaling roughly ten thousand lines of C code, performing mission critical functions. Since we can use the full range of capabilities of the model checker in the verification of this code, we make use of all complementary techniques as well, such as the use of search randomization methods [GJ08], multi-core verification algorithms [HB07], and swarm verification techniques [HJG08].

In Conclusion

The basic method of model-driven software verification that we have described in this paper differs substantially from earlier effort to develop this type of framework, including earlier attempts by the authors based on the FeaVer and Modex tools, e.g. as detailed in [HS99] and [H00]. In the earlier approach, abstractions were defined with the help of lookup tables that were used by a model extractor that converted the targeted C code functions into embedded code in a SPIN verification model. Statements were converted one at a time, while the control-flow structure of the C function was preserved. A benefit of the earlier approach was that it was very fine-grained, allowing the model checker to interleave the execution of multiple processes in a distributed environment in arbitrary ways.

Within the new framework we no longer attempt to extract models from C code, but instead we directly embed the targeted C code unchanged into the verification model, using the new PROMELA primitives that support such embeddings. The model checker now invoked the functions defined in C one function at a

time, without internal interleaving of statements. For the applications that are of primary interest to us at the moment, this method gives us the control that we need. The only non-determinism in these applications comes from the environment, which is captured in traditional PROMELA models, in abstract form. The code itself (e.g., the file system access routines) is executed atomically. The atomicity is a good match for sequential applications, and for concurrent systems it can be guaranteed with the help of semaphores. For applications that violate the atomicity assumption, the current framework does not apply. For a full verification of this more free-style code, the interleaving of statement executions must reach inside the embedded C code functions, which means that they cannot be embedded as atomic units. We still have the option to embed C code one statement at a time, but we have no tool support to facilitate this type of instrumentation. A discussion of a possible extension that can address also those problems within a more general setting can be found in [ZJ08].

One question that is often asked is how large of an application can reliably be verified with the help of a model checker, for instance with the approach we have outlined in this paper. The answer, however, cannot be expressed as a simple upper-bound on the number of lines of code in the application. It is possible to write an application of perhaps five lines of code that cannot be verified mechanically with any known technique, and it is possible to write an application of a million lines of non-branching sequential code that could be verified exhaustively in a fraction of a section. Our ability to verify the correctness properties of an application depends on the computational complexity of that application. A clear metric for complexity for the purposes of model checking is easily found. It is the number of reachable states that must be explored by the model checker, after the application the strongest possible types of abstraction and the strongest possible types of optimization. How many states can an explicit state model checker such as SPIN handle? The default standard partial order reduction algorithm that is used in SPIN can reduce the number of states that must be explored for a full verification by an exponential amount (very similar to the types of reduction obtained in symbolic model checkers with the use of algorithms based on binary decision diagrams),. After all optimizations and reductions have been taken into account, it comes down to a fairly simple calculation. If we have M bytes of memory available, and each unique reachable state takes up S bytes of memory to store, then we cannot store more than M such states before we run out of memory. Similarly, if we are willing to spend T seconds on the verification, and on average it takes E seconds to explore one new reachable state, then we cannot explore more than T/E states before we run out of time. The limit will be the lower of the two numbers. The two numbers are somewhat correlated, because in SPIN the time it takes to explore a state normally grows approximately linearly with the size of the state.

> Consider a system with 128 GByte of memory, and a state size of 128 bytes, the memory limit with a standard exhaustive verification is trivially 10^9 reachable states. If it takes 10^{-6} seconds to explore one state it would take 10^3 seconds to explore these 10^9 states, or roughly 17 minutes. There are many options in the model checker, though, to reduce the amount of space needed per state, often in return for a potential decreases in speed. One verification mode is especially interesting in this context. If bitstate hashing is used, each state would by default be recorded in just three bit-positions in the 128 GByte memory arena. This would allow for up to $3.7\ 10^{11}$ states to be stored in the same amount of memory. At the same speed of exploration of 10^{-6} seconds per state, though, this would now take a more noticeable $3.7\ 10^5$ seconds (over 4 days).

It has long been thought that memory was the real hard limitation to the scope of model checkers. The quick calculation above shows, though, that this is rapidly changing. It will not take too many years before we can expect to see machines with RAM memory sizes in the TerraBytes. Curiously, chip makers have abandoned the quest for ever faster CPUs, so it is quite possible that the anticipated machine with a TerraByte of main memory will run at close to the same speed as the machines we have today. If that is the case, all our model checking attempts will be time-limited. Clearly, we could not afford to wait for months or years for a large bitstate verification run for the extra large statespace sizes that we could theoretically explore then. On the bright side, though, the infrastructure that is available to us all through the internet is increasingly supporting access to high-speed networks of compute and data storage servers. At the time of writing (mid 2008) it is already possible to lease access to large compute farms with hundreds or thousands of CPUs for minutes or hours at a time. We believe that these developments will dramatically change the way we perform software verification. Some initial thoughts on ways to exploit this new network infrastructure are explored, for instance, in [HJG08].

Acknowledgement

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. The work was supported in part by NASA's Exploration Technology Development Program (ETDP) on Reliable Software Engineering.

References

[BBR07] J. Barnat, L. Brim, P. Rockai, Scalable multi-core LTL model checking, Proc. SPIN 2007, pp. 187-203.

[GJ08] A. Groce, and R. Joshi, Extending model checking with dynamic analysis, *Proc. VMCAI 2008*.

[HS99] G.J. Holzmann and M.H. Smith, Software model checking – extracting verification models from source code. In Proc. Formal Methods for Protocol Engineering and Distributed Systems, pp. 481-497, Kluwer Publ., Oct. 1999.

[H00] G.J. Holzmann, Logic verification of ANSI-C code with SPIN, *Proc. SPIN 2000*.

[H04] G.J. Holzmann, The SPIN model checker – primer and reference manual, Addison-Wesley, 2004.

[HJ04] G.J. Holzmann and R. Joshi, Model-driven software verification, *Proc. SPIN 2004*.

[H07] G.J. Holzmann and D. Bosnacki, The design of a multi-core extension of the SPIN model checker, IEEE Transactions on Software Engineering, Vol. 33, No. 10, pp. 659-774, October 2007.

[ZJ08] A. Zaks and R. Joshi, Verifying multi-threaded C programs with Spin, *Proc. 15th Spin Workshop*, UCLA, August 2008.

[HJG08] G.J. Holzmann, R. Joshi, and A. Groce, Tackling large verification problems with Swarms, *Proc. 15th Spin Workshop*, UCLA, August 2008.

[GJ08] A. Groce, and R. Joshi, Random testing and model checking: building a common framework for nondeterministic exploration, *Proc. Sixth International Workshop on Dynamic Analysis*, WODA, Seattle, July 21, 2008.

[MSL] For more information, see: http://mars.jpl.nasa.gov/msl/