Model-Driven Software Verification

Gerard J. Holzmann
Rajeev Joshi

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

Abstract. In the classic approach to logic model checking, software verification requires a manually constructed artifact (the model) to be written in the language that is accepted by the model checker. The construction of such a model typically requires good knowledge of both the application being verified and of the capabilities of the model checker that is used for the verification. Inadequate knowledge of the model checker can limit the scope of verification that can be performed; inadequate knowledge of the application can undermine the validity of the verification experiment itself.

In this paper we explore a different approach to software verification. With this approach, a software application can be included, without substantial change, into a verification test-harness and then verified directly, *while preserving the ability to apply data abstraction techniques*. Only the test-harness is written in the language of the model checker. The test-harness is used to drive the application through all its relevant states, while logical properties on its execution are checked by the model checker. To allow the model checker to track state, and avoid duplicate work, the test-harness includes definitions of all data objects in the application that contain state information.

The main objective of this paper is to introduce a powerful extension of the SPIN model checker that allows the user to directly define data abstractions in the logic verification of application level programs.

1. Introduction

In the classic approach to software verification based on logic model checking techniques, the verification process begins with the manual construction of a high-level model of a source program. The advantage of this approach is that the model can exploit a broad range of abstraction techniques, which can significantly lower the verification complexity. The disadvantage is that the construction of the model requires not only skill in model building, but also a fairly deep understanding of the functioning of the implementation level code that is the target of the verification. Any misunderstanding translates into a loss of accuracy of the model and thereby into a loss of accuracy of the verification. If errors are found in the model checking process, these misunderstandings can often be removed, but if no errors are found the user could erroneously conclude that the application was error free.

In this paper we describe a new verification method that in many cases of practical interest can avoid the need to manually construct a verification model, while still retaining the capability to define powerful abstractions that can be used to reduce

verification complexity. We call this method ‘‘model-driven software verification.’’

In Section 2 we discuss the basic method of using embedded C or C++ code within SPIN verification models, and we discuss a relatively small extension that was introduced in SPIN version 4.1 to support data abstraction on embedded C code. Section 3 contains a discussion of two example applications, Section 4 reviews related work, and Section 5 presents our conclusions.

2. Model Checking with Embedded C Code

SPIN versions 4.0 and later support the inclusion of embedded C or C++ code within verification models [8,9]. A total of five different primitives can be used to connect a verification model to implementation level C code. Some of these primitives serve to define what the *state* of the model is, with optionally some of the state information residing in the application. Other primitives serve to define either conditional or unconditional *state transitions*.

One of the primitives that can be used to define *state* is `c_decl`, which is normally used to introduce the types and names of externally declared C data objects that are referred to in the model. Another primitive of this type is `c_track`, which is used to define which of the data objects that appear in the embedded C code should be considered to hold state information that is relevant to the verification process.

Two other primitives define state *transitions* with the help of C code. The first of these is `c_code`, which can be used to enclose an arbitrary fragment of C code that is used to effect the desired state transition. The last primitive we discuss here is `c_expr`, which can be used to evaluate an arbitrary side-effect free expression in C to compute a Boolean truth value that is then used to determine the executability of the statement itself.

Figure 1 illustrates the use of these four primitives with a small example.

```
c_decl {
        extern float x;
        extern void fiddle(void);
};

c_track "&x" "sizeof(float)";

init {
        do
        :: c_expr { x < 10.0 } -> c_code { fiddle(); }
        :: else -> break
        od
}
```

Fig. 1. Embedded C Code Primitives.

The first statement in this example introduces definitions of an externally declared floating point variable named `x` and an externally declared function named `fiddle()`. Presumably, the variable holds state information we are interested in, and the function defines a state transition. To record the fact that the external variable holds state information that must be tracked by the model checker, the `c_track` primitive is used to provide the model checker with two salient facts: the address of the variable and its size in bytes. The model checker will now instrument the verification engine to copy the current value of variable `x` into the state descriptor after any transition in

which its value could have changed (i.e., after every execution of a `c_code` statement). Similarly, whenever the verifier performs a backtracking step for any statement that could have changed the value of the variable, the code of the verifier is again instrumented to reset the value of `x` to the copy of its previous value that was stored in the state descriptor for the earlier state.

The `init` process in this example model will now repeatedly call the external function `fiddle()` until it sees that the value of floating point variable `x` is less than 10.0, after which it will stop.

Effectively, these extensions allow us to introduce new datatypes in verification models, well beyond what PROMELA supports, and it allows us to define new types of transitions, with SPIN performing the normal model checking process.

It is important to note here that the `c_track` primitive, as used here, supports two separate goals: state tracking and state matching.

- State *tracking* allows us to accurately restore the value of data objects to their previous states when reversing the execution of statements during the depth-first search process.
- State *matching* allows us to recognize when a state is revisited during the search. When a state is revisited, the model checker can immediately backtrack to a previous state to avoid repeating part of the search that cannot yield new results.

We will see shortly that if we modify the way in which state information can be matched, while retaining accurate tracking, we can define powerful abstractions that can significantly reduce the complexity of verifications.

2.1. Tracking without Matching

There are cases where the value of an external data object should be tracked, to allow the model checker to restore the value of these data objects when *backtracking*, but where the data object does not actually hold relevant state information. It could also be that the data object does hold state information, but contains too much detail. In these cases we would benefit from defining abstractions on the data that are used in state *matching* operations, while retaining all details that are necessary to restore state in the application in *tracking* operations.

A relatively small extension that makes it possible to do this was included in SPIN, starting with version 4.1. The extension is to support an additional, and optional, argument to the `c_track` primitive that specifies whether the data object that is referred to should be matched in the statespace. There are two versions:

```
c_track "&x" "sizeof(float)" "Matched";
```

and

```
c_track "&x" "sizeof(float)" "UnMatched";
```

with the first of these two being the default if no third argument is provided (and backward compatible with the original definition of the `c_track` primitive in [9]).

The value of *unmatched* data objects is saved on the search stack, but not in the state descriptor.

The resulting SPIN nested-depth first search algorithm is identical to the one

```
1 proc dfs1(s)
2        add s to Stack1
3        add {f(s),0} to States
4        for each transition (s,a,s') do
5                if {f(s'),0} not in States then dfs1(s') fi
6        od
7        if accepting(f(s)) then seed := {f(s),1}; dfs2(s,1) fi
8        delete s from Stack1
9 end

10 proc dfs2(s)   /* nested search */
11       add s to Stack2
12       add {f(s),1} to States
13       for each transition (s,a,s') do
14               if {f(s'),1} == seed then report cycle
15               else if {f(s'),1} not in States then dfs2(s') fi
16       od
17       delete s from Stack2
18 end
```

Fig. 2. Nested Depth-First Search with Abstraction (cf. Fig. 8 in [1]).

discussed in [1] in the context of a discussion on symmetry reduction. For convenience, Figure 2 reproduces the algorithm as it was discussed in [1] (see [9] for a more basic description of the standard nested depth-first search algorithm). The abstract representation of a state is computed here by abstraction function `f(s)`.

2.2. Validity of Abstractions

The extension of the `c_track` primitive allows us to include data in a model that has relevance to the accurate execution of implementation level code, but no relevance to the verification of that code.

The simplest use of this new option is to use it to track data without storing it in the model checker's state-vector, where before the only way to track it would have been to do just that. When used in this way, the use of unmatched `c_track` primitives equates to *data hiding*.

Another use is to use unmatched `c_track` statements to hide the values of selected data objects from the state-vector, and then to add abstraction functions (implemented in C) to compute abstract representations of the data that will now be matched in the state-vector, using additional *matched* `c_track` primitives. We can now achieve true abstractions, though of course only of the value of data objects.

As a simple example of the latter type of use, consider two implementation level integer variables `x` and `y` that appear in an application program. Suppose further that the absolute value of these two variables can be shown to be irrelevant to the verification attempt, but the fact that the sum of these two variables is odd or even is relevant. We can now setup the required data abstraction for this application by defining the following `c_track` primitives:

```
/* data hiding */
        c_track "&x" "sizeof(int)" "UnMatched";
        c_track "&y" "sizeof(int)" "UnMatched";
/* abstraction: */
        c_track "&sumxy" "sizeof(unsigned char)" "Matched";
```

and we add the abstraction function:

```
c_code {
        void abstraction(void) { sumxy = (x+y)%2; }
}
```

which should now be called after each state transition that is made through calls on the application level code.

The abstractions have to be chosen carefully, to make sure that they preserve the logical soundness and completeness of the verification. This is clearly not true for all possible abstractions one could define with the mechanism we have discussed. Consider, for instance, the abstraction above in the context of the following model:

```
init {
        c_code { x = y = 0; abstraction(); };
        do
        :: c_code { x = (x+1)%M ; y = (y+1)%N ; abstraction(); }
        od
}
```

Suppose we are checking the invariant that $x + y$ is even. This invariant holds for the model above if both M and N are even, but not in general. For instance, it does not hold for the case $M = 3$, $N = 4$. In this case, the model checker would stop the search after exploring the first transition, and erroneously declare that the invariant holds. In the next subsection, we describe a sufficient condition for ensuring that the verification is sound with respect to the abstraction, for a given model.

2.3. Sufficient Conditions for Soundness

The model-checker checks properties by exploring the set of reachable states, starting from a predetermined initial state. Exploring a state consists of enumerating its successors, determining which of these states are potentially *relevant*, and recording the newly encountered states in some data-structure. This data-structure is, for instance, a stack in a depth-first search, and it is a queue in a breadth-first search.

Given a symmetric relation ~ on concrete states, a state s is considered *relevant* if the search has not seen any state t such that $s \sim t$. In the following discussion, we will refer to states that have been visited in the search at least once as *encountered* states, and to those encountered states whose complete set of successors has been computed as *explored* states. (Note that the use of no abstraction corresponds to the situation in which ~ is the identity relation. In this case a state is considered relevant if it has not been encountered before.)

The search algorithm we have outlined explores states in the concrete model, and in effect maintains a concrete path to each state on the depth-first search stack. Thus any abstraction relation is necessarily logically complete. To ensure that the abstraction relation ~ is also logically sound, we need to ensure that its use does not cause the search algorithm to miss any error states. Below, we present a condition for ensuring this. We use the following notation: w, x, y, z denote states, and σ, τ denote paths. We write σ_i to denote the i-th state in σ. We use $\rightarrow$ to denote the transition relation, so $x \rightarrow y$ denotes that there is a transition from state x to state y.

A symmetric relation on concrete states is a *bisimulation* [13] when it satisfies the following condition:

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

Thus 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. Given a bisimulation ~, we say paths σ and τ *correspond* when $\forall i$: $\sigma_i \sim \tau_i$.

The importance of bisimulation is given by the following theorem [1,2].

Theorem. Let ~ be a bisimulation, and let *AP* be a set of atomic propositions such that every proposition *P* in *AP* satisfies

$$\forall x, y: \; P(x) \wedge x \sim y \Rightarrow P(y) \tag{2}$$

Then, any two bisimilar states satisfy the same set of CTL* state formulas over propositions in *AP*. Furthermore, any two corresponding paths satisfy the same set of CTL* path formulas over propositions in *AP*. □

This means that when conditions (1) and (2) are both satisfied, the abstraction will preserve logical soundness.

3. Two Sample Applications

3.1. Tic Tac Toe

We will illustrate the use of the new verification option, and the types of data abstraction it supports, with a small example. For this example we will use a model of the game of tic tac toe. First, we will write the model in basic PROMELA, as a pure SPIN model and show its complexity. Then we will rewrite the model to include some operations in embedded C code, and we will show how abstractions can now be used to lower the verification complexity well below what was possible with the pure SPIN model, without sacrificing accuracy.

The pure PROMELA version of the model is shown in Figure 3.

We represented the 3x3 board in a two-dimensional array `b`, constructed with the help of PROMELA `typedef` declarations. Because the players in this game strictly alternate on moves, we can use a single process and record in a `bit` variable `z` which player will make the next move. Player 0 will always make the first move here. A square has value 0 when empty, and is set to either 1 or 2 when it is marked by one of the two players. In the first `if` statement, a player picks any empty square to place a mark. When no empty squares are left, a draw is reached and the game stops. When a move could be made, the player checks for a win, and if one is found it prints the board configuration for the winning position and forces a deadlock (to allow us to distinguish these states from normal termination where the process exits). We have enclosed the computation in an `atomic` sequence, to achieve that no intermediary states are stored during the model checking process, only the final board positions that are computed.

The verification of this model explores 5,510 states. Clearly, we are not exploiting the fact that the game board has many symmetries. There are, for instance, both rotational symmetries and mirror symmetries (left/right and top/bottom) that could be taken into account to reduce verification complexity. In principle, the SPIN model could be rewritten to account for these symmetries, but this is surprisingly hard to do, and risks the introduction of inaccuracy in the verification process.

With careful reasoning, we can see that there are only 765 unique board positions in the game, and 135 of these positions are winning for one of the two players, e.g. [14]. The maximum number of moves that can be made is further trivially 9. In our first verification attempt we therefore did considerably more work than is necessary.

```
#define SQ(x,y)   !b.r[x].s[y] -> b.r[x].s[y] = z+1
#define H(v,w)  b.r[v].s[0]==w && b.r[v].s[1]==w && b.r[v].s[2]==w
#define V(v,w)  b.r[0].s[v]==w && b.r[1].s[v]==w && b.r[2].s[v]==w
#define UD(w)   b.r[0].s[0]==w && b.r[1].s[1]==w && b.r[2].s[2]==w
#define DD(w)   b.r[2].s[0]==w && b.r[1].s[1]==w && b.r[0].s[2]==w

typedef Row   { byte s[3]; };
typedef Board { Row  r[3]; };

Board b;
bit z, won;

init {
         do
         :: atomic { /* do not store intermediate states */
            !won ->
            if     /* all valid moves */
            :: SQ(0,0) :: SQ(0,1) :: SQ(0,2)
            :: SQ(1,0) :: SQ(1,1) :: SQ(1,2)
            :: SQ(2,0) :: SQ(2,1) :: SQ(2,2)
            :: else -> break /* a draw: game over */
            fi;

            if    /* winning positions */
            :: H(0,z+1) || H(1,z+1) || H(2,z+1)
            || V(0,z+1) || V(1,z+1) || V(2,z+1)
            || UD(z+1)  || DD(z+1) ->
                /* print winning position */
                printf("%d %d %d\n%d %d %d\n%d %d %d\n",
                     b.r[0].s[0], b.r[0].s[1], b.r[0].s[2],
                     b.r[1].s[0], b.r[1].s[1], b.r[1].s[2],
                     b.r[2].s[0], b.r[2].s[1], b.r[2].s[2]);
                won = true /* and force a stop */
            :: else -> z = 1 - z                /* continue */
            fi;

            } /* end of atomic */
         od
}
```

Fig. 3. Tic Tac Toe, Pure PROMELA Model.

We will now revise the model to make use of a C function to store the board configuration in a C data structure, and to perform the moves that are non-deterministically selected with a SPIN model (which now starts to perform the function of a test-harness around a C application).

We will retain the same basic structure of the algorithm. Again, a win will result in a deadlock, and a draw will lead to normal process termination. In the first version of the model with embedded C code, shown in Figure 4, we track and match all relevant external data, which in this case includes just the the board configuration.

We have introduced variables `x` and `y` to record the square that is selected in the first part of the algorithm. The location of the square is passed to the C function that will now place the mark in that square and check for a win. The C function `play()` returns either a 2 if the last move made produced a win, or a 1 if it did not and the turn should go to the next player, as before.

```
#define SQ(a,b)   c_expr { (!board[a][b]) } -> x=a; y=b

c_decl {
         extern short board[3][3];
         extern short play(int, int, int);
};

c_track "&board[0][0]" "sizeof(board)";      /* matched */

byte x, y, z, won;

init {
   do
   :: atomic { !won ->
         if   /* all valid moves */
         :: SQ(0,0) :: SQ(0,1) :: SQ(0,2)
         :: SQ(1,0) :: SQ(1,1) :: SQ(1,2)
         :: SQ(2,0) :: SQ(2,1) :: SQ(2,2)
         :: else -> break   /* a draw */
         fi;
         c_code {
            switch (play(now.x, now.y, now.z+1)) {
            default: printf("cannot happen\n"); break;
            case 1: now.z = 1 - now.z; break;
            case 2: now.won = 1; break; /* force a stop */
            }
            now.x = now.y = 0; /* reset */
         }
      }
   od
}
```

Fig. 4. Tic Tac Toe, Version with Embedded C Code.

The external C function is shown in Figure 5. Perhaps not surprisingly, this model explores the same number of states as the pure SPIN model, and declares the same number of winning positions, though it does not have to search as deeply into the depth-first search tree (31 steps instead of 40 for the earlier model).

We will now change the last model into one that uses data abstraction. The new model is shown in Figure 6.

We have turned off state matching on the board configuration, while retaining the tracking capability that allows us to perform an accurate depth-first search. We have also introduced a new integer variable named `abstract` that we will use to record an abstract value of the board configuration, taking into account all symmetries that exist on the game board. This value is both tracked and matched. The rest of the test harness specification is unchanged.

We must now extend the C function a little, to provide the computation of the abstract board value. We do so by calling an extra function `board_value()` that is called in function `play()` immediately after the new mark is placed.

The details of this computation, shown in the Appendix, are not too important. Suffice it to say that each board configuration is assigned a unique value between 0 and 19682, by assigning a numeric place value to each square. The board value is computed for each rotation and mirror reflection of the board, and the minimum of the 8 resulting numbers is selected as a canonical representation of the set. That number is

```
#define H(a,b) (board[a][0]==b && board[a][1]==b && board[a][2]==b)
#define V(a,b) (board[0][a]==b && board[1][a]==b && board[2][a]==b)
#define UD(b)  (board[0][0]==b && board[1][1]==b && board[2][2]==b)
#define DD(b)  (board[2][0]==b && board[1][1]==b && board[0][2]==b)

short board[3][3];

short
play(int x, int y, int z)
{
    board[x][y] = z;       /*place mark */

    /* check for win: */
    if ((H(0,z) || H(1,z) || H(2,z)
    ||   V(0,z) || V(1,z) || V(2,z)
    ||   DD(z) || UD(z))
    {    Printf("%d %d %d\n%d %d %d\n%d %d %d\n",
         board[0][0], board[0][1], board[0][2],
         board[1][0], board[1][1], board[1][2],
         board[2][0], board[2][1], board[2][2]);
         return 2;         /* last move wins */
    }
    return 1;              /* game continues */
}
```

Fig. 5. C Source Code for Play().

stored in the state descriptor, and used in state matching operations. A state will now match if a board configuration is encountered that is equivalent to one previously seen, taking all rotational and mirror symmetries into account. Yet the execution of the actual code always works with the full detail on the actual board configuration.

Table 1. TicTacToe Verification.

Version	States	Depth	Wins
Pure SPIN Model	5510	40	942
Model with Embedded C Code	5510	31	942
Model with Embedded C Code and Data Abstraction	771	31	135
Minimum Required for Solution	765	9	135

The number of reachable states is with this data abstraction reduced to 771 states, with 135 of these state declared as winning positions, as shown in Table 1. Since the actual number of uniquely different board configurations is 765, SPIN encounters just six cases here where it revisits an old configuration with different values for the additional state variables `x`, `y`, or `z`.

It should be noted that we could also have used the `c_track` mechanism to introduce a new state variable that captures the non-abstracted board value (i.e., without taking into account the equivalence of rotations and mirror reflections of the game board). By storing and tracking the board value as a single 4-byte integer, rather than as the original 9-byte array of marks, we then define an application specific data compression on part of the state information, and similarly benefit by achieving a reduction of the memory requirements. This means that our new `c_track` mechanism can not just exploit user-defined abstractions, but also user-defined lossless or lossy data compression methods. Of course, for verification accuracy we will normally want to restrict

```
c_decl {
         extern short board[3][3];
         extern short play(int, int, int);
         extern short abstract;     /* board value */
};

c_track "&board[0][0]" "sizeof(board)" "UnMatched";
c_track "&abstract"    "sizeof(short)"; /* matched */

byte x, y, z, won;

init {
    do
    :: atomic { !won ->
        if    /* all valid moves */
        :: c_expr { (!board[0][0]) } -> x = 0; y = 0
        :: c_expr { (!board[0][1]) } -> x = 0; y = 1
        :: c_expr { (!board[0][2]) } -> x = 0; y = 2
        :: c_expr { (!board[1][0]) } -> x = 1; y = 0
        :: c_expr { (!board[1][1]) } -> x = 1; y = 1
        :: c_expr { (!board[1][2]) } -> x = 1; y = 2
        :: c_expr { (!board[2][0]) } -> x = 2; y = 0
        :: c_expr { (!board[2][1]) } -> x = 2; y = 1
        :: c_expr { (!board[2][2]) } -> x = 2; y = 2
        :: else -> break    /* a draw */
        fi;
        c_code {
            switch (play(now.x, now.y, now.z+1)) {
            default: printf("cannot happen0); break;
            case 1: now.z = 1 - now.z; break;
            case 2: now.won = 1; break; /* force a stop */
            }
            now.x = now.y = 0; /* reset */
        }
       }
    od
}
```

Fig. 6. Tic Tac Toe, With Data Abstraction.

the use to sound abstractions and lossless data compression.

3.2. Soundness

To see that the abstractions we have used in this example are logically sound, we show that they satisfies the conditions (1) and (2) from Section 2.3, as required by the theorem.

For given board configurations $b0$ and $b1$, the relation $b0 \sim b1$ is defined to hold when $b0$ and $b1$ evaluate to the same abstract board value (i.e., the configurations are equivalent upto rotations and reflections). It is easy to check that the state predicate P, where $P(b)$ denotes that b is a winning configuration, satisfies condition (2), since a win is invariant under rotations and reflections.

To see that the abstraction relation ~ satisfies condition (1), note that each transition in the model is either (i) a move by player 1, (ii) a move by player 2, or (iii) a win. We must then show that, for any two equivalent board configurations $b0$ and $c0$, if there is a transition from $b0$ to $b1$, then there is also a transition from $c0$ to $c1$ such that

$c1 \sim b1$.

Consider first a transition of type (i), in which player 1 places a mark at position (i, j). Suppose that $c0$ is a reflection of $b0$ across the vertical axis. Then, it is easy to see that the transition that places the same mark at position $(i, 2 - j)$ must be enabled in $c0$, and results in a state $c1$ that is equivalent to $b1$. A similar argument holds for the other ways in which $c0$ and $b0$ may be equivalent, and for transitions of type (ii).

For transitions of type (iii), it suffices to note that function `play` returns the same value for all equivalent board configurations.

3.3. A Larger Application

For a larger application we will discuss the verification of one of the modules from the flight software for JPL's Mars Exploration Rovers (MER).

The MER software contains 11 threads of execution. Each thread serves one specific application, such as imaging, controlling the robot arm, communicating with earth, and driving. There are 15 shared resources on the rover, to which access must be controlled by an arbiter, which is the target of our verification. The arbiter module prevents potential conflicts between resource requests, and enforces priorities. For instance, it would not make sense to start a communication session with earth while the rover is driving. The policy in this case is that communication is more important than driving, so when a request for communication is received while the rover is driving, the arbiter will make sure that the permission to use the drive motors is rescinded in favor of a new permission to use the rover's antennas.

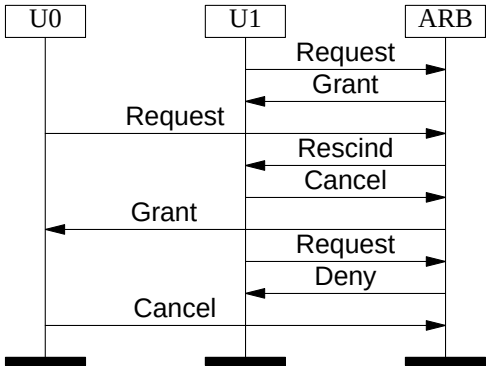

Fig. 7. Sample Communication Scenario.

Figure 7 shows a generic scenario for communication between user threads and the arbiter. In the scenario shown, user `U0` requests access to a resource, which is granted by the arbiter. Next, a different user `U1` makes a resource request that conflicts with the first one, but has precedence. The arbiter now sends a `Rescind` message to the first user, and waits for the confirmation that the resource is no longer used. Then the arbiter sends a `Grant` message to user `U1`. A new request from the first user while the second user still retains possession of the resource is now summarily denied by the arbiter. Eventually, the use of every resource must be completed by sending a `Cancel` message to the arbiter, which notifies the arbiter that the resource is now available for other users.

The arbiter module consists of about 3,000 lines of source code, written in ANSI standard C. The arbiter also makes use of a lookup table that records which combinations of resource requests conflict, and what the various priorities are.

With 11 users competing for 15 resources, the complexity of a full-scale exhaustive verification quickly becomes intractable. With the bitstate (supertrace) search mode, SPIN can randomly prune these large search spaces, within limits set by the size of available physical memory on the machine that is used. For exhaustive coverage a different strategy must be followed. One such strategy is divide and conquer. By breaking the problem down into smaller subproblems that can be checked exhaustively we can build confidence that the larger problem has the desired properties (although in this case we cannot conclusively prove it).

As an example, we will look at a subproblem with 3 user processes, competing for access to just 3 of the available resources, cyclically, and in random order. Without the use of abstraction, a problem of this size is at the edge of what can be verified exhaustively with roughly 1 Gbyte of available memory. An experiment like this can be repeated for different subproblems by making different selections of the 3 resources competed for, slowly increasing the confidence in the correctness of the complete solution.

Table 2. MER Arbiter Verification.

Version	States	Time(s)	Mem(Mb)
Pure SPIN Model	272,068	2.2	41.8
Model with Embedded Code	11,453,800	1458.0	701.7
Model with Embedded Code and Abstraction	261,543	5.1	55.2

Table 2 records the number of reachable states for three different versions of a verification model for the arbiter. The first version is a hand-built pure SPIN model of the arbiter algorithm, including only the lookup table from the original code as embedded C code. This model counts 245 lines of PROMELA code, plus 77 lines for the arbiter lookup table.

The second version uses the original arbiter C code, stubbed to isolate it from the rest of the flight code, with all relevant data objects that contain state information tracked and matched, using `c_track` statements, without abstraction. There is approximately 4,400 bytes of state information that is tracked in this way. This information includes, for instance, a linked list of current and pending reservations, and a freelist of reservation slots. A hand-built test-harness of just 110 lines of PROMELA surrounds the arbiter code (as it did for the much simpler tictactoe example we discussed before), simulating the actions of the 3 selected user processes. The verification for this version of the model could only be completed with the hashcompact state compression option enabled, which effectively reduced memory use to about 127 bytes per state. More effective than this compression option, though, is to use the data abstraction method we discussed. In the third version of the verification model we added these abstractions, restricting the state information that is recorded to its bare essence. In this version we exclude, for instance, the values of pointers that are used to build the linked lists.

Each of the three versions can find counter-examples to logic properties that have

known violations (including both safety and liveness properties). The first model, though efficient, can leave doubt about the accuracy of the modeling effort. The second model incurs the penalty of an implementation level verification, carrying along much more data than is necessary to prove the required properties. The third version of the model restores the relatively low complexity of a hand-built model, but has the benefit of precision and strict adherence to the implementation level code.

4. Related Work

There have been several different approaches to the direct verification of implementation level C code, following the early work on automated extraction of verification models from implementation level code as detailed in for instance [3,7,8].

Perhaps best known is the work on Verisoft [6], which is based on the use of partial order reduction theory in what is otherwise a state-less search. In this approach, application level code is instrumented in such a way that its execution can be controlled at specific points in the code, e.g., at points where message passing operations occur or where scheduling decisions are made. The search along a given path of execution is stopped when a user-defined depth is reached, and then restarted from the predefined initial system state to explore alternative paths. The advantage of this approach is that can consume considerably less memory than the traditional state space exploration methods used in logic model checking. It can therefore handle large applications. A relatively small disadvantage is that the method requires code instrumentation. A more significant disadvantage, compared to the method we have introduced in this paper, is that since no state space is maintained, none of the advantages from state space storage are available, such as systematic depth-first search, the verification of not only safety but also liveness properties, and the opportunity to define systematic abstractions. Abstraction in particular can provide significant performance gains. We believe that the methodology introduced in this paper is the first to successfully combine data abstraction techniques and *unrestricted* logic model checking capabilities with the verification of implementation level code.

A second approach that is comparable to our own is the work on the CMC tool [12]. In this tool, an attempt is made to capture as much state information as possible and to store it in a state space using agressive compression techniques, similar to the hash-compact and bistate hashing methods used in Spin [9]. Detailed state information is kept on the depth-first search stack, but no methodology available to distinguish between state information that should be matched, and state information that is only required to maintain data integrity. Potentially this method, though, could be extended with the type of data abstraction techniques we have described in this paper. We believe that effective use of data abstraction techniques will prove to be the key to the successful application of logic model checking techniques in practice.

5. Conclusions

The method we have described for verifying reactive software applications combines data abstraction with implementation level verification. The user provides a test-harness, written in the language of the model checker, that non-deterministically selects inputs for the application that can drive it through all its relevant states. Correctness properties can be verified in the usual way: by making logical statements about reachable and unreachable states, or about feasible or infeasible executions. The state tracking capability allows us to perform full temporal logic verification on

implementation level code.

The basic method of maintaining both concrete and abstract representations of states in the search procedure is very similar to algorithms that have been proposed earlier for using (predefined) symmetry reductions in model checkers, e.g. in [1,4].

The method is the easiest to apply in the verification of single-threaded code, with well-defined input and output streams. The two examples we discussed in this paper are both of that type. The method is not restricted to such applications though. Multi-threaded code can be handled, but requires more care. Each thread in the application will need to be prepared to run as standalone threads, with clearly defined inputs and outputs. The user must now identify the portions of program code that can be run as atomic blocks, setting the appropriate level of interleaving for the model checking runs. The test-harness that the user prepares now drives the thread executions directly, selecting the proper level of granularity of execution. The application we studied in [5] is of this type, and could be adapted to use the new method of data abstraction we have discussed here.

The capability to redefine how state information is to be represented, or abstracted, is also similar to the `view` function in TLC [11]. In our case, the abstraction can be defined in any way that C or C++ allows, but it is restricted to the representation of external data objects. Most of the data in a SPIN model could be treated as such (e.g., by declaring them to be `hidden` within the SPIN model itself), with the exception only of the program counters of active processes.

In the setup we have described, each call on the application level code is assumed to execute to completion without interleaving of other actions (i.e., atomically). This is the most convenient way to proceed, but we are not restricted to it. By using a model extractor, such as FeaVer or MODEX [10,15], we can convert selected functions in the application into PROMELA models with embedded C code, optionally using additional source level abstraction functions, and generate a finer-grained model of execution. The instrumentation required for model extraction can, however, complicate the verification process, and require deeper knowledge of the application.

Acknowledgements

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 idea to distinguish state information that is tracked on the stack from that stored in the state space was suggested by Murali Rangarajan. The authors also thank Dragan Bosnacki for inspiring discussions on the subject of this paper.

References

[1] D. Bosnacki, *Enhancing state space reduction techniques for model checking*. Ph.D Thesis, 2001. Eindhoven Univ. of Technology, The Netherlands.

[2] E.M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.

[3] J.C. Corbett, M.B. Dwyer, J.C. Hatcliff, et al., Bandera: Extracting finite state models from Java source code, *Proc. 22nd Int. Conf. on Softw. Eng.*, June 2000, Limerick, Ireland, ACM Press, pp. 439-448.

[4] E.A. Emerson, C.S. Jutla, A.P. Sistla, On Model-Checking for Fragments of mu-Calculus. *Proc. CAV93*, LNCS 697, pp. 385-396.

[5] P.R. Gluck and G.J. Holzmann, Using Spin Model Checking for Flight Software Verification, *Proc. 2002 Aerospace Conference*, IEEE, Big Sky, MT, USA, March 2002.

[6] P. Godefroid, S. Chandra, C. Palm, Software model checking in practice: an industrial case study, *Proc. 22nd Int. Conf. on Softw. Eng.*, Orlando, Fl., May 2002, ACM Press, pp. 431-441.

[7] K. Havelund, T. Pressburger, Model checking Java programs using Java Pathfinder, *Int. Journal on Software Tools for Technology Transfer*, Apr. 2000, Vol. 2, No. 4, pp. 366-381.

[8] G.J. Holzmann, Logic Verification of ANSI-C Code with Spin, *SPIN Model Checking and Software Verification*, Springer Verlag, LNCS Vol. 1885, pp. 131-147, Sep. 2000.

[9] G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.

[10] 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, 4, pp. 364-377, April 2002.

[11] L. Lamport, *Specifying Systems: the TLA+ language and tools for hardware and software engineers*, Addison-Wesley, 2002.

[12] M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, D.L. Dill, CMC: A pragmatic approach to model checking real code, *Proc. Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[13] D. Park, Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, pp. 167-183. Springer, 1981.

[14] `http://f2.org/maths/ttt.html`

[15] `http://cm.bell-labs.com/cm/cs/what/modex/`

APPENDIX

```
#define MAXVAL  19682   /* 3 x 3^8 - 1 */
#define B(a,b,c,d)      board[a][b]*r[c][d]

int abstract;

int r0[3][3] = {
        {   1,    3,    9, },
        {  27,   81,  243, },
        { 729, 2187, 6561, },
};

int r1[3][3] = {
        {  729,  27,  1, },
        { 2187,  81,  3, },
        { 6561, 243,  9, },
};

int r2[3][3] = {
        { 6561, 2187, 729, },
        {  243,   81,  27, },
        {    9,    3,   1, },
};

int r3[3][3] = {
        { 9, 243, 6561, },
        { 3,  81, 2187, },
        { 1,  27,  729, },
};

int
comp_row(int r[3][3], int L, int R, int T, int B)
{
        return  B(T,L,0,0) + B(T,1,0,1) + B(T,R,0,2) +
                B(1,L,1,0) + B(1,1,1,1) + B(1,R,1,2) +
                B(B,L,2,0) + B(B,1,2,1) + B(B,R,2,2);
}

void
min_row(int r[3][3])
{       int v;

        v = comp_row(r,0,2,0,2); if (v < abstract) abstract = v;
        v = comp_row(r,2,0,0,2); if (v < abstract) abstract = v;
        v = comp_row(r,0,2,2,0); if (v < abstract) abstract = v;
        v = comp_row(r,2,0,2,0); if (v < abstract) abstract = v;
}

void
board_value(void)
{
        abstract = 2*MAXVAL;
        min_row(r0);
        min_row(r1);
        min_row(r2);
        min_row(r3);
}
```