Verifying MIPS Assembly Programs via CTL Model Checking

Darko Poposki[*], Risto Petroski[†],
Dimitar Stojkovski[‡], Guido Sciavicco[§]
University for Inf. Sci. and Techn. of Ohrid, Macedonia

April 12, 2011

Abstract

Formal verification of low-level software has become a trend in the formal methods community. Over the past decade, there has been considerable advancement in the theory and practice of automated formal software verification, and one of the most promising paradigms to emerge in this area is software model checking. In this work, we consider a possible application of this methodology to the verification of MIPS Assembly programs. We consider very small programs and very simple safety requirements, such as *a register is never used before initialization.* We present a newly developed, non-optimized, software system to this purpose, whose distinctive feature is being specifically tailored for MIPS programs, and that allows a number of possible generalization in the type and the scope of the properties that can be checked. This work has been undertaken while three out of four authors were first-year undergraduate students at the University of Information Science and Technology in Ohrid (Macedonia).

1 Introduction

Formal verification of low-level software has become a trend in the formal methods community. This is justified by the fact that our lives rely on this software. Indeed, nowadays cars brake systems rely on such software, as does the flying controls of our airplanes, the vital medical devices in our hospitals, and the probes that maintain the security in our power plants. Beyond these crucial stakes, the difficulty of such software verification also motivates the community. Indeed this software verification presents a challenge. Low-level software are

[*] *darko.poposki@cse.uist.edu.mk*
[†] *risto.petroski@cse.uist.edu.mk*
[‡] *dimitar.stojkovski@cse.uist.edu.dk*
[§] *guido.sciavicco@uist.edu.mk*

complex because they directly manipulate their underlying hardware, because, for sake of code reusability, they are written in different languages and finally because, for most of them (like operating systems), implement subtle control-flow. Over the past decade, there has been considerable advancement in the theory and practice of automated formal software verification. One of the most promising paradigms to emerge in this area is *software model checking* (see, e.g., [1, 3, 4]) a combination of counterexample guided abstraction refinement with predicate abstraction. Model checking verifies that a program P satisfies a specification φ iteratively, as follows:

- (Abstraction) Construct a conservative model M from P via predicate abstraction;
- (Verification) Model check $M \Vdash \varphi$. If this is the case, then terminate with result YES. Otherwise let C be a counterexample returned by the model checker.
- (Validation) Check whether C corresponds to some concrete behavior of P.

One of the most widely used languages in the verification phase is *(point-based) temporal logic*, introduced by Pnueli [9] (see also [7]). It comes usually into two different variations: *linear* and *branching.* The most famous and used branching temporal logic for program specification is Computational Tree Logic (CTL, for short). This is a modal propositional logic, interpreted over tree-like (possibly infinite) structures, and it features *branch quantifiers* such as A (for all) and E (there exists), and *future operators* such as X (the next) and U (until). The distinctive feature of CTL is that branch quantifiers and operator must be used together; therefore, we can express, for example, $AX\varphi$ (for all branches rooted here, there exists a next point that satisfies φ), but not $A\varphi$ or $X\varphi$. As noticed in [5] this apparently small limitation allows one to perform model checking of a CTL-formula in linear time (w.r.t. the length of the formula and the dimension of the model). Probably, the most famous implementation for general purpose temporal logic model checking is SPIN [6]; among other tools for low-level software verification, we cite [2, 8].

In this work, we consider a possible application of this methodology to the verification of MIPS Assembly programs. We consider very small programs and very simple safety requirements, such as: *a register is never used before initialization*; but, in a very similar way, other properties can be checked. In particular, we can check *warning properties*, such as: *a register which is initialized, should be used in the future*, or more specific, instruction related, properties. We present here a newly developed, non-optimized, software system to this purpose, whose distinctive feature is being specifically tailored for MIPS programs, and that allows a number of possible generalization in the type and the scope of the properties that can be checked.

This work has been undertaken while three out of four authors were first-year undergraduate students at UIST in Ohrid (Macedonia).

Symbolic Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for the assembler
v0 - v1	2 - 3	Result Registers
a0 - a3	4 - 7	Argument Registers 1 to 4
t0 - t9	8 - 15, 24 - 25	Temporary Registers 0 to 9
s0 - s7	16 - 23	Saved Registers 0 to 7
k0 - k1	26 - 27	Kernel Registers 0,1
gp	28	Global Data Pointer
sp	29	Stack Pointer
fp	30	Frame Pointer
ra	31	Return Address

Table 1: MIPS register set.

2 MIPS Architecture and Assembly

MIPS (originally an acronym for Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies. The early MIPS architectures were 32-bit, and later versions were 64-bit. Multiple revisions of the MIPS instruction set exist, including MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS32, and MIPS64. The current revisions are MIPS32 (for 32-bit implementations) and MIPS64 (for 64-bit implementations). MIPS32 and MIPS64 define a control register set as well as the instruction set.

MIPS has 32 integer registers. Data must be in registers to perform arithmetic. The MIPS R2000 CPU (which is the one we are interested in) has 32 registers. 31 of these are general-purpose registers that can be used in any of the instructions. Register \$0 always holds 0, and register \$1 is normally reserved for the assembler (for handling pseudo instructions and large constants). Even though any of the registers can theoretically be used for any purpose, MIPS programmers have agreed upon a set of guidelines that specify how each of the registers should be used. Programmers (and compilers) know that as long as they follow these guidelines, their code will work properly with other MIPS code. Registers are shown in Tab. 2.

For the purposes of this work, we will use only a small subset of the MIPS instruction set. In particular, we will limit ourselves to the following type of operations: arithmetical operations, label definitions, unconditioned branches, conditioned branches, immediate operations, syscalls. Moreover, we exclude the possibility of defining functions and using the stack. As a general rule, (pseudo) instructions are written in the following form:

$$opcode\ dest\ param1\ param2$$

where *opcode* is the operation code, *dest* is the destination register (or the label in case of a branching instruction), *param*1 (resp., *param*2) is the first (resp.,

Symbolic Name	Meaning	Usage Example
li	Load Immediate	*li* $\$t0$
add	Addition	*add* $\$t0, \$t1, \$t2$
mult	Multiplication	*mult* $\$t0, \$t1$
div	(int.) Division	*div* $\$t0, \$t1$
b	(unc.) Branch	*b label*
bgt	Branch if >	*bgt* $\$t0, \$t1, label$
syscall	System Call	*syscall*

Table 2: Some MIPS Assembly Instructions.

second) register used in the operation. Special cases are simple labels, of the form:

$$label:$$

which represent specific points of the program reachable through branching instructions (in order to modify the control flow), and branching instructions, which can be unconditioned, such as

$$b\ label$$

that moves the control flow immediately after the label *label*, or conditioned, such as, for example,

$$bgt\ \$t0, \$t1, label$$

which implies that the control flow goes after the label *label* if and only if the current value of $\$t0$ is greater than the current value of $\$t1$. The complete set of operations which are of interest for us is shown in Tab. 2.

The system calls, as well as the operations of (integer) *mult* and *div* are also particular cases for us. Indeed, system calls are managed as follows: the operation *syscall* has an effect that depends on the particular code currently loaded into the register $\$v0$. For some system calls, such as *printing an integer on the console*, it is also needed another parameter, usually loaded into $\$a0$. Multiplications and divisions present a different problem: after a multiplication, the result is loaded into the register *lo*, while (possible) overflow, into the register *hi*; after a division, the result is loaded into *lo*, and (possible) rest, into *hi*. Therefore, the format will be different, since we do not need to specify the destination registers.

2.1 An Example of Assembly Program

Suppose that we want to solve the following simple problem: by means of a cycle, we want to sum all natural numbers from 1 to 10 into a certain register. We use the register $\$t0$ as a counter, starting from 1, and the register $\$t1$ as the repository for all partial results. For the purposes of this example, we omit the details such as the input/output system calls, as well as the necessary call

$.text$
$main:$
$li\ \$t0, 1$
$li\ \$t1, 0$
$beg\_cycle:$
$bgt\ \$t0, 10, end\_cycle$
$add\ \$t1, \$t1, \$t0$
$addi\ \$t0, 1$
$b\ beg\_cycle$
$end\_cycle:$

Table 3: The example MIPS program to compute the sum of the first 10 integers

to ensure the correct exit from the program. The directive *.text* denotes the beginning of the instructions, and the label *main* is a necessary label to ensure that the interpreter knows the starting point. Therefore, our program will be as in Tab. 3.

3 The Temporal Logic CTL Interpreted over Assembly Programs

In this section, we recall the basic definitions of the syntax and the semantics of Computational Tree Logic [5], commonly used for specifying and verifying software and hardware models in the past two decades. Later, we will show how we use this logic in order to verify the properties that interesting for us.

The syntax of CTL is based on a set of propositional letter $\mathcal{AP} = \{p, q, \ldots\}$, using classical connectives such as $\neg$ and $\vee$, the unary temporal operators AX, EX, and the binary temporal operators AU and EU. Formulas are generated by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \psi \mid AX\varphi \mid EX\varphi \mid AU(\varphi, \psi) \mid EU(\varphi, \psi).$$

CTL is interpreted over *models* of the type $M = \langle\langle W, s\rangle, V\rangle$, where $\langle W, s\rangle$ is a *discrete tree order*, $s \subset W \times W$ is an *immediate successor relation*, and $V : W \rightarrow 2^{\mathcal{AP}}$ is a *valuation function* that assigns the set of all and only propositional letters true at every *world* $w \in W$. The transitive closure of the successor relation, denoted by s^*, univocally defines the set of all *branches* $\beta_1^w, \ldots, \beta_n^w$ rooted in a world w. Intuitively, A and E are the universal quantifier over the possible futures (branches) starting at the current world; X denotes the *next time* operator, and it is true if and only if its argument is true on the next world in the order. The symbol U stays for *until*, and it imposes the existence of a future world v on which a certain property is true, while another property

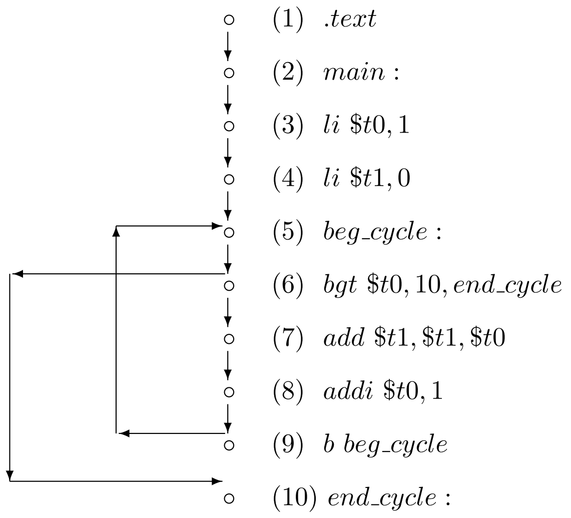

Figure 1: The temporal model corresponding to the MIPS program in Tab. 3.

must be true in every world between the current one (included) and v itself. Notice that CTL does not allow us to use X or U without quantifying over the branches. Formally, the semantics of CTL is given by the following clauses:

- $M, w \Vdash p$ if and only if $p \in V(w)$;
- $M, w \Vdash \neg\varphi$ if and only if it is not the case that $M, w \Vdash \varphi$;
- $M, w \Vdash \varphi \vee \psi$ if and only if $M, w \Vdash \varphi$ or $M, w \Vdash \varphi$;
- $M, w \Vdash AX\varphi$ if and only for all v such that $s(w, v)$, it is the case that $M, v \Vdash \varphi$;
- $M, w \Vdash EX\varphi$ if and only there exists some v such that $s(w, v)$ and $M, v \Vdash \varphi$;
- $M, w \Vdash AU(\varphi, \psi)$ if and only for all branches β_i^w there exists a world $v \in \beta_i$ such that $w \neq v$, $M, v \Vdash \psi$, and, for every world $w_j \in \beta_i$, where $w = w_1$, $s(w_1, w_2)$, ..., it is the case that $M, w_j \Vdash \varphi$;
- $M, w \Vdash EU(\varphi, \psi)$ if and only there exists a branch β_i^w such that there exists a world $v \in \beta_i$ where $w \neq v$, $M, v \Vdash \psi$, and, for every world $w_j \in \beta_i$, where $w = w_1$, $s(w_1, w_2)$, ..., it is the case that $M, w_j \Vdash \varphi$.

Other classical operators, such as $\rightarrow$ and $\wedge$, can be defined as standard. Similarly, other commonly used temporal operators, such as AF (for all branches, there exists in the future) or EF (there exists a branch such that there exists in the future), can be defined in terms of this language; for example:

$$EF\varphi = EU(\top, \varphi).$$

A CTL formula φ is said to be *satisfiable* if and only if there exists a model M and a world w on it, such that $M, w \Vdash \varphi$. In this paper, we are interested in a different, but equally well-known, problem, called *model checking.* This is the problem of establishing, given a model M and a world w on it, if $M, w \Vdash \varphi$. Notice that if a specific model satisfies a specific formula φ, then φ is indeed satisfiable, but not the other way around. Therefore, model checking can be considered a simpler problem. For CTL, this problem has been deeply studied.

CTL has been successfully used to express useful temporal properties. In this paper, we will express in CTL interesting temporal properties that we are interested to check over a MIPS program. As a (running) example, we consider the following property: *it is never the case that a register is used if it has not been initialized before.* Notice that over a specific program, we need to check this kind of requirement for all register: we will solve this problem by successively instantiating the corresponding formula to each specific register of interest. Therefore, by denoting with the propositional letter a the fact that the (current) register is accessed, and by i the fact that it is instantiated, the formula corresponding to the above requirement, is:

$$\neg(EU(\neg i, a)). \tag{1}$$

As observed in [5], a (concurrent) program can be modeled as a directed, possibly cyclic, graph. In our case, an MIPS assembly program can be seen as a sequence of instructions which can be modeled as a sequence of worlds temporally related to each other. When there are no branching instructions, then the logical sequence of the instructions follows exactly the order in which they are written. In case of a branching instruction, the logical sequence might change; in particular, a world corresponding to a unconditional branching implies that, in general, the (only) possible future is different from the structural future, and a world corresponding to a conditional one implies the existence of two, different, possible futures, one of them corresponding to the structural future and the other one different from it. So, for example, the MIPS program shown in Tab. 3 can be modeled as in Fig. 1, where each instruction has been given a number only for the sake of clarity.

In order to complete the link between CTL and (the semantics of) MIPS programs, we just need to *unravel* a cyclic graph such as the one of Fig. 1. The effect of this unraveling is that our program is seen an infinite tree, each branch of which represents a possible *computation* (or *evolution*) of the program. As for example, a possible computation is:

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 5, 6, 7, 8, 9, 5, 6, 10.$$

Now, consider the formula (1). In order to answer the question of if it is satisfied my the model corresponding to our program, we need to enrich the model itself in such a way that each node is labeled with those propositional letters (appearing in the formula) that are true at it. Therefore, the computation above becomes:

$$1, 2, 3(i[t0]), 4(i[t1]), 5, 6(a[t0]), 7(i[t1], a[t0, t1]), 8(i[t0]),$$
$$9, 5, 6(a[t0]), 7(i[t1], a[t0, t1]), 8(i[t0]), 9, 5, 6(a[t0]), 10.$$

It is easy to see now that 1 satisfies (1); therefore, this specific computation is *safe* w.r.t. to the property we are interested in. The problem of checking all (infinitely many) possible computations in a finite amount of time, has been solved in [5], and, in the next section, we will see how we applied that algorithm to our case.

4 The Verifier

Our system is designed into three, connected, parts, as follows.

1. MIPS Program reading and model building;
2. Safety requirement reading and tree-creation;
3. Checking.

Let us now examine each one of these parts.

Phase 1. MIPS programs are supposed to be written in text files. Each line must contain an instruction and, possibly, a comment line. For each line of the program, a new node of the model is created, and the physical (structural) link is created. Moreover, for each line, we interpret the instruction and we set the truth value of two boolean arrays $ValueSet$ and $Accessed$, such that, for each position i, $ValueSet[i]$ (resp., $Accessed[i]$) is 1 if and only the register i is initialized (resp., accessed) at that instruction. Notice that we use a unique enumeration of all (interesting) registers. After the creation, a new function sets the logical sequence of operations: when there is no branching instruction, the logical sequence is exactly equivalent to the physical one; for every branching instruction, the logical sequence is modified (as in Fig. 1) in order to take into account all possible evolutions of the computation. At the end of this phase, the graph is prepared with two (possibly different) links from one node to it(s) next one(s). This process might possibly generates cycles.

Phase 2. A different text file contains the formulas we want to be checked. For the sake of simplicity, we assume that each formula is preceded by a unique code that identifies its 'type'. In this first experiment, we limit ourselves to the code '0', by which we mean that the formula must be interpreted as universally quantified over all registers. So, for example, the formula (1) actually represents the conjunction of formulas:

$$\forall reg(\neg(AU(\neg i[reg], a[reg]))). \tag{2}$$

Moreover, formulas are supposed to be written in *prenex - left first* order. Therefore, our example formula (2) is actually written as:

$$0\neg AU\neg ia. \tag{3}$$

A suitable, unique encoding is used to represent temporal and classical operators. At the end of this function, it is returned the pointer to the binary tree that represents the formula in question.

Phase 3. This stage consists of applying Emerson and Clarke's algorithm to our model. The process is carried on over all levels of the formula, from the greatest to the lowest. For each level, for each (sub)formula ψ at that level, and for each node w of the structure, it is checked if $w \Vdash \psi$; if the answer is positive, a link is created from w to the node (in the formula tree) corresponding to ψ. Boolean combinations are taken care of by means of a purely local analysis: for example, $\tau_1 \wedge \tau_2$ is true at the node w if and only if both links to τ_1 and τ_2 are present at the node w itself (notice that τ_1 and τ_2) are at a lower level w.r.t. τ, and therefore are taken car of in the previous cycle). As for AX and EX, the analysis ranges over w itself and every possible w' such that w' is the next node in any of the possible futures of w. Similarly, AU and EU are treated by analyzing all branches rooted in w: notice that either the branches are finite, or they must be analyzed up to the first repetition of a node.

If, at the end of all steps, a pointer from the root of the structure to the root of the formula tree is present, then that formula is true in the current model.

5 Example of Verification

Let us consider the program in Tab. 3, and the formula (3). After Phase 1, a structure such as the one displayed in Fig.1 is created. The formula (3) is represented into a binary tree with exactly 4 levels. Iteratively, we start by checking the instance of this formula corresponding to the register $\$t0$. We start with level 3 (the last one), and we associate i with the node corresponding to the instruction 3. The, as for level 2, a is associated with the nodes 6 and 8, and $\neg i$ with all nodes but 3. As for level 1, the formula $EU\neg ia$ is not associated with any node, and, therefore, at level 0, all nodes, and, in particular the root, is associated with $\neg EU\neg ia$, proving that our program is safe w.r.t. the particular requirement instantiated over $\$t0$. An identical process is completed for the instance $\$t1$.

6 Conclusions and Open Problems

References

[1] S. Chaki. Sat-based software certification. In *Proceedings of the 12th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS2006)*, pages 151–166, 2006.

[2] S. Chaki and J. Ivers. Software model checking without source code. *ISSE*, 6(3):233–242, 2010.

[3] E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *Proceedings of ICSE 2003*, pages 385–395, 2003.

[4] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *Proceedings of the 10th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS2004)*, pages 168–176, 2004.

[5] E. Allen Emerson. Temporal and modal logic. In *HANDBOOK OF THEORETICAL COMPUTER SCIENCE*, pages 995–1072. Elsevier, 1995.

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

[7] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer, 1992.

[8] N. Marti. *Formal Verification of Low-Level Software.* PhD thesis, 2008.

[9] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, pages 510–584. 1986.