Data Race Detection for Interrupt-Driven Programs via Bounded Model Checking

Xueguang Wu[†] xueguangwu@sina.cn
Yanjun Wen[†] y.j.wen@263.com
Liqian Chen[†‡] lqchen@nudt.edu.cn
Wei Dong[†] dong.wei@263.com
Ji Wang[†] wj@nudt.edu.cn

[†] National University of Defense Technology, Changsha 410073, China
[‡] National Laboratory for Parallel and Distributed Processing, Changsha 410073, China

***Abstract*—In Cyber-Physical Systems with interrupt mechanism, interrupts may cause unexpected interleaving executions and even wrong execution results. A kind of frequently occurred errors are caused by data race. We present an approach under the framework of bounded model checking (BMC) to detect data race for interrupt driven programs. The key idea is to automatically serialize a concurrent interrupt driven program as a non-deterministic sequential program, whose possible execution set includes all the possible executions of the interrupt driven program. Moreover, our approach checks data race in the sequential program and collects all the path condition of the data race location. On this basis, we leverage bounded model checking to convert all the path conditions into SMT formulae. Furthermore, our analysis uses a decision procedure to determine whether the formula is satisfiable, from which the analysis eliminates false alarms which can't occur in real concurrent executions. A prototype based on CBMC is implemented and preliminary experimental results are encouraging.**

Keywords*—Key words: interrupt-driven programs; data race; bounded model checking; Satisfiability Modulo Theories; Cyber-Physical Systems; concurrent

I. Introduction

Interrupt driven program is becoming more and more popular in Cyber-Physical Systems (CPS), because interrupt mechanism is suitable for many features of CPS. However, the execution of interrupt driven program may cause complicated interleaving state space which may contain some unexpected execution results. One significant problem is caused by data race. Roughly speaking, a data race on a shared variable means that two threads access that variable simultaneously and the accesses are conflicting in concurrent program. Conflict assessing means that at least one of the accessing is writing the variable. Although some races are unharmful, bugs are often caused by data race. Hence, data race detection are valuable for concurrent programs.

Data race may cause software failure, and such kind of problem is hard to discover via software testing. Consider the following example in Fig.1 , which is an example of data race in the interrupt-driven program. The function *Get_on_board_time* and *Second_interrupt* are called by task and interrupt respectively and *s_second* and *s_millisec* are shared variables. If the interrupt is triggered when the task execute between line 4 and 5, then we will find that the time value in *result* is incorrect. Such kind of problems may cause serious results in some application scenarios such as aerospace and aeroplane. Because this kind of bug can only be triggered in a very special situation, software testing is hard to find this kind of problem.

```
1  /*main task*/
2  Time Get_onboard_time(void){
3    Time   result;
4    result.second = s_second;
5    result.millisec = s_millisec;
6    return   result;
7  }
8  /*interrupt*/
9  void Second_interrupt(void){
10   s_second ++;
11   s_millisec = 0;
12 }
```

Fig. 1: Example of Data Race in Interrupt-Driven Programs

Many methods have be adapted to detect data race in recent years. Static analysis of data race provides a powerful approach to find race condition, and much progress has been achieved in the past two decades [1][2][3]. However, static analysis itself has limited ability in precision. Dynamic analysis of data race is also used in industry [4][5]. Compared with static analysis, dynamic analysis is incomplete in race detection. Software bounded model checking is used to overcome the state explosion of software model checking. Many progress have been achieved in recent years [6][7]. Most of the previous works focus on symbolic bounded model checking, but in recent years much attentions have been focused on explicit bounded model checking.

In this paper, we present an approach to detect data race in the framework of bounded software model checking. We firstly serialize an interrupt-driven program into a sequential programs. Then we use the traditional relationship of accessing variables to find the potential data race. Lastly, we use software bounded model checking to eliminate the false alarms. On this basis, a prototype is implemented and preliminary experimental results are presented on benchmark programs.

This work is supported by the National Natural Science Foundation of China (NSFC) under Grant Nos.90818024, 91118007, 61120106006, 61202120, and the National High Technology Research and Development Program of China (863 Program) under Grant No. 2011AA010106.

The rest of the paper is organized as follows. Section II discusses some related work. Section III describes the features of our interrupt driven program. Section IV presents a race detection method based on accessing relationships of shared variables and a false alarm elimination method via BMC. Section V presents our prototype implementation together with potential experimental results before Section VI concludes.

II. Related work

Lock set data race detection. Data race detection for concurrent program has gained much attention within the past decade [8][4][1][5]. And various methods have been used for race detection and false alarm elimination. Such as high level race detection [8], binary rewriting [4], locksmith's component analysis [1]. Cyrille Artho et al. [8] raise a new kind data abstraction method in high level. Because the classical notion of data race is not powerful enough, which may miss some consistencies occurring in practice and the inconsistence may cause false alarms in the classical data race detection. E.g., the structural type data may have to be treated atomically in traditional race detection, but in high level race detection this data will be treated as different shared variables. This method indeed will reduce some kind of false alarms. Stefan Savage et al. [4] propose a dynamic data race detection method and implement a race checking tool Eraser. They focus on the data race caused by improper synchronization such as lock set. In Eraser they use binary rewriting techniques to record all the shared variables and verify whether the shared-memory reference is consistence. In fact because Eraser is a dynamic checking tool they need not consider whether the execution path is flexible. Whereas, since we use static analysis we need to consider whether the race location can be executed in the real execution. Recently, Polyvios Pratikakis et al [1] propose a static analysis method for C programs, and they also implement a static analysis tool named LOCKSMITH. They mainly focused on the efficiency and effective of data race detection for multi thread C program based on lock set. Their work also make some progress in C structures and void pointers. Compared with their work, our method is path sensitive static analysis method and the scalability of LOCKSMITH is better than our tool. Recently, Xinwei Xie et al [5] presents a dynamic race detection method based on optimize the happens before relationship (HBR) which is a classic method to reduce the state space of concurrent programs. The efficiency of their method is better or no worse than most of the previous HBR data race detection methods such as ERASER and FASTTRACK.

Path sensitive data race detection. Recently much attention has been focused on path sensitive data race detection for concurrent programs [2][3][9]. Thomas A. Henzinger et al [2] propose data race detection by context inference which is based on software model checking. Their method use predicate abstraction to get the proper model and counter example guided abstraction refinement (CEGAR) to permit automatic inference of models. Their method has been implemented in BLAST which is a model checking tool for sequential C programs. Compared with our work, BLAST uses predicate abstraction and CEGAR to automatic generate the interleaving state space, while we use BMC to get the proper interleaving state space. The work in [3] presents a data race detection method by symbolic execution [10]. Their method considers all the potential executions in static race detection. They implemented their work in Java PathFinder (JPF) which is a symbolic execution tool for Java byte code. The differences between JPF and our work is that JPF uses symbolic execution to find proper state space while we use BMC to exclude the unreachable states. More recently Vojdani et al [9] propose a path sensitive data race detection method. They propose a global invariant approach to sidestep the state space explosion and precision lose in both context and path sensitive. They have implemented their method in a tool called Goblint. Furthermore Goblint has been tested in about 25 thousand lines of code which is the "safe" subset of C. In fact the global invariant approach is much like model checking but their method adapted some optimization to avoid state explosion in data race detection.

Data race detection for interrupt driven programs. Much attention has been focused on data race detection of interrupt driven program [11][12]. One work that is close to our approach is race detection for embedded systems in [11]. Their work mainly focuses on the space flight control systems. They consider race detection synchronized by mask and unmask interrupts by bit-vectors and every byte of which represents whether the interrupt is masked or unmasked. They also support precise pointer alias checking which may find more precise data race. Compared with their work, our method doesn't consider synchronization at moment, our method mainly focuses on the path sensitive race detection to eliminate false alarms caused by unreachable execution path. Recently Martin D. Schwarz et al [12] propose a kind of race detection for interrupt driven programs with synchronization via ceiling protocol. Their work is the progressing work of Goblint. Their method is for priorities of tasks changing dynamically. For this kind of program they provide static analysis for race detection between tasks in which the priority is dynamically change. They also introduce a precise analysis of affine equalities adopted from value analysis. Compared with our work, their method is concerned about task priority dynamic changing, but we consider the fixed priority task. On the other hand, they use affine equalities to store the value and relationship of all variables which can be used to judge whether the path conditions can be satisfied, but we use SMT solver to solve whether there exists some values satisfying the path condition.

III. Interrupt Driven Programs

The Interrupt Driven Programs which we discuss in this paper have the following features:

- Program consists of one task and finite number of interrupts.
- All the interrupts have a fixed priority and every priority level has only one interrupt.
- A higher priority level interrupt can interrupt the task or lower priority level interrupts at any time. When task or lower priority level interrupts are interrupted they can execute again when the higher priority level interrupt finish.

In order to simplify the analysis procedure, we assume that interrupt driven program only have four kinds of basic syntax statements which are assignment, function call, condition goto

and return statement. All the other kinds of statements can be transformed to these four kinds. We use control flow automaton (CFA) to depict task and interrupts. The states of a CFA are associated to control points of CFG in the program. The transition between states in CFA is the control flow between the control points in the program. Thus, a CFA of a program is an abstraction of program's control flow graph which means that the CFA contains all the possible traces of program execution. Formally, the CFA is a 4-tuple $A = \langle S_{CF}, I_{CF}, T_{CF}, L \rangle$ where:

- S_{CF} is the set of control states.
- $I_{CF} \in S_{CF}$ is the set of initial control states.
- $T_{CF} \subseteq S_{CF} \times S_{CF}$ is the set of transitions between the control states.
- $L : S_{CF} \setminus \{END\_FUNCTION\} \rightarrow Stmt^{\star}$ is the label function where $Stmt^{\star}$ is the set of program expressions, $END\_FUNCTION$ is the end of function control states.
- S_{CF} has finite number of control states, but contains only one final control state $END\_FUNCTION$.

The transition between the control states represents the control flow in the program. $L(I_{CF})$ is the entry point of the function, and $(s_1, s_2) \in T_{CF}$ iff one of the following conditions hold:

- if $L(s_1)$ is an assignment or a function call statement, then $L(s_2)$ is its only successor.
- if $L(s_1)$ is a condition goto statement, then $L(s_2)$ is its condition true successor or condition false successor.
- if $L(s_1)$ is a return statement, then $L(s_2) = END\_FUNCTION$.

As we have shown before the interrupt-driven program consists of one task and many interrupts. We use $Func = task \cup Irp$ to represent the set of task and interrupts where Irp represents the set of all interrupts. Task is the normal routine, while interrupts are some emergency events. If task is interrupted by some interrupts, then the system scheduler will hang up the task and assign the processor to the interrupt processing function. The system allows interrupt nesting. Because we can't determine when and where the interrupt will come, we make a conservative assumption that the interrupt may come after every instruction of task or interrupts. This assumption will ensure that we will not ignore any possible program execution paths. We can can describe interrupt-driven programs' execution states based on the CFA. Formally, the semantic of interrupt driven program execution is a 5-tuple $M = \langle S_{MI}, S_{IS}, I_{IS}, T_{IS}, L \rangle$ where:

- $S_{MI} = S_{CF} \cup \{INIT\}$ is the set of all global control states.
- $S_{IS} \subseteq S_{MI} \times S_{MI} \cdots \times S_{MI}$ is the set of interleaving execution states where the number of S_{MI} is $size(Func)$.
- $I_{IS} \in S_{IS}$ is the initial interleaving states.
- $T_{IS} \subseteq S_{IS} \times S_{IS}$ is the set of transitions between the interleaving states.
- $L : S_{CF} \setminus \{INIT, END\_FUNCTION\} \rightarrow Stmt^{\star}$ is the label function where $Stmt^{\star}$ is the set of program expressions.

S_{MI} consists of the global states including the control states of task and interrupts. T_{IS} is the set of transitions of interleaving control states. We can give the definition of transition conditions for T_{IS} similarly as that T_{CF}. $(S_1, S_2) \in T_{IS}$ iff one of the following condition holds. Let $size(Func) = n$, $S_1 = \langle s_{10}, s_{12}, \ldots, s_{1n} \rangle$ $S_2 = \langle s_{20}, s_{22}, \ldots, s_{2n} \rangle$ where s_i is the priority level ranked as i-th interrupts.

- $\langle s_{1i}, s_{2i} \rangle \in T_{CF}$, if $s_{1i} \neq INIT$ and for $\forall j.j < i$ we have $s_{1j} = s_{2j}$, for $\forall k.k > i$ we have $s_{1k} = s_{2k} = INIT$ where $i, j, k \in [1, n]$.
- If $L(s_{1i})$ is function call expression and s_{2i} is entry point of the function call, then for $\forall j.j < i$ we have $s_{1j} = s_{2j}$, for $\forall k.k > i$ we have $s_{1k} = s_{2k} = INIT$ where $i, j, k \in [1, n]$.
- $\langle s_{1i}, s_{2i} \rangle \in T_{CF}$, if $s_{1i} = INIT$ and for $\forall j.j < i$ we have $s_{1j} = s_{2j}$, for $\forall k.k > i$ we have $s_{1k} = s_{2k} = INIT$ where $i, j, k \in [1, n]$.

These conditions show that task or interrupts execute one step, task or interrupts call other function, task or interrupts are interrupted by interrupts, respectively. At the same time, these conditions satisfy that if the highest priority level interrupt is executing then the task and other interrupts will not be executed.

In order to find all data races in program, we need to consider all possible interleaving states. We assume that interrupts may come after every instruction of task or interrupts. The complexity of the interleaving states S_{IS} space is: $size(S_{IS}) = size(S_{F_1}) \times size(S_{F_2}) \cdots \times size(S_{F_n})$ where we assume that $size(Func) = n$ and S_{F_i} is the number of control states of the rank i-th priority level interrupt. We notice that the state space of interleaving states S_{IS} may be very large when interrupt-driven programs consist of too many interrupts. In fact, we will not store all these interleaving states, so the state space wouldn't cause data race check failure because of memory out. However, the scale of state space will affect the efficiency of data race detection.

On the other hand, we notice that these interleaving states may contain some unreachable states in real program execution, such as the exclusive branch condition in task and interrupts. However, the generation of interleaving states doesn't take this condition into account, so the resulting of interleaving states include these unreachable states. These unreachable states may cause false alarms for data race detection. This is because the method of the generation of interleaving states is path insensitive. We will introduce a false alarm elimination method based on bounded model checking in Section. IV-C.

IV. Data Race Detection Method

Our data race detection method is based on analyzing accessing relationships of shared variables. This detection method can be divided into three steps: interleaving states generation, potential race detection and false alarm elimination. Interleaving states generation which is depicted in Section. III is to get the whole program interleaving states. In

order to simplify this problem, we firstly convert concurrent program to sequential program. After the conversion we collect the interleaving states based on the serialized program. Race detection method is that we firstly define a data race violation rule and then we explore the interleaving state space to find the violation states. False alarm elimination can be achieved by coding the path condition of data race states and then judging whether the path condition can be satisfiable. The first two steps can be done together. When we get the data race states, we covert the program to SSA and collect the path condition. The relationship of the these steps is shown in Fig. 2.

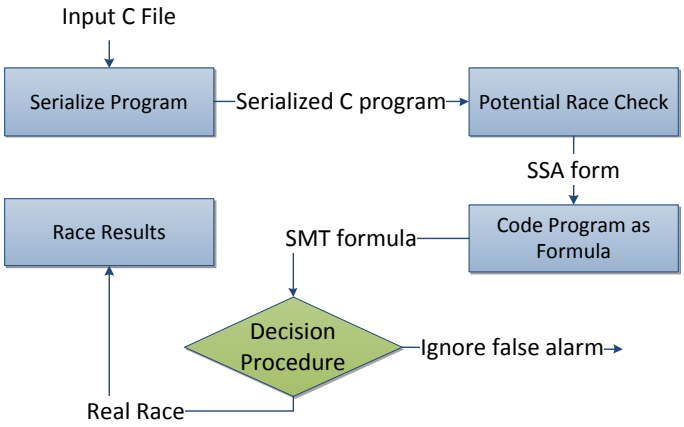

Fig. 2: Data Race Detection Procedure

A. Convert Concurrent Programs to Non-deterministic Sequential Programs

In order to simplify the race detection and make it easy to generate the interleaving state space we convert interrupt-driven programs to non-deterministic sequential program. The main idea of serializing procedure is adding non-deterministic branches after every instruction. The branch condition is an unknown value, so that when we coding the program as formula we will consider both the branches. The true branch of the added branches is the function call to the interrupt and the false branch is empty. If there are more than one interrupt then we need to add many this kind of branches. At the same time, we should note that we not only add the branch to the task but also add the branch to interrupts (if there more than one interrupt). The method of adding branches in task and interrupts is almost the same. In the task and interrupts we should add the interrupt which has the priority level next to the current task or interrupt. This means the highest priority interrupt need not to add any branch. This process is shown in Fig. 3 where we assume that *brandom* is an unknown value, $size(Irp) = n$ and *INTi* represents the function name of the i-th ranking priority interrupt.

From the serialization procedure we can see that the serialization process contains all the possibility of interrupt driven programs' execution. In the task we add the ranking first interrupt shows the task can be interrupted by any interrupts. While in an interrupt we only add the interrupts which has higher priority than the current interrupt, which shows the interrupt only can be interrupted by the interrupts of higher priority.

```
1  /*main task*/
2  void TASK(void){
3     ...
4     /*after every instruction*/
5     if(brandom) INT_1();
6     ...
7  }
8  ...
9  /*interrupt*/
10 void INTi(void){
11    /*at the beginning of interrupt*/
12    if(brandom) INTi+1();
13    ...
14    /*after every instruction*/
15    if(brandom) INTi+1();
16    ...
17 }
```

Fig. 3: Convert Concurrent Programs to Sequential Ones

When adding the branch to the task function, we may find that we have to take special consideration for function calls. Because in S_{CF} the successor of function call expression is the successor in the control flow, but in S_{IS} the successor of the function call is the entry point of the function. From the perspective of analysis we also need to add the branch to the expression of function body, because data race may be happened in function call. In order to simplify the analysis procedure and overcome the previous problem, we inline all the function calls. In fact in order to make the following race detection and false alarm elimination simple, we inline the whole program in three steps Firstly we inline the program of the task and all the interrupts, Secondly we add the branch to the task and interrupts, Lastly we inline the task procedure. Then we get the serialized program with only assign and condition goto expression.

B. Race Detection In Sequential Programs

We detect data race based on shared variables by exploring the interleaving state space. The data race violation happens iff one of the following condition holds:

- If the read variables set of current task or interrupts is written by other interrupts simultaneously .
- If the write variables set of current task or interrupts is accessed other interrupts simultaneously.

We focus on interrupt-driven program executing on single core CPU, which in fact is a serialization execution process.

Now we can give the race detection procedure. We define *Variables* as the set of all the shared variables, for any expression $e \in Stmt$, $var(e)$ represents the set of read and write variables where $var : Stmt \rightarrow 2^{Variables}$. For any control states $S \in S_{CF}$, $v\_read(S)$ and $v\_write(S)$ represent the set of read and write respectively where $v\_read, v\_write : S_{CF} \rightarrow 2^{Variables}$. Because we have serialized the program which convert the program into the one with only assign and condition goto expression, the read and write set of every

expression can be acquired easier. We can give the data race definition as follows.

$$data\_race : S_{IS} \rightarrow Bool$$

$data\_race(S) = true \Longleftrightarrow \exists i, j \in [1, n], i \neq j, (r_i \cap w_j) \cup (w_i \cap w_j) \cup (w_i \cap r_j) \neq \emptyset$

where $S \in S_{IS}, S_i \in S_{CF}, S = \langle S_1, S_2, \ldots, S_n \rangle$, $r_i = v\_read(S_i)$, $w_i = v\_write(S_i)$, $r_j = v\_read(S_j)$ and $w_j = v\_write(S_j)$.

In fact we detect data race when we convert the program into a serialization program. When finding interleaving states satisfy the data race violation condition, the analyzer records the violation state. The algorithm of our data race detection is depicted in Algorithm.1 where S_{MI} is the control states of the whole program, ipt_state is a list of interleaving states.

Algorithm 1 Race Detection In Sequential Programs

Initialization:
Set $S = S_{MI}$; $ipt\_state = [\ ]$; $ipt\_flag = false$;
$it = begin(S)$; $pre\_it = it$;
1: **while** $it\ != NULL$ **do**
2: **if** $Is\_int\_begin\_flag(it)$ **then**
3: $ipt\_flag \leftarrow true$;
4: $Push\_back(ipt\_state, pre\_it)$; Continue;
5: **end if**
6: **if** $Is\_int\_end\_flag(it)$ **then**
7: **if** $Is\_empty(ipt\_state)$ **then**
8: $ipt\_flag \leftarrow false$;
9: **end if**
10: $Pop\_back(ipt\_state, pre\_it)$; Continue;
11: **end if**
12: **if** $ipt\_flag == true$ **then**
13: **if** $Is\_race(ipt\_state, it)$ **then**
14: $Add\_potential\_flag(it)$;
15: **end if**
16: **end if**
17: $pre\_it \leftarrow it$; $it \leftarrow next(it)$;
18: **end while**
19: **return** S;

When data race detection is finished we transform programs to static single assignment (SSA) which will simplify coding the program as SMT formulae. There are many efficienct and effective methods for converting program to SSA such as [13][14] and we will not discuss this problem here.

C. False Alarm Elimination via Bounded Model Checking

There are many different bounded model checking algorithms in [15][16]. In this section we only consider how to use the idea of BMC to eliminate false alarms of race detection.

Before we code the program as formulae, we still have one important problem to solve. That is how to deal with loops. Though the program only consists of assignment and condition goto statements, the conditional goto may go back to a previous program point which means that it may be a loop. The method we adapting is loop unwinding. The main strategy is as follows:

- If user has set the max unwind, then either we unwind the loop as the max unwind when the loop condition always hold, or we unwind the loop until the loop condition does not hold.
- If user doesn't set the max unwind, then we unwind the loop until the loop conditions does not hold.

Loop unwinding is an important method to simplify the complexity of the problem in this paper. Because the complexity of software, software bounded model checking always has the problem of scalability. In the experiment, we find that when the program consists of many loops, the data race detection method may be failed due to memory out. So we may always set the max unwind number except the program is simple.

After dealing with loops, we can begin to convert the program to formula. In fact, there are many method to code the program as formula in [17][16]. We use the method described in [17], since the syntax form of our program is simple, it is easy to adopted this method in our program. Because the program has been serialized and transformed as SSA we only need to consider assign and branch condition. The branch expression can be changed to a conditional operator, the assign expression can be still as an assign formula. This process can be depicted in Fig. 4.

Once programs are encoded as SMT formulae, we can use these formulae to eliminate false alarms. Firstly we get all the potential race program points from the previous section, then we can get SMT formulae from the beginning of the program to the potential race program point. Secondly we use a SMT solver to solve whether these formulae can be satisfied. Lastly we eliminate false alarms of potential race points by the results of SMT solver.

V. Implementation and Experiments

The data race detection method from Section IV is implemented one top of the analyzer CBMC for Ansi-C [17] [18]. The analyzer CBMC is a Bounded Model Checker for ANSI-C programs. It also supports SystemC using Scoot front-end. It can be used to verify array bounds, buffer overflows, pointer safety, exceptions and user-specified assertions. Furthermore, it can check the consistence of ANSI-C with other languages, such as Verilog. The verification procedure of CBMC is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure. The aim of this analyzer is for embedded software, it also supports dynamic memory allocation using *malloc* and *new*. Because of CBMC taking path conditions into account the analysis may exclude some unreachable execution paths and therefore may raise less false alarms.

Our test suite consists of sample programs from our own examples together with the Goblint implementation [12]. Because the benchmark of Goblint implementation is from the nxtOSEK[19] and the programs' syntax of which is a little different with standard C programs, we convert the program to C under the premise of keeping program semantic unchange. The main differences between nxtOSEK programs and C programs lie in that some of the key words are different and nxtOSEK programs support synchronization via resource competition. For the first difference, we change the key words of nxtOSEK program to the equal semantic key words of C programs. For the second difference, we use a global variable

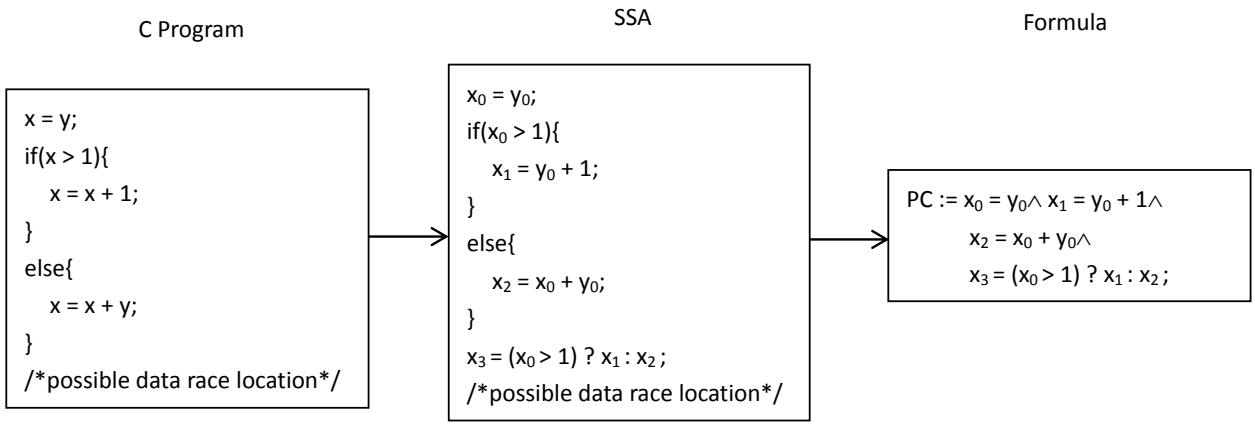

Fig. 4: Example of Converting Program to Formula

to represent the resource, and the get and release operations of resource become test and set of the shared global variable. This is a simple way to rewrite the program, but many of the program properties have been changed, such as the priority of the resource can't represent. However, we don't change the read and write relation between task and interrupts which is our main focus.

Program posture_control is part of the control software of a kind of aerobat. posture_control has a task which is dealing with the normal procedure and a interrupt to process the exception of aerobat posture. The rest programs are all from Goblint examples which are converted to C and renamed by adding a prefix "C_" before their original name. Each of these programs consists of one task and multi-interrupts. Program C_privative is an example for data race between task and interrupts, and the interrupt uses a soft lock-set to mutually access share variable. Programs C_example and C_example_fun are examples of [12], and these two programs are examples for improper using synchronization variable and inter-procedure data race checking respectively. Program C_privatintervals which consisting of one task and three interrupts is an example for more than two interrupts data race situation. Program C_suffix consists of an task which initials two shared variables and stores the result of adding the two shared variables. The initial and store procedure are protected by different mutex-lock and interrupts which modify the shared variable and may cause data race under some condition.

The results of running the analyzer on these programs are shown in TABLE I. For each program, the column "Size" gives the lines of the programs, "Time(s)" presents the analysis times which takes average of 10 times analysis and the unit of time is second, "Int" gives the number of interrupts and "VCC" gives the number of verifying conditions during bounded model checking. We ran these experiments on a Intel(R) Core(TM)2 i5-2320 CPU machine with 3.00GHz and 2GB memory under Fedora 12. The decision procedure of CBMC is Minisat 2.2.0.

After the analysis, we find that all programs have data races. For posture_control, because this program has many loops, most of the loops have a deterministic boundary, such as 256. The CBMC takes loop unwinding to deal with loop as we have mentioned before. At first we unwind all the loops without setting unwind depth, the analyzer ran about 10 minutes and threw a memory out exception. So we set the max unwind depth to 10, then the analyzer can finish race detection. The data races in posture_control program are caused by the following situation during the normal routine, task may access the global timer to get current time or reset it, while on the other hand during the exceptional routine interrupt may change the aerobat posture at some time interval and the interrupt also needs to modify the global timer. This causes data race happening. The data races in C_privative program are caused by improper synchronization via mutual variable. The task uses a proper synchronization variable to protect the shared variable, but the lower priority interrupt uses one synchronization variable, while the higher priority interrupt uses another variable to protect the accessing of shared variable. It causes data races between interrupts. The data races in both versions of the example program are discovered. The race warning in C_privatintervals occurs because the synchronization variables are different between the interrupts and there is data race between interrupts. For C_suffix, race warnings are produced for the task getting the sum of the shared variable, which may race with the interrupt. The reason why the posture_control consumes so much time compared with the others is that the other program contains fewer loops and the size is not very large, but posture_control consists many loops which will cause the state space growing rapidly.

Program	Size(loc)	Time(s)	Int	Race	VCC
posture_control	169	551.247	1	42	395
C_privative	36	0.164	2	6	12
C_example	38	0.169	2	3	9
C_example_fun	51	0.186	2	7	12
C_privatintervals	51	0.191	3	4	23
C_suffix	40	0.177	2	3	9

TABLE I: Experimental Results for Benchmark Examples

VI. Conclusion

We have presented an approach in the framework of bounded model checking for analyzing data race for interrupt driven programs. The main idea is to use accessing relations over shared variables to detect potential data race and use path condition to eliminate some false alarms. The data race detection algorithm based on shared variable is implemented by exploring the serialized program CFG. The algorithm analyzes each interleaving control states to find the violation of shared variable accessing rule. This method is a lightweight way to detect data race but may cause many false alarms. Thus, on this basis, we gather the path conditions of all the possible race locations, then we use SMT solver to decide whether the path condition is satisfiable. The path condition is unsatisfiable means that the data race execution path is infeasible in real execution and the potential data race with this path condition is false alarm. A key benefit of our approach is the ability to check data race for interrupt driven programs path sensitively.

Future work will consider two aspects to improve our race detection method. On one hand, we want to extend the scalability our approach to reason large scale programs. Because shared variables are what we focus on during data race detection, we can use code slice according to shared variables to reduce the code size of the analyzed program. On the other hand, we want to improve the efficiency of the analyzer. We can use inter-procedure optimization method such as function summarization to improve the efficiency of our race detection method.

References

[1] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Practical static race detection for c. *ACM Trans. Program. Lang. Syst.*, 33(1):3:1–3:55, January 2011.

[2] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 1–13, New York, NY, USA, 2004. ACM.

[3] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2:366–381, 2000.

[4] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[5] Xinwei Xie and Jingling Xue. Acculock: Accurate and efficient detection of data races. In *Symposium on Code Generation and Optimization*, pages 201–212, 2011.

[6] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and LenoreD. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer Berlin Heidelberg, 2005.

[7] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. In Antti Valmari, editor, *Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*, pages 146–162. Springer Berlin Heidelberg, 2006.

[8] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[9] V. Vojdani and V. Vene. Goblint: Path-sensitive data race analysis. *Annales Univ. Sci. Budapest., Sect. Comp*, 30:141–155, 2009.

[10] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[11] Rui Chen, Xiangying Guo, Yonghao Duan, Bin Gu, and Mengfei Yang. Static data race detection for interrupt-driven embedded software. In *Secure Software Integration Reliability Improvement Companion (SSIRI-C), 2011 5th International Conference on*, pages 47–52, June.

[12] Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, Peter Lammich, and Markus. Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In *POPL'11*, pages 93–104, 2011.

[13] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16(6):1684–1698, November 1994.

[14] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. volume 58 of *Advances in Computers*, pages 117 – 148. Elsevier, 2003.

[16] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34, 2001.

[17] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[18] Daniel Kroening et al. The CBMC homepage, 2013. http://www.cprover.org/cbmc/.

[19] Takashi Chikamasa et al. OSEK platform for MINDSTORMS, 2010. http://lejos-osek.sourceforge.net/.