Modelling race conditions in multithreading programs in terms of Petri nets

D I Kharitonov and D S Odyakova
Institute of Automation and Control Processes Far Eastern Branch of the Russian Academy of Sciences (IACP FEB RAS), 5, Radio st., Vladivostok, 690041, Russian Federation
Email: demiurg@dvo.ru, darlene@dvo.ru

Abstract. The article describes an approach to detecting race conditions with respect to a variable in a program, based on the automatic construction of a program model in terms of Petri nets. A simple sample program having race condition is given and the analysis of its behavior shows four different results of execution due to the race condition. Compositional model of the program in terms of Petri nets is presented that illustrates the process of automatic model generation. Notions of program control flow, global area and environment models are introduced. Reduction rules for the control flow model are defined and compositional representation of reduced program model is described. The formula for detection of race conditions is defined as: for a race conditions to occur, it is necessary that between the moment a variable is read by a thread and the moment this thread writes to a variable, an event of writing to the variable by another thread may occur. Compositional model of global variable capable of intercepting writing to variable events that lead to race condition is presented. By synchronizing the program control flow model with the global variable model it is possible to detect race conditions by building the reachability tree and checking for the presence of error event in it.

1. Introduction
Sir Tony Hoare has claimed that “the construction and application of a verifying compiler that guarantees correctness of a program before running it” is a grand challenge of 21st century [1]. One of the problems the verifying compiler should deal with is a race condition. In software, a race condition means a situation where the correct execution of a program depends on the sequence or time of execution of processes or threads [2]. In most cases, race conditions occur when processes or threads depend on some common variable, and in this sense can be referred to as data races (though it is not strictly follows the definition of data race). The incorrect behavior of the program due to the race condition is difficult to reproduce and debug [3], because it depends on the relative synchronization between concurrent threads, having non-deterministic delays in the execution. Sometimes the consequences of the race condition appear only after a long period of time and in a completely different part of the application. At the same time, problems that arise in the software can disappear when executing program in debug mode, saving debug information to log files or connecting an external debugger. Attempts to cope with the problem of race condition are carried out at different levels. There are examples of approaches to detecting race conditions by analyzing program statically and executing program dynamically [4], in addition there are approaches offering to think about the
race conditions already at the design stage [5]. This article discusses an approach to detecting data races based on the construction and analysis of a program model. In this research, a completely automatic method for constructing a model in terms of Petri nets based on syntactically controlled translation is considered.

2. A sample program containing race condition.
To build a minimal example of a program containing race condition is quite simple. It is required two processes to read and write to the shared variable the same time. Let’s call such a program RCmin abbreviating the term race condition.

```c
#include "stdio.h"
#include "pthread.h"

int a;

int main()
{
    int res;
    pthread_t tId;
    a=0;
    res=pthread_create(&tId, NULL, Thrd2, NULL);
    if (res==0)
    {
        Thread1();
        pthread_join(tId, NULL);
        printf("result is %d\r\n", a);
        return res;
    }

    void *Thread1(void*arg)
    {
        int b;
        b=a;
        a=b+1;
        printf("1st a=%d\r\n",a);
        return NULL;
    }

    void *Thread2(void*arg)
    {
        int b;
        b=a;
        a=b+2;
        printf("2nd a=%d\r\n",a);
        return NULL;
    }
```

Figure 1. Source code for RCmin containing race condition.

Figure 1 shows the RCmin source code. This program consists of a global scope, the main function and two simultaneous executable functions Thread1 and Thread2. In the global area, space is allocated for a single global variable a. The main function initializes this variable, launches the Thread1 and Thread2 functions in two threads, waits for the end of their execution, and displays the value of the variable a on the console. The Thread1 function reads the variable “a”, increments it, and prints the result to the console. The Thread2 function reads the variable “a”, adds two to it, and prints the result to the console.

```
$ ./Tests
first a=1
second a=2
result is 2
```
```
$ ./Tests
first a=1
second a=2
result is 1
```
```
$ ./Tests
second a=2
first a=1
result is 1
```
```
$ ./Tests
first a=1
second a=3
result is 3
```

Figure 2. Options for the results of the RCmin program.

The RCmin program contains the data race, because each of the threads read the variable “a” and write to it sequentially. This means that there is a non-zero probability that between reading and writing to the variable the thread execution will be interrupted by the operating system thread scheduler and the same time there will be a competitive operation of writing to the variable by another thread. Since the RCmin program is very short, and does not contain a significant CPU load, the probability of such thread interruption is very small. To increase this probability, it is enough to insert the usleep(0) function call between reading and writing. It should stop the thread execution for a given time period, however, with zero-time interval it simply transfers to the thread scheduler the ability to switch the execution context to another thread. Figure 2 shows four different versions of the RCmin program execution, augmented with usleep(0) calls, which were obtained in just 15 runs. It can be
seen from the figure that due to the “voluntary” suspension of thread execution, not only the final value of the global variable changes but also the order of thread execution completion. This is a consequence of the fact that a large number of factors independent of the RCmin program affect the operating system work and, thus, determine the execution order of the threads. As a result, the choice of the next thread by the operating system can be considered as a random function.

3. A compositional model of the program.

The construction of a program model in an automated way is performed on the base of an abstract semantic graph (ASG). The main advantage of ASG in comparison with the result of source code parsing is its significantly concise representation, which does not directly depend on changes in the programming language grammar that ensures a longer life cycle of all the algorithms developed. The ASG graph of the program elements is converted into a model in three stages. The first one is the construction of compositional Petri nets of the control flows for program elements. For this, substitution of template Petri nets corresponding to the constructions of the programming language into the nodes of the graph is made. The first stage is performed by procedure for traversal of an abstract semantic graph, during which a visit to the node also generates operations of composition by places between the template Petri nets. In addition, Petri nets modelling control flows are labelled with access points by transitions, through which synchronization between function calls and the net modelling the function itself should be performed. The resulting compositional Petri net describes the “control transfer” relationships between simple components of the control flow model. At the second stage, all composition operations by places are performed that gives the Petri net, which is the control flow model of a particular program element.

At the third stage, in order to obtain a general program model, it is necessary to model the interaction between various program elements and the environment in which the program is to be executed. For such synchronization, a specially developed operation of directional composition is used [6]. In this operation outgoing and incoming access points to Petri nets are distinguished. An outgoing access point is used in the nets with function calls, and incoming access points describe call interface in the function model. During the operation of two nets directional composition, a set of synchronization transitions incident to places in each of the nets is formed. Then, the initial transitions labelled with the outgoing access point are deleted from the resulting net, but not ones labelled with the incoming access point. Thus, the net can synchronize once through the outgoing access point, and multiple times through the ingoing access point. To model the interaction of program elements at the third stage, a search of access points with compatible identifiers is made and synchronization operations between nets with these access points by transitions are added. The compositional representation of the RCmin program model after the third stage is shown in Figure 3. In the RCmin program considered, the models of next elements are distinguished: the functions main, Thread1 and Thread2, the global area and the environment the program is to be executed in. All nets except the Environment can be built in an automated way from the program source code. The Environment net
should describe the program loader, operating system functions and library functions without source code provided, i.e. printf, pthread_create, pthread_join functions.

4. Reduced program model
The data race in the program occurs in the Thread1 and Thread2 functions, taking this fact into account the program model can be refined and simplified without compromising the analysis process. To do this, program model reduction rules in the Figure 4 can be used.

![Figure 4. Program model reduction rules.](image)

The upper part of the figure on the left shows the function call of a simple function without visible actions. In the calling net, it is modeled by transition-place-transition triple [7], where the transitions indicate events of the beginning and end of the function call, and the place – the waiting state for the function to execute. In the called net, such triple describes the execution of a function. After the first arrow, the figure shows the Petri net obtained as a result of synchronization of these two nets (note that instead of transition labelling with access points, a simple labelling is used). When analyzing the behavior of a sequential program, the events of the beginning and end of the function will always occur one after another. However, when analyzing multi-threaded programs, they can be scattered in the program execution trace, generating redundant states in the model that actually do not affect other program properties. Therefore, the first reduction rule allows to shrink the template constructions like in the resulting net to a single transition, as shown at the top right in the figure. The bottom left part of the figure shows the resulting net with a call to a more complex non-recursive function. The net upper place models the waiting for the end of function execution at the call point in the program and does not provide any additional information when analyzing the program behavior. Thus, the second reduction rule allows to reduce this place, as shown in the figure below to the right. The third rule of reduction does not need an illustration. It states that incoming access points that will no longer participate in composition operations along with transitions labelled with these access points should be removed from Petri nets.

Figure 5 shows a shorter model of the program obtained from the model in Figure 3. To obtain it, a model of environment was built “manually”, including a minimal loader and primitive models of functions printf, pthread_create, pthread_join. After that, composition operations were performed between the Petri nets modeling control flows of the functions main, Thread1, and Thread2, and the Environment model. Then, in the resulting Petri net, the reduction procedure was performed, according to the rules described earlier, after which each call to the printf, pthread_create, pthread_join functions is represented by a single transition in the net. The resulting net shown after cosmetic editing in the figure is called Program control flow. In order to reproduce the race condition in the model, the net has transitions for the events of variables reading and writing. It should be noted that the notation of colored Petri nets [8] was used to model control flows. In this case, the only color is used to indicate the thread identifier of the control flow. In fact, it can be considered equal to one for the first thread (t1) and two for the second (t2). Besides this net, the figure also shows nets modeling the internal function variables, and the net simulating the global area of the program, consisting of a sole variable “a”.
Since the internal function variables cannot cause a race condition, further, when analyzing their models, they can simply be removed from the program model, along with labelling of transitions by access points through which synchronization with the program control flow takes place. So the only net modeling a global variable remains unconsidered. This net is to be built “manually” in a way to make the race condition event explicit.

5. Global variable model

To begin with, it should be noted that reading a variable cannot cause a race condition, while writing to a variable can. The race condition happens when a thread (or process) that writes to a variable has incorrect information about the state (value) of this variable. Let’s formalize the condition for the occurrence of race condition when handling a single variable. For a race condition to occur, it is necessary that between the moment a variable is read by a thread and the moment this thread writes to the variable, an event of writing to the variable by another thread may occur. This is a necessary condition for the occurrence of a race condition, since the negation of this statement guarantees the absence of a race condition. However, this is not a sufficient condition, since in some cases, threads can write to a common variable concurrently and this is the normal behavior for the program. For example, threads can record the time of the last activity of some network service in a global variable, and simultaneous writing to a variable from different processes can correct the recorded time for the shortest fractions of seconds, which is not critical for the program. Thus, the necessary condition can be checked in automatic mode, but it is a person to decide about the existence of a race condition.
The formula of a race condition shows that non-deterministic behavior of the program due to the concurrent execution of its threads takes place. It does not matter how many times a variable has been changed between its reading and writing by the thread under consideration, to detect the race condition – it is enough to catch just one of the recording events. Figure 6 shows a compositional model of a global variable that uses the idea of an alternative bit protocol [9] to intercept such a recording event. Instead of the concrete value of the variable “a”, the model can save its state in one bit, and each time the variable “a” is written, its state changes from zero to one and vice versa. Then, instead of monitoring the value of the variable, it will be enough to control the change in its state, reflecting the very fact of the value changing. It is extremely difficult to control what information about variable state the writing thread possessed at the level of the control flow model in terms of Petri nets, since this requires maintaining the value of the variable in all places the thread can be. Instead, it is possible to save separately the state of variable known to each of the threads and check this correspondence at the time of writing. The first part of the model in Figure 6 is called Interface of “a”, it defines the events of variable initialization and deletion, read and write events, as well as an additional event of a new thread initialization. All transitions in this net are labelled with external “e” and internal “i” access points. An external access point is used to communicate with the program control flow model and its labelling uses thread identifiers and variable values. The internal access point uses the single-bit state of the variable instead of its value. This access point is used to synchronize with the second part of the model in Figure 6 called Thread mem of “a”. In this net, the correspondence between the thread identifier and the last state of the global variable known to this thread is preserved. In addition, the writing event in this part is labelled with an access point for synchronization with the third part of the model represented on the figure by the Check net. The writing event is parameterized simultaneously by the current and last known to the thread states of the variable, which allows the Check net to compare these values and determine whether the race condition exists (the transition is labelled as RCerror). To detect the race condition, it is necessary to synchronize the Program control flow with the Global area model and build the reachability tree of the resulting Petri net. If there are events labelled as RCerror in the reachability tree, then the program meets the necessary conditions for the data race.

6. Conclusion
The article describes an approach to checking the presence of the necessary conditions for a race condition with respect to a variable in a program, based on the automatic construction of a program model in terms of Petri nets. The approach can be used both to verify the correct handling of each variable individually, and to simultaneously check multiple variables. The considered approach shows how the automatically generated control flow model can be used to search for errors in the program, since instead of a variable model focused on searching for race conditions, other types of variable models can be used, for example, those aimed at searching for incorrect handling of pointers or memory leak errors.

Acknowledgment
Financial support provided by the state-funded theme of the Ministry of Education and Science of the Russian Federation No. AAAA-A17-117040450019-8 is greatly appreciated.

References
[1] Hoare T 2003 The Verifying Compiler: A Grand Challenge for Computing Research In: Modular Programming Languages. Lecture Notes in Computer Science eds. Böszörményi L and Schojer P 2789 (Berlin, Heidelberg, Germany: Springer)
[2] von Praun C 2011 Race Detection Techniques. In: Encyclopedia of Parallel Computing eds. Padua D (Boston, MA, US: Springer)
[3] Chiu Y-C, Shieh C-K, Huang T-C, Liang T-Y and Chu K-C 2011 Data race avoidance and replay scheme for developing and debugging parallel programs on distributed shared memory systems. *Parallel Comput.* **37**(1) 11-25

[4] Choi J-D, Lee K, Loginov A, O’Callahan R, Sarkar V, Sridharan M (June 2002) Efficient and precise datarace detection for multithreaded object-oriented programs. In: Conference on programming language design and implementation (PLDI’02), pp 258–269

[5] Chernenok and S A Nepomniaschy V A 2015 The Application of Coloured Petri Nets to Verification of Distributed Systems Specified by Message Sequence Charts *Proc. Spring/Summer Young Researchers’ Colloquium on Software Engineering 2015* 3

[6] Kharitonov D I, Tarasov G V and Golenkov E A 2017 Modelling of object-oriented programs with Petri net structured objects *Computing and Informatics* **36** 1063–1087

[7] Kharitonov D and Tarasov G 2014 Modeling function calls in program control flow in terms of Petri Nets *ACSIJ Advances in Computer Science: an International Journal* **3**(6(12)) 82-91

[8] Jensen K and Kristensen L M 2009 Coloured Petri Nets: Modeling and Validation of Concurrent Systems (Springer) p 384

[9] Feijen W H J and van Gasteren A J M 1999 The Alternating Bit Protocol *In: On a Method of Multiprogramming* (New York, NY, US: Springer)