Modeling imperative programs operations on memory in terms of Petri nets

Dmitry Kharitonov¹*

¹Institute of Automation and Control Processes Far Eastern Branch of the Russian Academy of Sciences, 690041 Vladivostok, Russia

Abstract. The article discusses an approach to automatical creation of imperative programs models, designed to find errors in memory operations. The approach is based on the model division into compositional parts, including control flow models, variable, data types and pointers models. Models of data types and pointers are developed from the purposes of modeling to a proper level of detail sufficient to detect errors. To analyze the correctness of a program, a reachability graph of the model Petri net is constructed, on which deadlocks, loops, and explicitly marked erroneous events are searched. The article provides an example of a program with an error in accessing a previously freed memory block in a race condition state of parallel program threads. A model constructed for the example program in terms of Petri nets allows to track the moment an error occurs in the reachability graph.

1 Introduction

Software is the product of the joint work of a large number of different specialists. At the same time, despite real-life examples of long-lived programs, it can be stated that the life of the vast majority of programs at present is limited to five, maximum ten years. This state of affairs is caused by the constant variability of the electronic world, in which the evolution of processors, input-output devices, communication networks, operating systems, languages and programming tools occurs. As a result, it can be declared that currently programs mostly unreliable, and it is seen in the form of various freezes, program crashes, and excessive consumption of system resources. One of the main sources of this behavior is errors in memory operations, including [1]:

- access beyond the boundaries of the arrays (buffers);
- use of dynamic memory after it is freed, including:
  - accessing a local object after returning from the function where the object was defined;
  - accessing a local object outside its scope;
- re-release of once freed memory;
- initialization of global variables in the wrong order;
- loss of pointers to allocated memory (memory leaks).

* Corresponding author: demiurg@dvo.ru

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).
Finding such errors through testing and debugging is a complicated routine process, that depends on many factors, including input data, the presence of simultaneously running programs, and the speed of various computer subsystems [2]. In search of suitable formal means for finding errors in data operations, graph theory [3], SMT solvers for symbolic interpretation [4-5], and template description of data structures [6] are used. Despite the presence of an impressive list of program analysis tools, it must be admitted that no software tools yet exist that can guarantee the absence of errors in the source code of programs [7]. This article discusses an approach to automating modeling the behavior of imperative programs in terms of Petri nets, which allows to verify memory operations and catch errors in these operations. The first section describes an example of a program containing an error, then a compositional representation of the program model is provides along with Petri nets that model individual functional blocks of the program, than the process of constructing the state space of the program is considered and errors found are analyzed. Finally, a comparison of the proposed method with existing error detection approaches is provided.

2 Program example

Let’s consider an example program presented in the figure 1. In this program, the control function main creates a thread for the function Master and calls the function Slave. The Master and Slave functions interact with each other through two variables: DataPrepared and Value. The DataPrepared variable is a flag indicating that the Value variable contains data ready to be used. The Master thread generates data in a loop, sets the DataPrepared flag and waits in the so-called spinlock cycle until the flag is cleared. The Slave thread also waits in a loop until the DataPrepared flag is set, than does some useful work on that data, and resets the flag. In addition, this thread summarizes all processed data to obtain some final indicator.

```
1: int DataPrepared=0;
2: int *Value;
3: int main()
4: {
5:   pthread_t ThreadId;
6:   int err= pthread_create(&ThreadId, NULL, Master, NULL);
7:   if (err != 0) return 1;
8:   Slave();
9:   void *res;
10:   pthread_join(ThreadId,&res);
11:   return 0;
12: }
```

Fig. 1. Significant part of the example C program

The test example is deliberately designed in such a way that in the Slave function, lines numbered 8 and 9 are the cause of a possible memory access error. The source code is wrong because if after line 8 of the Slave function resets the DataPrepared flag, and line 9 of the Master function gets its time slice and frees the memory referenced by the Value data pointer, then the execution of line 9 in the function Slave will cause unpredictable behavior or even
The peculiarity of the above test program is that in the vast majority of cases this error will not be visible in any way during program execution. First, "deleting" a pointer does not mean that the memory it was referencing becomes unavailable to the process. In fact, accessing memory by pointer that has already been freed will result in an error only when the entire memory page within which the block was freed is marked as unwritable or completely “taken away” from the executing process. And secondly, the event when line 9 in the Master function is executed before line 9 in the Slave function is highly unlikely so that, in order to demonstrate this error in practice, it is necessary to introduce artificial delays in the execution of the program. At the same time, access to freed memory is a critical error after which the operating system stops executing the program. This means that the erroneous handling of memory reproduced in the example is, in practice, a rare and therefore difficult to detect error.

1.1 Building a program model

When modeling programs, it is necessary to take into account that the information about the program behavior necessary to build a model, is not fully represented by the source texts. An other essential part of the information for building a model is located, in the case of using the C language, in the so-called header files that describe the interface of the libraries in use. But these libraries can, in turn, lead to other libraries, or call functions of the operating system kernel, or directly access hardware, the behavior of which can only be learned from documentation. This means that for automatic construction of program models, sooner or later the source text of the computer-executed code will either be unavailable or will be completely absent. From this point of view, the automatic construction of a program model cannot be performed without a human description of some environment in which the program will be executed. Accordingly, the analysis of program behavior will be correct only when the description of the environment has been done correctly.

![Diagram](image.png)

**Fig. 2.** Reduced compositional model of the sample program

The basis of the program model automated construction is an abstract semantic graph obtained after parsing the source texts [8, 9]. Automation of construction is achieved by dividing the model into compositional Petri nets of different control flows, described in the program by functions and methods. For each function and method, a routine for traversing the abstract semantic graph is performed, during which visiting a node of the abstract semantic graph performs the substitution of template Petri nets corresponding to programming language constructs of each graph node, with the following generation of composition operations by places between these template Petri nets. In addition, Petri net models of control flows are marked with access points along transitions through which synchronization must be performed between each function calls and the net modeling the
function itself. The resulting compositional Petri net describes the "control transfer" relationships between the simple components of the complete program control flow.

![Petri Net Diagram]

**Fig. 3.** Model of the function *Master* in terms of Petri nets

The final compositional model of the program in terms of Petri nets should include models of all the functions and shared variables in hand, as well as models of the program loader and execution environment, which also includes models of data types and libraries used. Thus, the model of the program presented in Figure 1 should include models of functions *main*, *Slave*, *Master*, models of global variables *DataPrepared* and *Value*, models of data types *pthread_t*, *int* and models of pointers to data types *int* and *void*, as well as the model of the library *pthread*. The beginning and end of the work of such a program model are determined by the model of the *loader*, which initializes global variables, calls the *main* function, waits for its completion and finally deinitializes global variables. Wherein,
initialization of global variables in the model actually consists of placing the initial markings in the models of variables.

However, to analyze considered example program, it is possible to use a more concise model, consisting only of function models Slave, Master, global variables, data type int and pointer models. This shortened model of the example program is shown in Figure 2. In opposition to the full model, the beginning and end of the program are determined by the models of the Master, Slave functions, modified relatively to the full model by setting the initial markings. Upon these functions execution completion, the markings of nets Master and Slave become empty. Models of the Master and Slave functions in terms of Petri nets are shown in the figures 3 and 4. Transitions in the model nets are signed by program actions and can be easily mapped to lines of source text. Program operations with variables have been moved to labeling of of transitions with access points. At the same time, one transition can correspond to more than one labeling designed to synchronize either with global variables or with data type models. Labelings are parameterized, allowing positional parameters to be used to transfer information between compositional Petri nets.
As an example, consider the transition \( w9 \). The first labeling in it corresponds to the access point to the global variables model, while the value inside the labeling \( Value(id) \) is intended to obtain information about the memory cell number used for the variable \( Value \). The second labeling \( ptr\_int: get(id, p) \) corresponds to the access point to the model of a pointer to data type \( int \). The value of the labeling \( get(id, p) \) is intended to obtain the memory cell number \( p \) pointed to by the variable \( Value \); for this, the pointer model must have a token in which the number of the pointer cell coincides with number obtained from the first labeling. The third part of the transition \( w9 \) labeling - \( int: free(p) \) corresponds to an access point to the \( int \) data type, with a value signaling the \( int \) data type to free the memory cell with number \( p \). It should be noted that the above description is made in imperative terms, implying sequential execution of actions, while the implementation of the labelings performs all actions simultaneously, as if "selecting" a set of transitions that corresponds to this description.

Fig. 5. Petri nets modeling data types \( int \), a pointer to a data type \( int \) and global variables

Figure 5 shows three of the four Petri nets that model the environment and global variables in the example program. The \( int \) net models the \( int \) data type and is divided into two independent subnets. One subnet is for variables allocated on the stack, that is, automatically created local variables. This subnet uses place \( i1 \) to store the memory cell number and variable value. The second subnet, used for dynamic variables, uses the \( i2 \) place. The \( ptr\_int \) net models a pointer to the \( int \) data type, and is very similar to the \( int \) net, except that it lacks the atomic addition elementary action. The \( Globals \) network uses two places solely to store the cell numbers of global variables that will be used in nets that model operations over variables in imperative code through transitions labeling. There is no net in the figure that models a pointer to an undefined data type, since this network practically coincides with the \( ptr\_int \) net.

3 Building a program state space
all actions simultaneously, as if “selecting” a set of transitions that corresponds to this description.

![Petri nets diagram](image)

**Fig. 5.** Petri nets modeling data types `int`, a pointer to a data type `int` and global variables `DataPrepared` and `Value`

Figure 5 shows three of the four Petri nets that model the environment and local variables in the example program. The `int` net models the `int` data type and is divided into two independent subnets. One subnet is for variables allocated on the stack, that is, automatically created local variables. This subnet uses place `i_1` to store the memory cell number and variable value. The second subnet, used for dynamic variables, uses the `i_2` place. The `ptr_int` net models a pointer to the `int` data type, and is very similar to the `int` net, except that it lacks the atomic addition elementary action. The `Globals` network uses two places solely to store the cell numbers of global variables that will be used in nets that model operations over variables in imperative code through transitions labeling. There is no net in the figure that models a pointer to an undefined data type, since this network practically coincides with the `ptr_int` net.

### 3 Building a program state space
The program state space is builded in the form of a reachability graph in interleaving semantics, in which only one transition in a Petri net can be fired at a time. Interleaving semantics significantly reduces the number of arcs in the program’s reachability graph. Moreover, despite the possibility of parallel execution of the program code actions on modern multi-core processors, the critical part - the ability to write to a memory cell - most accurately coincides with interleaving semantics.

**Fig. 6.** Structure of the reachability graph of the example program model

Technically, to build a state space, it is necessary to transform the compositional representation of the program to a form consisting of one single Petri net and then build a graph of all possible sequences of transition firings in this net. However it is not possible to display the reachability graph in the form of a single figure within the framework of this article, but it is possible to step-by-step describe the graph having in mind the properties of the compositional model.

First, in all the component Petri nets of the built model, places are named with letters that make it easier to recognize their affiliation: \( m_i \) for places from the Master net, \( s_j \) – from the Slave net, \( i_1, i_2 \) - from the data type int model, \( p_1 \) – from the model of a pointer to data type int, etc. Second, in the constructed model, all operations of composition are performed via transitions, that ensure the number of places in the resulting network is preserved, since composition operations only change sets of transitions and arcs. In practice, the operations of network composition by transitions ensure synchronization of the nets, ensuring that only transitions marked by access points that have the same labeling can fire in networks. Taking this fact into account, in our case, obtaining the single net will not simplify the analysis process, but will only complicate the visual representation by increasing the number of arcs required to trigger synchronized transitions. Since in our model all nets are synchronized with the Master and Slave nets, this allows to build a reachability graph by considering the firing of transitions only in these networks and then selecting the firing of appropriately labeled transitions in the nets synchronized with them. And finally, thirdly, synchronization of the Master and Slave functions through a spinlock splits the reachability graph into a sequential chain of blocks in such a way that there is a connection between the blocks through a single transition. Figure 6 shows the structure of the reachability graph, where the connections between blocks are signed by iteration numbers in the function loops Master and Slave.
The program state space is built in the form of a reachability graph in interleaving semantics, in which only one transition in a Petri net can be fired at a time. Interleaving significantly reduces the number of arcs in the program's reachability graph. Moreover, despite the possibility of parallel execution of the program code actions on modern multi-core processors, the critical part - the ability to write to a memory cell - most accurately coincides with interleaving semantics.

Figure 6. Structure of the reachability graph of the example program model

Technically, to build a state space, it is necessary to transform the compositional representation of the program to a form consisting of one single Petri net and then build a graph of all possible sequences of transition firings in this net. However, it is not possible to display the reachability graph in the form of a single figure within the framework of this article, but it is possible to step-by-step describe the graph having in mind the properties of the compositional model.

First, in all the component Petri nets of the built model, places are named with letters that make it easier to recognize their affiliation: \( m_i \) for places from the Master net, \( s_j \) - from the Slave net, \( i_1, i_2 \) - from the data type int model, \( p_1 \) - from the model of a pointer to data type model int, etc. Second, in the constructed model, all operations of composition are performed via transitions, that ensures the number of places in the resulting network is preserved, since composition operations only change sets of transitions and arcs. In practice, the operations of network composition by transitions ensure synchronization of the nets, ensuring that only transitions marked by access points that have the same labeling can fire in networks. Taking this fact into account, in our case, obtaining the single net will not simplify the analysis process, but will only complicate the visual representation by increasing the number of arcs required to trigger synchronized transitions. Since in our model all nets are synchronized with the Master and Slave nets, this allows to build a reachability graph by considering the firing of transitions only in these networks and then selecting the firing of appropriately labeled transitions in the nets synchronized with them. And finally, thirdly, synchronization of the Master and Slave functions through a spinlock splits the reachability graph into a sequential chain of blocks in such a way that there is a connection between the blocks through a single transition. Figure 6 shows the structure of the reachability graph, where the connections between blocks are signed by iteration numbers in the function loops Master and Slave.

When displaying a reachability graph, the marking of the graph nodes begins with an ellipsis, which replaces the same beginning of the markings of absolutely all nodes of the graph: \( g_1^3 + g_2^4 \). This beginning of marking corresponds to: \( g_1^3 \) - the global variable DataPrepared with value 3 for the number of the used memory cell for the variable, and \( g_2^4 \) - the global variable Value with memory cell 4. Cell number 3 for the DataPrepared variable is also used in the int data type model at place \( i_2 \), taking the values \(<3,0> \) and \(<3,1>\), where 0 and 1 are the value the variable itself. Cell number 4 is used in the pointer to int model, in the place \( p_1 \), which takes values from \(<4,8> \) to \(<4,16>\), where the second element of the tuple is also the memory cell number and is used in the data type int model in the place \( i_1 \), taking a spectrum of values \( \{A, B, C, D, E, F, G, H, I\} \),

and as far as the actual values are not important from the point of view of analyzing the behavior of the program because they do not participate in statements that affect the control flow of the program, so they can be left in symbolic form.
Figure 7 shows the first part of the reachability graph for the example program model, in which the Master and Slave functions initialize local variables, enter the loop, reach and pass the first synchronization through the spinlock variable DataPrepare. In the Master function, a local variable \( i \) is initialized, which maps memory cell number 5 and value 1 in the place \( i_2 \) of the int data type model; the memory cell number then appears in all markings as the value of tokens in places \( m_i \). In the Slave function, local variables \( i \) and sum are initialized, which are mapped to memory cell number 6 with the value 1 and memory cell number 7 with the value 0 in the place \( i_2 \) of the int data type model, the memory cell numbers then appear in all markings as the value of tokens in places \( s_i \), in the form of a pair \(<6,7>\).

Figure 8 shows the first of eight almost identical internal blocks of the reachability graph, differing in the values of the loop counter variables \( i \), as well as the values of sum and Value. During the formation of the graph, the firing of transitions in the Master function was displayed in arcs leading down and to the left along the graph, and in the Slave function - down and to the right. Changes in markings when transitions are fired in the Master function are tinted in green, and when transitions are fired in the Slave function are colored red. This block clearly shows how, for two independent processes, the program state space is formed in the form of a Cartesian product of the states of each process.
Movement along arcs only down to the right corresponds to the execution of the Slave function while the Master function is stopped, and vice versa, only down to the left - to the execution of the Master function when the Slave function is stopped. Thus, starting the movement from the uppermost node down to the right, the program “stops” at the place s2, corresponding to the spinlock waiting until the value of the variable DataPrepared becomes non-zero. Moving from this point down to the left, the program reaches the state m5, corresponding to the spinlock waiting for the value of the variable DataPrepared to be equal to zero, while the variable DataPrepared itself at this moment takes the value 1, which is indicated in the place marking i2 token <3,1>. From this state, the program can only continue down and to the right, reaching the place s5, the next step after which the value of DataPrepared will again be set to 0, and this block can be repeated until the loops complete.

Figure 9 shows the last block of the reachability graph of the example program model, in which the functions Master and Slave complete their execution. The final state of the example program model is the marking \( g_1^4 + g_2^4 + p_{i1}^{4,16} + i_2^{3,0} \), which fixes the values of global variables at the moment when the program ends and the fact that global variables were not released due to the absence of a loader in the considered model.

Fig. 8. Internal part of the reachability graph for the model of example program
4 Reachability Graph Analysis

The correct behavior of the program is modeled as the achievement of the final state for any variant of program execution. From the point of view of analyzing program behavior in terms of Petri nets, there are three main possibilities for detecting errors. Firstly, these are deadlocks, which describe the state of the model in which not a single transition can be fired. Deadlocks can occur, for example, in the event of synchronization errors in parallel running processes in a program due to exhaustion of resources, as well as due to incorrect interaction of objects in the program. Secondly, errors in the program are identified by specially marked transitions indicating incorrect actions. Such errors are possible, for example, when using file handling library models or network communication library models, which pre-provide typical error cases, such as accessing an unopened or already closed file. Thirdly, errors in a program can be represented as freezes in “endless loops”, that is, a cyclic sequence of program actions from which there are no exit options.

In the built reachability graph there are no deadlocks or errors determined by explicitly labeled transitions. However, in all internal blocks of the program there are endless loop states, colored gray in the figures, containing in the marking $m_5 + s_6^{<6,7>}$, where $m_5$ is the spinlock synchronization state of the Master function, and place $s_6$ is the moment the function Slave accesses the variable Value by the number of the previously freed cell. But looking higher in the reachability graph, it is obvious that the three program states preceding the moment of looping are already erroneous, since the Slave function is already in its own deadlock state, unable to access a non-existent pointer. Therefore, even without referring to the source code of the program, the graph shows that the moment when the error situation rises is the transition from the state $... + m_5 + s_6^{<6,7>} + ...$ to the state $... + m_5 + s_6^{<6,7>} + ...$, which also leads to the same error in the program termination block.

5 Conclusion

There are two very different almost opposite approaches to program analysis, which are also applicable to finding errors in memory operations. The first approach is static, based on analysis of the program source code, using program models or symbolic execution. The second approach is dynamic, designed to analyze the state of the program during its execution. In the extreme case, the dynamic approach to program analysis turns into a process...
of testing and debugging, which, however, supplemented by modern means of annotating binary code, makes it possible to detect quite complex cases of erroneous program behavior. For example, Valgrind, one of the most popular dynamic program analysis tools, for this purpose uses the technique of executing a program in a virtual machine to gain maximum control over the operations performed by the processor during the execution of executable code [10]. An alternative approach is demonstrated by clang and gcc [1], which introduce modified operations into the code, instead of real operations of allocation, release and access to memory, that perform a number of checks, thus providing different levels of control over the correctness of operations. When erroneous behavior is detected, dynamic analysis of a program provides maximum information about the necessary initial data, the conditions for obtaining the error, and the execution paths of the computational processes that led to the error. At the same time, using dynamic analysis, you can only check a limited set of input data sets under limited program execution conditions, so there is no guarantee that there is no errors in the program code.

In the case of static analysis of a program, a graph of program control paths is usually built, on which error situations are searched. When it is possible to construct a graph with real values of variables, it describes the complete state space of the program, the analysis of which clearly guarantees the presence or absence of errors in the source text of the program. However, the so-called state explosion problem, which arises from the need to multiply the number of valid values of the set of variables used, makes it possible to construct a complete state space only for a very small number of rather primitive programs. To overcome the state explosion problem, symbolic calculations [11, 12] or algebraic-set representation [13], as well as contract mechanisms [14] can be used, and also a composition from the previously listed.

When analyzing a program, using the algebraic-set theory, instead of clearly defined values of variables, ranges of values can be used within which the behavior of the program can be considered unchanged. This approach can significantly reduce the number of considered combinations of variable values, but is not always feasible in practice. For example, for a simple expression like $x \% 7 > y \% 5$, where $x$ and $y$ are integer variables of type int in C++, it is difficult to enlist all the ranges of variable values for which the expression will be true, because the number of continuous ranges will exceed the memory of a modern computer. Symbolic calculations allow you to store symbolic expressions as definitions of ranges of variable values; accordingly, for the above example, two ranges are possible, where the value $x \% 7 > y \% 5$ is true and false. However, if you move along the control flow path graph, then the expression of variable value ranges will become more and more complicated with each transition. Moreover, for symbolic expressions there is a question of decidability, formulated by Hilbert in 1928, for which in general there is no positive answer. In this regard, it is necessary to shorten the length of the control flow path analysed at a time, which can be done, for example, using the contract mechanism. It involves calculating the minimum strict restrictions on input functions under which the function runs error-free. If such restrictions are found, then in the place where the function is called, it is enough to check the passed parameters for compliance with the restrictions. Since this place can also be restricted by a contract, then by extending contracts to all functions in the program, it is formally possible to ensure the correctness of the source texts. Calculation of a function contract also falls into the conditions of the solvability issue, and for polymorphic methods the situation is further complicated by the need to calculate conditions, the fulfillment of which for all implementations of the method will not lead to an error situation. It should be borne in mind that side effects, race conditions, and interactions of parallel control flows that affect the execution of functions are sometimes impossible to even express symbolically.

The alternative considered in this article involves the use of Petri nets to model the behavior of sequential interacting processes. The reachability graph of a program model can
be used as a program control flow graph to further analyze the values of variables at different points in execution time. At the same time, there is an attractive opportunity to perform as much of the analysis, as possible, in terms of Petri nets. To do this, it is necessary not only to build the program model in terms of Petri nets, but also to formulate the conditions for the occurrence of errors also in terms of Petri nets.

This article demonstrates an approach that allows to build automatically a program model as a composition of many Petri nets, some of which describe data types, another part - variables and pointers to variables, and finally, the main part - program control flows. In this case, various data type models can be used that describe the behavior of objects in a program with different levels of detail, due to which a reduction in the state space of the program can be achieved. Analysis of the built reachability graph for the model of the example program presented in the article shows the possibility of detecting errors and the conditions for their occurrence in the graph. Moreover, the error found in the sample program is not just a memory access error, but occurs under racing conditions, that is, it is a technically difficult to detect error. In general, this allows us to conclude that the proposed approach is applicable for modeling imperative programs in terms of Petri nets for detecting incorrect memory handling operations.

References


