SUMMARY

Development of methods and software tools for finding deadlocks in parallel Java programs.

Roman N. Guliak.

Supervisor Prof. Dmitry Zaitsev (http://daze.ho.ua).

The goal of this thesis is to verify the correctness of parallel Java programs.

The object of research are parallel Java programs that use an implementation of MPI (Message-Passing Interface) standard. The subject of reasearch is the presence of deadlocks in parallel Java programs. Research method is creation of program model in the form of Petri net.

Section 1 contains a review of MPI (Message-Passing Interface) specification and its implementation. In Section 2, a development and an implementation of analyzer are described. In Section 3, several examples of Petri nets for code fragments are presented and analyzed. Also, deadlock prevention recommendations are provided for MPI programs.

As a result of this work, a Petri net and a method of its construction for a given MPI program have been developed. The developed Petri net is aimed at checking that there are no deadlocks in the original MPI program. A static code analyzer has been developed. The analyzer automatically performs Petri net construction. The implementation of the analyzer uses both traditional approaches from the field of compiler development and new approaches specialized for the current task. The developed models showed their applicability both on a simple example, in which deadlock occurs every time the program is run, and on a more complex example, in which deadlock occurs nondeterministically.

Master's thesis: 30 pictures, 8 tables, 24 sources.

CONTROL FLOW GRAPH, DEADLOCK, JAVA DOMINATOR, MPI, MPJ EXPRESS, PARALLEL SYSTEM, PETRI NET, SCALA, SSA, STATIC CODE ANALYZER, TINA.

TABLE OF CONTENTS

ABBREVIATIONS

OS - operating system.

CPU - central processing unit.

AST - abstract syntax tree.

HIL - higher level intermediate language.

LIL - lower level intermediate language.

MIMD - multiple instruction, multiple data.

MPI - Message Passing Interface.

SSA - static single assignment.

INTRODUCTION

Today, distributed and parallel software systems play a huge role in a wide variety of applications in both science and industry. Parallel programs have been developed for supercomputers for decades. However, the development of parallel programs is becoming increasingly important for personal computers as well. This is due to the fact that the growth rate of processor cores has decreased. Nowadays, to get an increase in program performance, one needs to use many processor cores.

One of the tools for developing parallel software systems is MPI. MPI allows to split a program into processes that exchange messages with each other. During this message exchange, undesirable situations may occur, which are known as deadlocks or simply deadlocks. These situations occur when a program cannot continue its work because the processes are mutually dependent. Deadlocks are not easy to detect at runtime because it is not always clear which processes are blocked.

Hence the need for code analyzers that would automatically search for deadlocks in MPI programs. Such tools can be dynamic, which run the program, and static, which only analyze its source code. Dynamic analyzers are more accurate, and the advantage of static analyzers is that they do not run the program.

Therefore, the aim of this work is to find an alternative method of static analysis of MPI programs that would have better accuracy. A possible area of application is serious MPI programs in the field of high-performance computing.

A methodology for modeling MPI programs with Petri nets that can adequately represent the behavior of parallel systems has been developed. Also, a static code analyzer has been developed that creates a Petri net for a given Java program that uses an MPI implementation. The analyzer's implementation uses both traditional approaches from the field of compiler development and new approaches specialized for the current task.

1 OVERVIEW OF THE SUBJECT AREA

1.1. Description of MPI

1.1.1 Description of the standard and implementations

MPI (Message Passing Interface) is an interface specification for message passing interface libraries. MPI programs consist of processes that interact with each other by exchanging messages. Each process has its own address space, and processes operate on the MIMD principle (multiple instruction, multiple data). The main content of the MPI standard is procedures or functions that can be called from the FORTRAN and C programming languages.

When creating the MPI standard, its authors pursued the following goals:

- achieving effective communication between processes;
- user-friendly interface for C and FORTRAN languages;
- interface semantics should be language-independent;
- reliable communication interface: problems with communication errors should be solved at the level of the communication subsystem, not by the user.

The creators of MPI tried to use the most attractive features of the message passing systems that existed at that time. MPI was influenced by such systems as NX/2, p4, PARMACS, Zipcode, Chimp, PVM. Representatives of large parallel computer vendors, scientific researchers, government laboratories, and industry representatives participated in the standardization of MPI. The first version of the MPI standard was released in 1994 under the name "A Message-Passing Interface Standard Version 1.0". In 1995, the MPI 1.1 standard was published. MPI 1.1 described such functionality as sending and receiving messages between individual processes, collective process interaction, working with groups of processes, and process topologies.

The MPI 2.0 standard appeared in 1997. It also included the MPI 1.2 standard, which had slight differences from version 1.1. MPI 2.0 had the following innovations: an extended computing model, dynamic process creation, one-way communication, and parallel I/O.

After minor versions 2.1 (2008) and 2.2 (2009), the MPI 3.0 standard was released. It contained non-blocking versions of collective operations and an extension of one-way communications. The MPI 3.1 standard (2015) contains edits and clarifications.

And finally, the latest to date is the MPI 4.0 standard, which was released in 2021. The biggest changes are the use of a large integer type in many procedures to overcome the limitations of a simple integer type, split communications, and improvements for error handling.

Several MPI implementations are widely used today. Argonne National Laboratories' MPICH and OpenMPI are the two most common. Hardware vendors often have specialized versions of one of these two implementations for their platforms.

In this paper, we will use MPJ Express [1]. MPJ Express is an open-source Java messaging library that enables application developers to write and execute parallel applications for multi-core processors and computing clusters/clouds. In the following, we will use the terms MPI and MPJ Express interchangeably.

1.1.2 The relevance of using MPI

For more than two decades, microprocessors based on single central processing units have shown particular improvements in performance and cost reduction in computer applications [2]. These improvements made it possible to add more functionality to software, create better user interfaces, and produce more useful results. However, in 2003, this trend began to slow down due to power consumption and heat dissipation issues, which limited the increase in processor frequency and the level of productive activity that could be performed in each clock cycle within a single CPU. Since then, virtually all microprocessor vendors have shifted to models that use multiple processor units, called processor cores, in each token to increase processing power. This transition has had a huge impact on the software development community. Now, to get performance gains in applications, it is not enough to create traditional sequential programs that run on only one processor. Instead, to take advantage of improvements in computer hardware, programs must consist of multiple threads of execution that interact with each other to get the job done faster.

The practice of parallel programming is not new. The HPC community has been developing parallel programs for decades. These programs usually run on large and expensive computers. Most performance-demanding applications have traditionally been in the scientific field. But artificial intelligence and machine learning applications are predicted to become the main users of large-scale computing [3]. Some examples of such applications include:

– fire modeling to help fire brigades;
– modeling tsunamis and storm surges from hurricanes;
– voice recognition for computer interfaces;
– modeling the spread of the virus and developing a vaccine;
– modeling climate conditions over decades and centuries;
– image recognition for self-driving cars.

MPI is successfully used in high-performance computing.

Applications written in MPI successfully run on cluster computing systems with more than 100 thousand nodes [2].

1.2 Overview of the MPJ Express software interface

Next, we will consider the basic concepts of MPI. As mentioned, an MPI program consists of interacting processes. Processes can belong to groups. Groups are an ordered set of processes. Each process in the group is associated with an integer rank. Rank numbering starts from zero. Communicators are created on the basis of groups. Each communicator contains a group of participants; this group always includes a local process. The sender and recipient of a message are identified by the process rank within this group.

There is a special communicator - MPI_COMM_WORLD. This is the initial communicator, which consists of all local processes with which the process can interact after initialization, including itself. In the MPJ Express package, a similar communicator is the static COMM_WORLD field in the mpi.MPI class.

1.2.1 General purpose operations

The MPI standard has general-purpose operations that are not related to message transfer.

One of these operations is Init. In MPJ Express, this method is declared in the mpi.MPI class and has the following definition:

```
public static java.lang.String[] Init(java.lang.String[] argv) throws
MPIException
```

The MPI analog is the MPI_Init operation. This method initializes the MPI environment. Programs that use MPJ Express must start by calling Init. Calling most other operations makes sense only after calling Init. We also see that Init throws an MPIException. This is a common exception thrown by most of the methods in the MPJ Express library.

The next important operation is Finilize, which is also defined in mpi.MPI:

```
public static void Finalize() throws MPIException
```

The MPI analog is the MPI_Finilize operation. This operation clears the state of MPI. If the program terminates normally, and not due to an Abort call or other unforeseen situation, then each process must call Finilize before exiting.

The Abort operation belongs to the mpi.Comm class and has the following declaration:

```
public void Abort(int errorcode) throws MPIException
```

The MPI analog is the MPI_Abort function. This operation tries to abort all tasks in the communicator group. The error code errorrcode will be passed to the runtime.

The Size and Rank operations belong to the mpi.Comm class and are declared as follows:

```
public int Size() throws MPIException
public int Rank() throws MPIException
```

The MPI analogs are the MPI_Comm_size and MPI_Comm_rank functions, respectively. Size returns the number of processes in the communicator group. Rank returns the rank of the process in the communicator group.

1.2.2 Point-to-point operations

Sending and receiving messages by processes is the basic communication mechanism in MPI. The basic operations are Send and Recv. Here is a simple example of how to use these operations.

```
import mpi.MPI;

public class MpiSendRecv {
  public static void main(String[] args) {
    int rank = MPI.COMM_WORLD.Rank();
    int[] buffer = {0, 2, 3, 4};
    if (rank == 0) {
      MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
    } else if (rank == 1) {
      MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, 0);
    }
  }
}
```

In this example, we see how the process with rank 0 (rank = 0) sends a message to the process with rank 1; the interaction takes place inside the COMM_WORLD communicator. The first four arguments for Send are a buffer with data, an offset from the beginning of the buffer, the number of elements, and the data type. In addition to the data itself, a so-called *message envelope* is also transmitted. This envelope indicates the recipient of the message and also contains additional information that will help the Recv operation select a particular message. The last two arguments for Send are the recipient's rank and the message *tag*. The tag is an identifier that helps distinguish messages from each other and is part of the message envelope.

In turn, the process with rank 1 calls the receive operation - Recv. The message is selected according to its envelope, then the message data is saved to the receive buffer. Once again, the first four arguments for Recv specify information about the buffer: the buffer itself, offset, number of items, and type. The last two arguments specify the sender's rank and tag.

Point-to-point operations can be:

– blocking operations - the call does not end until it is safe to use the buffer;
– non-blocking operations - the call ends immediately.

The analyzer will process only blocking operations.

Next, the Send and Recv methods are described in detail. The Send method belongs to the mpi.Comm class and has the following definition:

```
public void Send(java.lang.Object buf, int offset, int count, Datatype
datatype, int dest, int tag) throws MPIException
```

The MPI analog is the MPI_Send function. Table 1 shows the parameters of the Send method.

Table 1 - Parameters of the Send method

Parameter name	Parameter description
buf	buffer array to be sent
offset	offset from the beginning of the array
count	number of items to be sent
datatype	data type of each element in the array
dest	recipient's rank
tag	message tag

The Send operation is a blocking operation. The actual argument associated with buf must be a one-dimensional array. The value offset is the lower index in this array, which determines the position of the first element of the message. The length of the message is determined by the count parameter and indicates the number of elements of a given type, not the number of bytes.

The Recv method also belongs to the mpi.Comm class and has the following definition:

```
public Status Recv(java.lang.Object buf, int offset, int count, Datatype
datatype, int source, int tag) throws MPIException
```

The MPI analog is the MPI_Recv function. This is also a blocking operation Table 2 shows the parameters of the Recv method. The actual argument associated with buf must also be a one-dimensional array. The Recv method returns an object of the mpi.Status class, which is described below.

Table 2 - Parameters of the Recv method

Parameter name	Parameter description
buf	a buffer array to which the data will be written
offset	offset from the beginning of the array
count	the number of items to receive
datatype	data type of each element in the array
source	sender rank
tag	message tag

The mpi.Status class contains information about the status of message receipt. Let's consider the most important fields of this class:

```
public int source;
public int tag;
```

The source field contains the sender's rank, and the tag field contains the tag of the received message. The reason why this information can be useful is as follows. When calling Recv, the user can specify the mpi.MPI.ANY_SOURCE constant as the sender rank. In this case, the message will be selected from any sender. To find out from which process the message was received, the user can use the source field. Similarly, the user can specify mpi.MPI.ANY_TAG as a message tag and later use the tag field in the mpi.Status class object.

The Send and Recv operations have the following differences:

- to perform the Send operation, a recipient must be specified;
- to perform the Recv operation, a sender may be specified, but it is not required.

1.2.3 Collective operations

Collective operations are operations that involve a group or groups of processes [4]. One of the key arguments for collective operations is the communicator that defines the group of processes. The syntax and semantics of collective operations are consistent with point-to-point operations. Therefore, the same data types are applicable in collective operations as in point-to-point operations. Several collective operations have a single source or receiving process called the *root process*. Some operation arguments are meaningful only to the root process. Collective operations do not have tags as an argument.

Bcast and Barrier operations are among the most frequently used collective operations, according to [5].

The Bcast method is defined in the mpi.Intracomm class:

```
public void Bcast(java.lang.Object buf, int offset, int count, Datatype
type, int root) throws MPIException
```

The MPI analog is the MPI_Bcast function. This method distributes data from the root process, which is indicated by the root parameter, to other processes in the communicator group. A description of the parameters of the Bcast method is given in Table 3.

Table 3 - Parameters of the Bcast method

Parameter name	Parameter description
buf	buffer array
offset	initial offset in the buffer
count	number of array elements
datatype	data type of each element in the array
root	rank of the root process

Fig. 1 shows how a root process with rank 0 sends an array {11, 22, 33, 44, 55} to processes with ranks 1, 2, and 3.

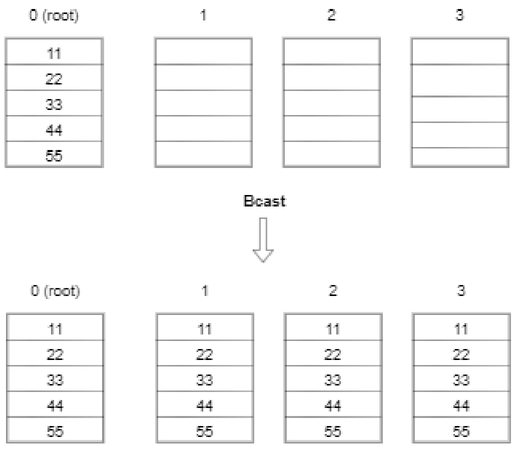

Figure 1 - The result of the Bcast operation

The Barrier method belongs to the mpi.Intracomm class and is defined as follows:

```
public void Barrier() throws MPIException
```

The MPI analog is the MPI_Barrier function. The Barrier method blocks the calling process until all processes in the communicator's group also call it. This method is useful when there is a need to synchronize all processes in a group at one point.

1.2.4 Compile and run MPJ Express programs

Below is a program that uses MPJ Express.

```
import mpi.MPI;
public class HelloMPJ {
  public static void main(String[] args) {
    MPI.Init(args);
    int size = MPI.COMM_WORLD.Size();
    int rank = MPI.COMM_WORLD.Rank();
    System.out.printf("Process %d of %d\n", rank, size);
    MPI.Finalize();
  }
}
```

This simple program initializes MPI, gets the number of processes in the group and the rank of the current process, and displays this information in the console.

To work with MPJ Express, one needs to install this package, which can be downloaded from the official website [6].

If the package is installed at the path specified in the MPJ_HOME environment variable, a program using MPJ Express can be compiled in Windows as follows:

```
javac -cp %MPJ_HOME%/lib/mpj.jar HelloMPJ.java
```

This creates the HelloMPJ.class file, which can then be run as follows:

```
%MPJ_HOME%/bin/mpjrun.bat -np 4 HelloMPJ
```

The mpjrun.bat script is needed to distribute the program to processes. The -np 4 command line argument indicates that the number of processes should be four. The result of the run may be as follows:

```
MPJ Express (0.44) is started in the multicore configuration
Process 2 of 4
Process 0 of 4
Process 3 of 4
Process 1 of 4
```

1.3 The problem of deadlocks in MPI programs

This subsection describes the problem of deadlocks when using blocking calls for point-to-point operations such as Send and Recv. A *deadlock* or *deadlock* is a situation in which processes cannot continue working due to mutual dependence.

During a blocking send, an MPI implementation often writes the original message to a buffer [5]. In this case, the send operation may complete before the corresponding receive operation is called. On the other hand, the buffer space may be insufficient. In this case, the sending operation will not be completed until the corresponding receive operation is called and the data is transferred to the recipient.

During a blocking receive, the operation call is blocked in any case - the operation is completed only after the transferred buffer contains the received message. A receive can also complete before the corresponding send is completed.

The MPI standard describes various guarantees for Send and Recv operations. Below are the most important ones for this work.

Suppose that the sender calls Send twice for the recipient. The recipient calls the Recv operation that responds to both messages. In this case, the recipient is guaranteed to receive the first message.

If a process sends two messages sequentially and they both respond to a Recv call in the recipient's process, then the second Send operation cannot start until the first one has completed.

If two processes send messages to a third process, and the receive operation in the third process responds to messages from both processes, then MPI does not guarantee which message will be received by the third process.

Below are three examples of how to use send and receive operations.

Example 1: In this example, process side 0 sends and receives in sequence. On the side of process 1, there is receiving and sending.

```
if (rank == 0) {
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 1, tag);
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 1, tag);
} else if (rank == 1) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 0, tag);
}
```

This fragment will execute successfully in any case, even if the buffer space is insufficient.

Example 2: This example differs from the previous example in that process 0 first calls Recv, then Send.

```
if (rank == 0) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 1, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 1, tag);
} else if (rank == 1) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 0, tag);
}
```

A deadlock will always occur in this fragment. The Recv operation in process 0 can complete only if Send is called from process 1. Recv in process 1 must complete before it calls Send, but this can only happen if Send is called from process 1. Thus, both processes are mutually waiting and blocking.

Example 3: In this example, both processes call Send and then Recv first.

```
if (rank == 0) {
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 1, tag);
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 1, tag);
} else if (rank == 1) {
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 0, tag);
}
```

This fragment can be executed successfully only if message buffering is happening during sending. In this case, the Send call will be terminated and the Recv operation will be called, for which a suitable message will be found. However, if the buffer is not saved, a deadlock will occur, as in the previous example.

1.4 Overview of available MPI analyzers

Analyzers that check the correctness of MPI programs can be divided into two groups:

– static analysis;
– runtime analysis.

Static analyzers work only with the source code of a program, extracting the necessary properties from it. Then they check these properties for correctness. In this case, the program does not run.

When using runtime analyzers, on the contrary, the program itself is launched. To find erroneous states, calls to some operations are intercepted. The information about the calls is processed and thus the analyzer can draw conclusions about the correctness of the target program.

One of the static analyzers for MPI programs is MPI-Checker [7]. It is based on the Static Analyzer infrastructure of the Clang compiler. MPI-Checker extracts information about a program using only the abstract syntax tree of the program. One part of the checks is aimed at blocking calls. The second part of the checks makes sure that the types for MPI calls are correct. For example, MPI-Checker reports an error if the passed buffer for the MPI_Send call is of type double* and the data type is MPI_INT.

MPI-SV [8] can also be classified as a static analyzer. MPI-SV verifies non-blocking MPI operations. It uses symbolic execution and model checking. MPI-SV is based on KLEE, a symbolic execution engine. Symbolic execution [9] is an analysis technique in which the actual input data of a program is replaced by arbitrary symbolic values. The program is then interpreted according to the semantics of the programming language. In this case, the data in the program can be represented by formulas from the input symbolic data. Note that this approach does not allow to execute the program, but only to interpret it. Model checking is a technique for automatically proving that a given model of any system meets the specified properties. Model checking works by systematically examining all states of the model.

In [10], the authors show a runtime analyzer. This tool runs an MPI program and tracks the sequence of MPI function calls. From this sequence of calls, a formula is built using a special logical coding scheme. Then it checks whether the formula is executable. This analyzer is aimed at finding deadlocks.

1.5 Overview of Petri nets

1.5.1 Overview of the Petri net structure

Petri nets consist of 4 elements: places, transitions, arcs, and tokens (see Figure 2). Places are represented by circles, transitions by rectangles. Arcs can connect places to transitions or transitions to places, but they cannot connect places to places or transitions to transitions. Tokens are placed inside places and are represented by bold dots.

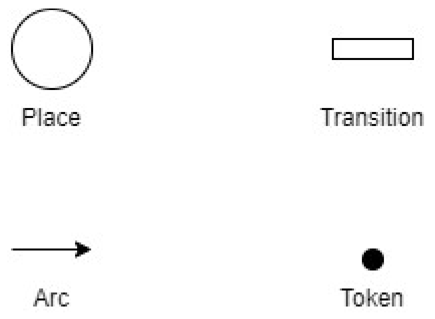

Figure 2 - Elements of the Petri net

If the arc is directed from a place to a transition, the place is called the input place for this transition. If the arc is directed from a transition to a place, the place is called the output place.

When the transition is triggered, the tokens are removed from the input places and added to the output places. To trigger a transition, it is necessary that all the input places of the transition contain tokens. If this condition is met, the transition is called allowed.

Fig. 3 shows an example of a transition being triggered. Before the triggering, the transition is allowed, but after the triggering, it is not, because the input places do not contain tokens.

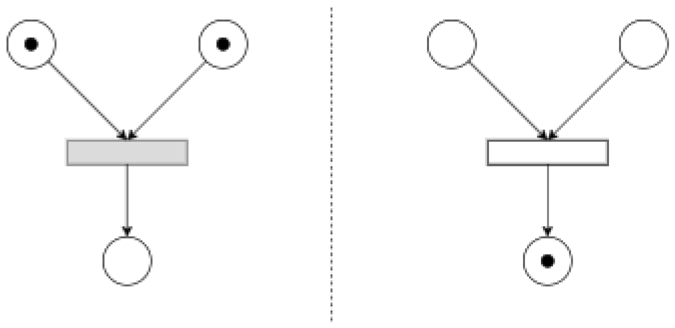

Figure 3 - Triggering the transition

It is possible that a Petri net has more than one excited transition. In this case, the transition selected at random is triggered. For this reason, nondeterminism occurs in the Petri net. The presence of nondeterminism in Petri nets allows to naturally represent the parallel essence of the modeled systems.

In addition to single arcs, multiple arcs are allowed in Petri nets [11]. In this case, the triggering conditions must be met for each arc instance. As a rule, multiple arcs are depicted by writing the number of arcs above a single arc. Fig. 4 shows an example of a transition triggering on a Petri net with multiple arcs.

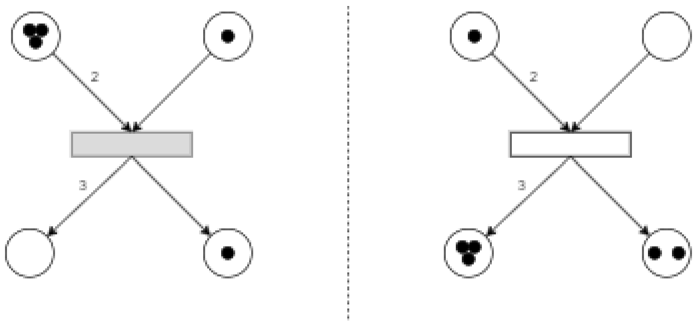

Figure 4 - Triggering a transition with multiple arcs

1.5.2 Overview of the properties of Petri nets

To present the properties of Petri nets that are important for this paper, we formally define a Petri net according to [12]. We assume that NN is a set of positive integers including 0. A Petri net is a set $N=(P, T, F, M_0)$where:

P is a finite set of all places;

T is a finite set of all transitions;

$F: P\times T\cup T\times P\rightarrow\mathbb{N}$ - incident rate function that sets the arcs and their multiplicity;

$M_0 : P \rightarrow \mathbb{N}$ - initial net marking.

Let's introduce special designations for the input and output arcs of places:

$$\{t \vee F(t,p) > 0\},\ p^{\bullet} = \{t \vee F(p,t) > 0\}.$$

Similarly, we will introduce special designations for the input and output arcs of transitions:

$$\{p \vee F(p,t) > 0\},\ t^{\bullet} = \{p \vee F(t,p) > 0\}.$$

The functioning of a Petri net is described formally using the set of trigger sequences and the set of marking achievable in the net.

The state of the Petri net is determined by its marking. The marking of the net *N* is a function of $M : P \rightarrow \mathbb{N}$. The marking defines the distribution of tokens by places. The *state space* of a Petri net is the set of all labels.

Assuming that all places of the net *N* are strictly ordered in some way, i.e. $P = (p_1, \ldots, p_n)$, then the marking *M* of the net (including the initial marking) can be specified as a vector of numbers $M = (m_1, \ldots, m_n)$ such that

$$\forall\ i,\ 1 \leq i \leq n : m_i = M(p_i).$$

If the places are ordered, then we can map two integer vectors to each transition $t\ (t)$ i $F^{\bullet}(t)$ of length *n*, where $n = |P|$:

$(t)=(b_1,\ldots,b_n)$, where $b_i=F(p_i,t)$,

$F^{\bullet}(t)=(b_1,\ldots,b_n)$, where $b_i=F(t,p_i)$.

A transition *t* can be triggered (transition *t is* allowed) for some markup *M* of the net *N* if

$$\forall\ p\in\ :M(p)\geq F(p,t),$$

i.e., each input place *p* of a transition *t* has a number of tokens at least equal to the multiplicity of the arc connecting *p* and *t*. This condition can be rewritten in vector form as follows:

$$M\geq(t).$$

The triggering of a transition *t* in the marking of *M* generates a new marking *M '* according to the following rule:

$$\forall\ p\in P:M'=M(p)-F(p,t)+F(t,p),\text{ or}$$
$$M'=M-(t)+F^{\bullet}(t).$$

Thus, the triggering of a transition *t* changes the marking so that the marking of each of its input places *p* decreases by $F(p,t)$that is, by the multiplicity of the arc connecting *p* and *t,* and the marking of each of its output places *p* increases by $F(t,p)$that is, by the multiplicity of the arc connecting *t* and *p*.

Let's introduce a direct marking relation on the set of markings:

$$M \to M' \Leftrightarrow \exists\, t \in T : (M \geq (t)) \wedge (M' = M - (t) + F^{\bullet}(t)).$$

We will use the qualifier $M \to_t M'$if M' immediately follows *M* as a result of the triggering of the transition *t.*

A marking is said to be *M'* is reachable from a marking *M* if there exists a sequence of markings $M, M_1, M_2, \ldots, M'$ and a sequence of transitions $t_1 t_2 \ldots t_k$ from the set *T* such that

$$M \to_{t_1} M_1 \to_{t_2} M_2 \ldots \to_{t_k} M'.$$

The set of markings reachable in the net *N* from the marking *M* is denoted by $R(N, M)$. The set of reachable markings of the net *N* is the set of all markings reachable in *N* from the initial marking $M_0$and denoted by $R(N) = R(N, M_0)$.

Next, we consider the liveness property and other related properties.

A transition *t* in the Petri net $N = (P, T, F, W, M_0)$ is called potentially alive when labeled $M \in R(N)$if

$$\exists\, M' \in R(N, M) : M' \geq (t),$$

i.e., there exists a marking reachable from *M* M'at which the transition *t* is allowed. If $M = M_0$then *t is* called potentially live in the net *N*. A transition *t is* dead at *M* if it is not potentially live at *M*. A transition *t* is dead if it is dead at any reachable marking in the net.

A transition *t* is called live in a Petri net *N* if

$$\forall\ M \in R(N), \exists\ M' \in R(N, M) : M' \geq (t),$$

i.e., a transition t is potentially live for any reachable marking in the net. A transition t is potentially dead if there exists a reachable marking M such that for any marking $M' \in R(N, M)$ transition t cannot be triggered. In this case, the marking M is called *t-deadlocked*. If it is *t-deadlocked* for all $t \in T$ then it is deadlocked. If M *is a deadlocked marking,* then $R(N, M) = \{M\}$. A net is called live if all its transitions are live.

1.5.3 Overview of the Tina package tools

Tina (Time Petri Net Analyzer) is a set of tools for editing and analyzing Petri nets. The Tina package was developed at OLC, then VerTICS, a research group of the LAAS/CNRS research laboratory.

Tina contains a graphical editor for Petri nets. On Windows, the editor can be found from the Tina installation directory at the following path: bin\nd.exe. In the editor it is possible to create a Petri net as shown in Fig. 5.

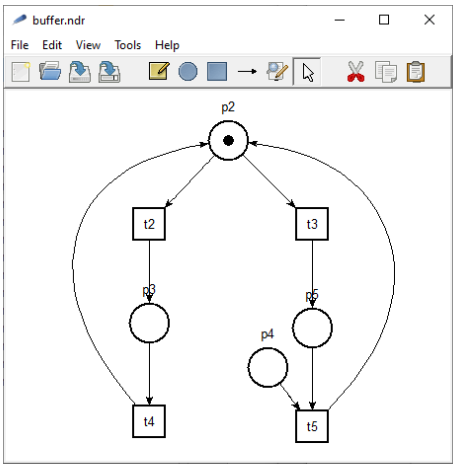

Figure 5 - Tina Petri net editor

The nd graphical editor has the ability to simulate the launch of a Petri net. To switch to the simulation mode, go to the Tools \ stepper simulator menu item shown in Fig. 6.

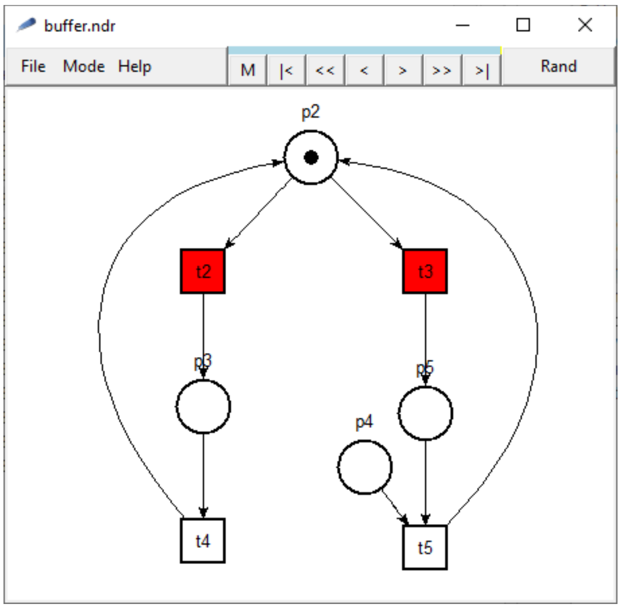

Figure 6 - Petri net simulation mode

Allowed net transitions are highlighted by the simulator. When user clicks on a transition, it is triggered. With the help of the simulation, one can make sure that the net comes to a dead end when selecting the transition t3. In this case, the token will be only in place p5, but not p4, and therefore neither the t5 transition nor any other transition will be allowed.

Another tool from the Tina package that we will use is sift. sift builds various state space abstractions for Petri nets [13]. It accepts as input descriptions in text form (.net, .pnml, .tpn formats) or graphical form (.ndr files created by nd, .pnml with graphics). sift allows to efficiently process large state spaces.

Let's consider the operation of sift on the example of a Petri net that was saved in the file simpleNet.ndr:

```
> C:\programs\tina-3.7.0\bin\sift.exe .\simpleNet.ndr -dead

some state violates condition -f:
  p5
  firable:
3 marking(s), 3 transitions(s)
```

The -dead option indicates deadlocks and stops sift if they are found. The result of the sift shows that when marking p5, i.e. when the token is located only in place p5, there are no allowed transitions (firable).

2 DEVELOPMENT OF THE ANALYZER

2.1 Petri net as a tool for modeling distributed systems

MPI programs consist of processes that interact with each other. The nature of these interactions is parallel. To model this behavior, Petri nets were chosen for this work. As mentioned in sub-section 1.5.1, Petri nets allow for a natural representation of the parallel nature of the modeled systems. In addition, Petri nets are easy to represent graphically, which facilitates their understanding and construction. Also, Petri net theory clearly formulates the properties of nets that relate to deadlocks.

As an example of the successful use of Petri nets for modeling parallel systems, we can point to [14]. In this work, a model of a multidimensional interaction lattice for a switch was built, which implements a non-deterministic packet forwarding solution.

Effective software packages have been developed for Petri nets, such as Tina. Tina is the winner of the Model Checking Contest [15].

One of the alternative approaches to checking the correctness of parallel systems is process algebra. In process algebra, the behavior of a system is described in the form of algebraic expressions [16].

As noted in [17], process algebra belongs to models that exhibit interleaving. In such models, parallel execution of tasks is simulated as a choice of their ordering. On the contrary, Petri nets are models that show true concurrency. The behavior of systems in such models is represented in the form of causal relationships between events in different locations of the system [17].

2.2 Building a Petri net for the MPI program

The main contribution of this paper is the development of a method for building a model in the form of a Petri net according to a given MPI program with the aim of further checking for deadlocks. Currently, this method has the following limitations:

– the program must be represented in one method; calls to other methods defined by the user are not processed;
– it is possible to process only loops with conditions that can be calculated at the compilation stage;
– the process rank can be determined at the compilation stage;
– the number of processes in the program should be limited.

Most MPI programs use this scheme. At the very beginning of the program, the rank of the current process is obtained. Then the process is checked for process rank and each process performs its task.

Below is an example of a program where a deadlock occurs.

```
int rank = MPI.COMM_WORLD.Rank();
if (rank == 0) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 1, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 1, tag);
} else if (rank == 1) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 0, tag);
}
```

For this example, the Petri net shown in Fig. 7. The resulting net is cyclic. Back edges are always present because the places corresponding to the beginning and end of the program are connected by an intermediate transition.

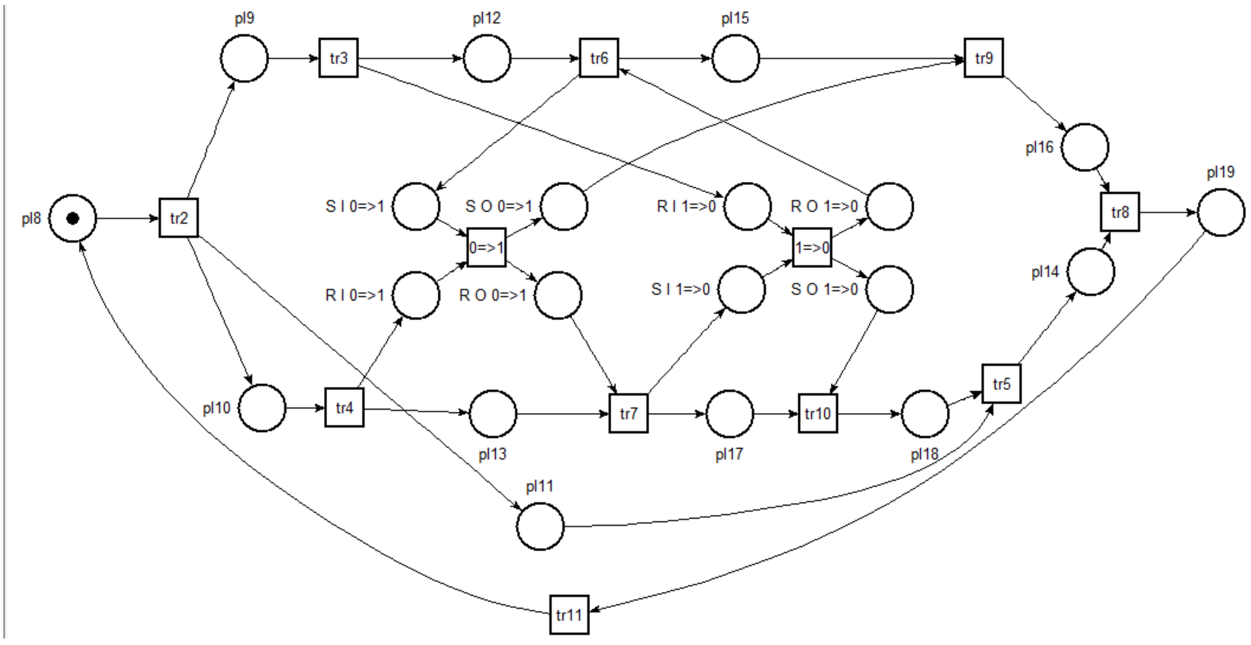

Figure 7 - The constructed Petri net

A program is correct from the analyzer's point of view if it does not have deadlocks caused by Send and Recv operations. Then the Petri net corresponding to a correct program should not have any labels that can be reached from the initial marking and are deadlocks. In other words, the net must be deadlock-free.

In the initial marking of the constructed Petri net, the token is located in the place that corresponds to the beginning of the program. In Fig. 7, this place is pl8. In Fig. 8, the tokens are in the following places: pl11, pl12, pl13, "R I 0=>1" and "R I 1=>0". The token in the place "R I 0=>1" indicates that process 0 called the operation of receiving a message from process 1. Similarly, the token in the place "R I 1=>0" indicates that process 1 called the operation of receiving a message from process 0.

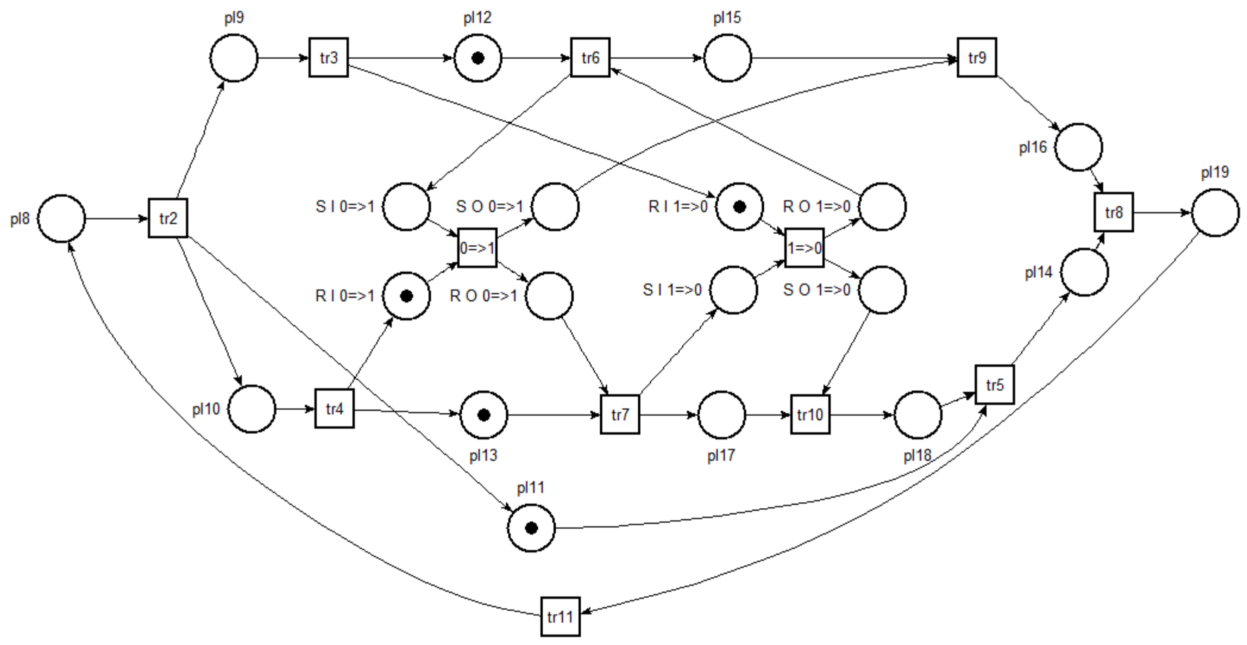

Figure 8 - Deadlock marking of the Petri net

The marking shown in Fig. 8 is a deadlocked marking because no transitions are allowed. For example, the transition tr6 requires that process 0 has finished receiving a message from process 1, which is indicated by the place "R O 1=>0". But this is impossible because process 1 has not even started the sending operation at this time. Similarly, transition tr7 is not allowed because it requires process 1 to finish receiving the message, i.e., it requires a token at the place "R O 0=>1".

When building a Petri net, the following important points should be considered:

- modeling of blocking during MPI operations;
- the sequence in which operations are called;
- points of separation of processes;
- points of synchronization of processes.

To model blocking in MPI operations, groups of places and transitions are used. We call such groups gateways. Gateways must block both processes that exchange messages until both the send and the receive are called. If the program contains repeated calls to MPI operations for the same sender and receiver pair, then the gateway is not duplicated for them.

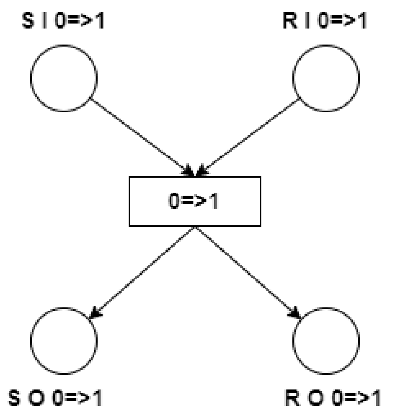

Figure 9 - Gateway for sending a message from process 0 to process 1

Fig. 9 shows an example of a gateway for sending a message from process 0 to process 1. In the middle is the transition, which is marked with the symbols "0=>1". Above are the places representing the inputs to the operations: "S I 0=>1" - input to the send operation, "R I 0=>1" - input to the receive operation. Below are similar places that correspond to the outputs of operations: "S O 0=>1" and "R O 0=>1". The "0=>1" transition is not allowed until Send and Recv are called for the corresponding processes. After this transition is triggered, further operation of the Petri net becomes possible.

A simple combination of places and transitions is used to represent a sequence of operations. Later, such sequences can be associated with specific points in the source program.

Below is a code snippet in which process 0 sends two messages to process 1 sequentially.

```
if (rank == 0) {
  MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
  MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
}
```

Figure 10 shows a fragment of the corresponding Petri net. In Fig. 10 on the right shows the gateway described above. Places pl1, pl2, pl3, pl4 and transitions tr1, tr2, tr3 on the left in Fig. 10 express the sequence of operations in the program. Place pl1 corresponds to the point in the program before the first call to Send. The pl2 place indicates that Send was called for the first time, but not completed. The pl3 place indicates that the first Send call was successfully completed and the second call was started. The pl4 place corresponds to the point in the program after the second Send call has been successfully completed. One gateway is used because both messages are sent from process 0 to process 1.

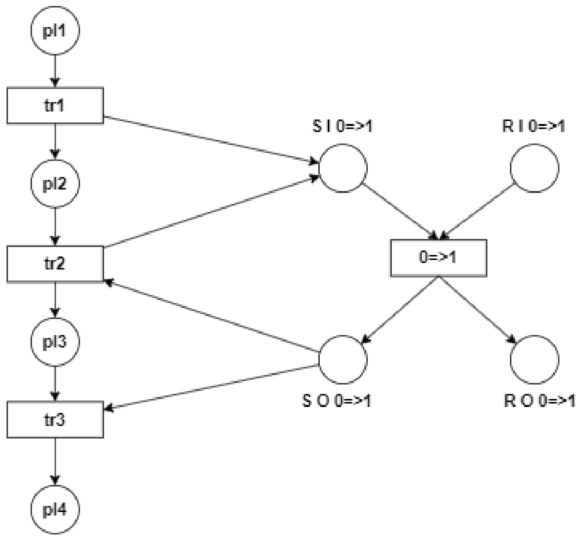

Figure 10 - A fragment of the Petri net showing the sequence of operations

In fact, the same MPI program is executed by multiple processes in parallel. In this program, there are parts that are executed by all processes, and there are parts that are executed only by certain processes. On the constructed Petri net, the parts of the program that are executed by all processes are not duplicated. This reduces the number of transitions and places in the net. Therefore, it becomes necessary to represent process separation points and synchronization points in the net.

The set of Petri net elements shown in Fig. It is also known as AND-split [18]. AND-split splits the execution thread into two parallel processes.

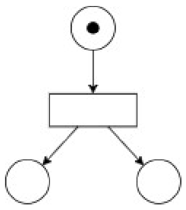

Figure 11 - AND-split

To represent the synchronization points, a set of AND-join elements [18] is used, shown in Fig. 12. AND-join models the connection of two parallel threads.

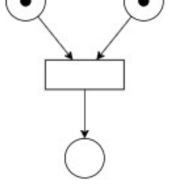

Figure 12 - AND-join

2.3 General description of the Petri net construction algorithm

An algorithm was developed to automatically build the Petri net. To analyze the program, we used both traditional approaches from the field of compiler development (dominators, SSA form, data flow analysis, and others) and new approaches specialized for the current task, such as determining the ranks of processes and the graph of operations.

Below is an algorithm for building a Petri net according to a given program. The corresponding block diagram of the algorithm is shown in Fig. 13.

– The text of the input program is parsed using the Eclipse JDT library. As a result, an abstract syntax tree (AST) is created.
– AST is transformed into a form of HIL - a high-level intermediate language.
– The HIL form is transformed into the LIL form, a low-level intermediate language.
– Based on the LIL form, a control flow graph is built.
– The LIL shape and the control flow graph are used to partially execute the program. This is mostly done for the purpose of deploying loops.
– The analysis of the variables' liveness is performed.
– Information about dominators is calculated.
– Based on the analysis of liveliness and information about dominators, the SSA form is built.
– Predicate propagation is performed. It determines which code blocks are executed by which process.
– The operation graph is being built.

- The operation graph is transformed into a Petri net.
- the Petri net is written to a file with the .net extension of the Tina format.

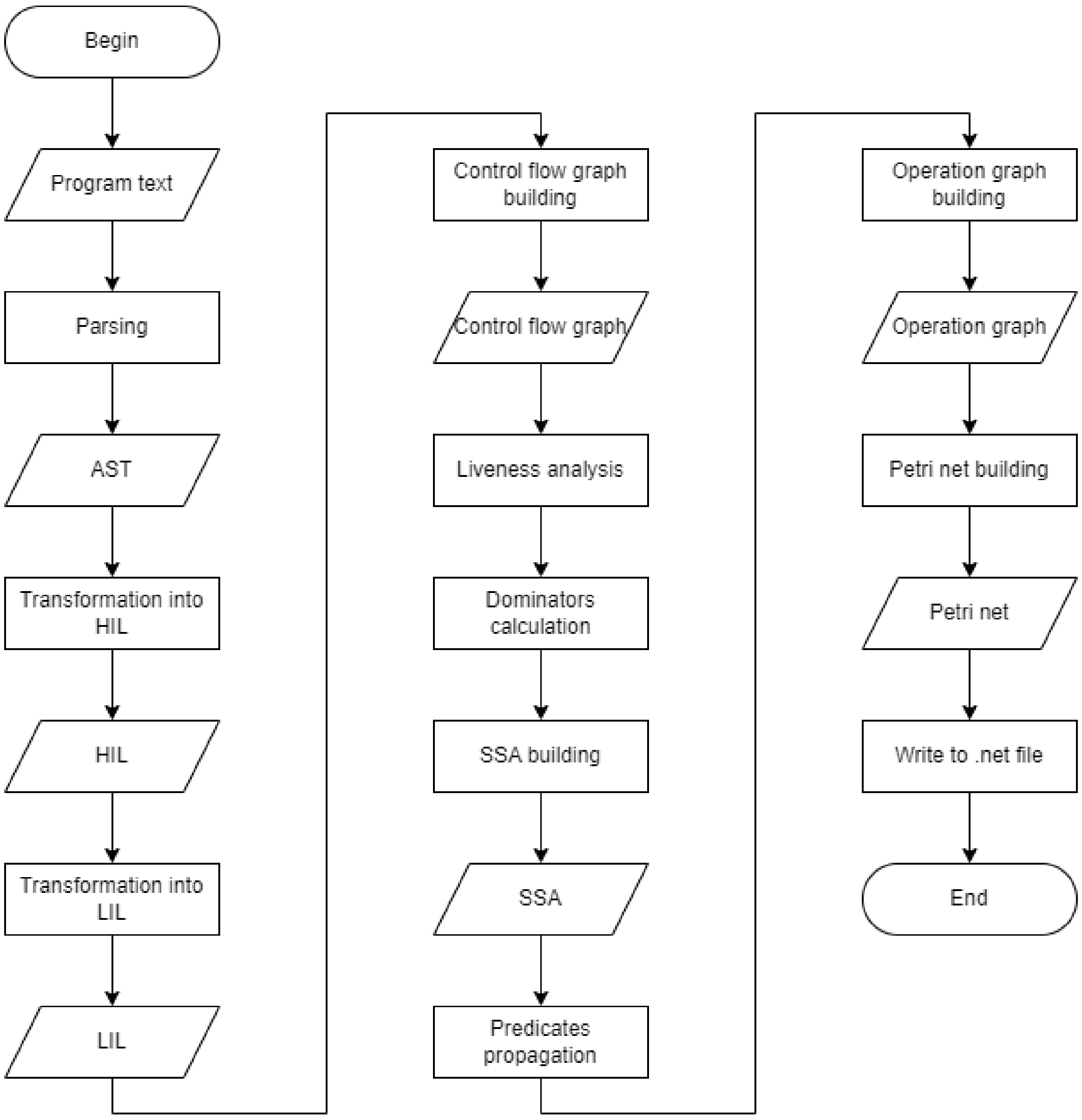

Figure 13 - Flowchart of the Petri net construction algorithm

Fig. 14 shows the relationship between intermediate views and auxiliary data structures. The program text, Eclipse AST, HIL, and LIL are converted to each other sequentially. The fundamental intermediate view is the control flow graph, which is used to build almost all subsequent intermediate views. Important data are the dominator information used in the construction of the SSA form and the process rank information. When building the final Petri net, only the operations graph is used, so they are quite similar structurally and differ in the level of detail.

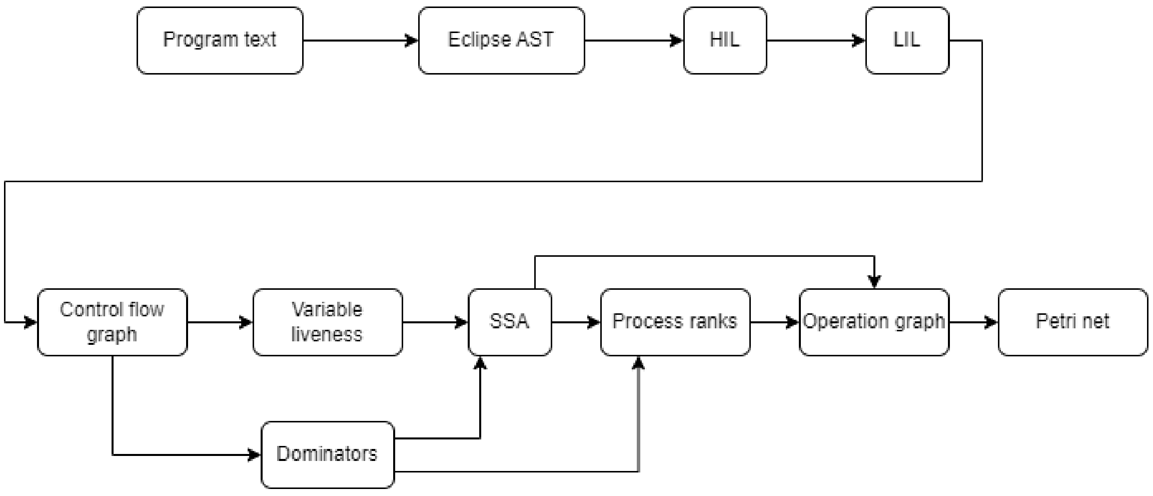

Figure 14 - Dependency of intermediate forms and data structures

The analyzer converts a Java program using MPJ Express into a Petri net in Tina format. The resulting net can be examined for deadlocks using the tools of the Tina package.

The analyzer is written in the Scala language, which runs on the JVM (Java Virtual Machine). The analyzer uses the Eclipse JDT library to parse Java program texts.

2.4 Parsing Java with Eclipse JDT

Eclipse is an integrated software development environment. It is developed and maintained by the Eclipse Foundation. Eclipse was originally developed as a successor to the IBM VisualAge development environment. Eclipse is open source software.

Development environments for various programming languages have been created on the basis of Eclipse: Java, C/C++, Python, PHP.

The Java Development Tools (JDT) library is a set of plug-ins that, together with the Eclipse platform, form a development environment for Java. This development environment has the following capabilities: creating and managing Java projects, editing code, automatic hints, code refactoring tools, tools for exploring the Java project model (for example, call hierarchy, inheritance hierarchy, reference search), project assembly, debugging Java programs, integration with various build systems (such as Maven), and others.

JDT consists of the following components:

– JDT APT - annotation processing;
– JDT Core is an infrastructure for working with code: builder, language model, code writing support, refactoring, and symbol search;
– JDT Debug is a Java application debugger;
– JDT Text - syntax highlighting, code formatting;
– JDT UI - implements a graphical user interface for working with Java projects.

Our analyzer uses the JDT Core component, in particular its parser. A parser is a component that converts a sequence of source code characters into an abstract syntax tree.

An abstract syntax tree (AST) is a finite set, labeled and oriented tree in which the internal nodes are mapped to the corresponding programming language operators and the leaves to the corresponding operands.

Let's take a look at how to work with the JDT parser.

```
def parse(source: String): CompilationUnit =
  val parser = ASTParser.newParser(AST.JLS16)
  val mpjJar = Path.of(System.getenv("MPJ_HOME"))
    .resolve("lib").resolve("mpj.jar")
  parser.setSource(source.toCharArray)
  parser.setKind(ASTParser.K_COMPILATION_UNIT)
  parser.setResolveBindings(true)
  parser.setEnvironment(Array(mpjJar.toString),
    Array.empty, Array.empty, true)
  parser.setUnitName("Unit name")
  parser.createAST(null).asInstanceOf[CompilationUnit]
```

The parser is created using ASTParser.newParser(). This method passes the version of the Java language with which the parser will work; in this case, version 16 is specified. A reference to the created parser is stored in the parser variable.

Next, the path to the MPJ library is searched using the MPJ_HOME environment variable.

The call parser.setSource(...) gives the parser the text of the program for which a syntax tree will be built.

The parser.setKind(...) call specifies what should be the result of the analysis. In this case, it is a compilation unit, in other cases it can be a separate statement or expression.

The call to parser.setResolveBindings(true) indicates that in addition to parsing the program, bindings also should be analyzed. Bindings connect the places where symbols, such as variables or methods, are used with the places where they are defined. This helps, for example, to determine the type of a variable at each point of its use.

Using the parser.setEnvironment(...) call, the parser receives information about the program's dependencies, here the MPJ library is such a dependency.

Next, we set the name for the compilation unit with parser.setUnit-Name() and get the result with parser.createAST().

So, the JDT library defines the Java parser and the Java syntax tree with all its nodes. In addition, JDT provides the ASTVisitor class. A visitor is a design pattern that traverses tree-like data structures. In JDT, such a data structure is a syntax tree with nodes whose base class is ASTNode. For each type of node, ASTVisitor has two methods defined - visit() and endVisit(). The first is called before visiting child nodes, the second is called after visiting child nodes. In Fig. 15 shows the class diagram for the "visitor" design pattern. Only a few of the ASTNode's inheritors are shown - CompilationUnit and IfStatement.

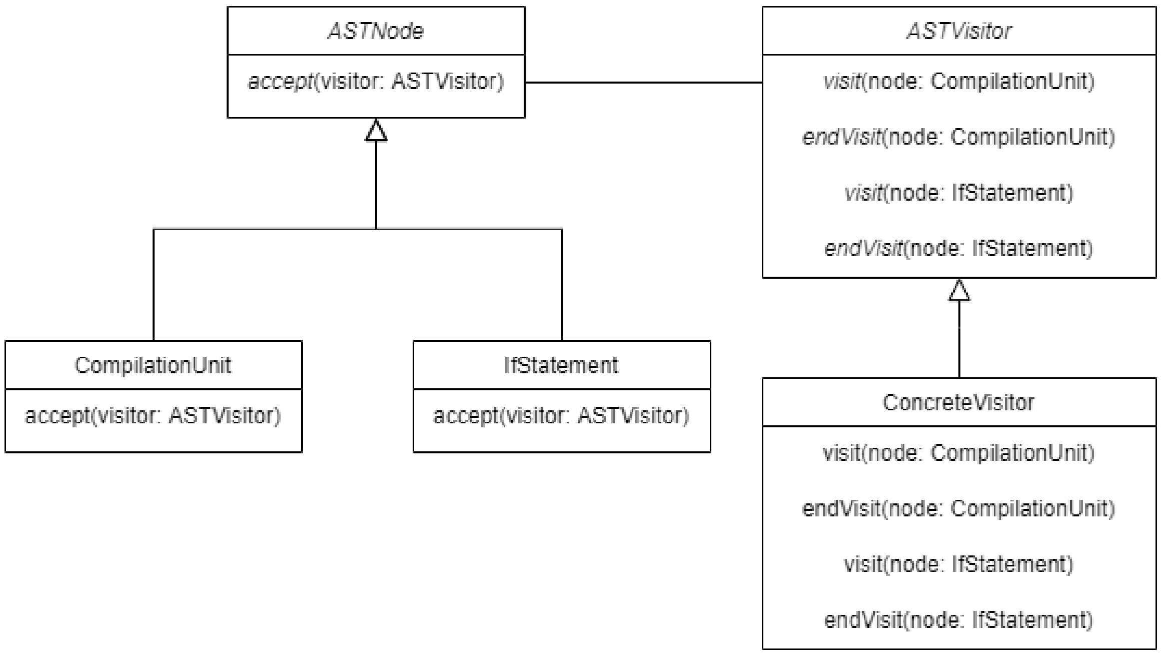
ASTNode
accept(visitor: ASTVisitor)
ASTVisitor
visit(node: CompilationUnit)
endVisit(node: CompilationUnit)
visit(node: IfStatement)
endVisit(node: IfStatement)
CompilationUnit
accept(visitor: ASTVisitor)
IfStatement
accept(visitor: ASTVisitor)
ConcreteVisitor
visit(node: CompilationUnit)
endVisit(node: CompilationUnit)
visit(node: IfStatement)
endVisit(node: IfStatement)

Figure 15 - Visitor template

To use the ASTVisitor class, one needs to create its inheritor class and override the visit() and endVisit() methods for the nodes of interest. Then to accept() method should be called. Creating and using a visitor is shown in the example below.

```
def apply(cu: CompilationUnit): Program =
  val visitor = new Visitor
  cu.accept(visitor)
  Program(visitor.getFuncs)
```

2.5 High-level intermediate language

HIL (higher level intermediate language) is a high-level intermediate language. HIL is created by transforming the syntax tree obtained from the JDT parser.

The highest type in HIL is a program. It consists of a list of function definitions (FuncDecl). Each function consists of a list of parameters and a code block (Block). The code block contains a sequence of instructions (Stmt - short for statement).

The instructions may be as follows:

– interrupt the cycle (Break);

– move to the next iteration of the cycle (Continue);

– return (Return);

– Assignment;

– declaration of a variable (VarDecl);

– function call instruction (CallStmt).

– conditional statement (IfThenElse);

– loop (WhileLoop);

– block.

Some instructions, such as assignment and return, contain expressions. Expressions can be simple or complex. Simple expressions include:

– access to a variable (Variable);

– constant (IntLiteral);

– access to a static field (StaticFieldAccess).

Complex expressions, unlike simple ones, can contain other expressions, but only simple ones. This limits expression nesting. Let's consider the types of complex expressions:

– unary expression (UnaryExpr);

– a function call (CallExpr). It differs from the CallStmt;

– a binary expression (BinaryExpr).

Below is the grammar for HIL.

```
<Program> ::= <FuncDecl>+
<FuncDecl> ::= "func" <Name> "(" [<Param> ["," <Param>]+] ")" ":" <Type>
<Block>
<Param> ::= <Name> : <Type>
<Stmt> ::= <Break> | <Continue> | <Return> | <Assignment> | <VarDecl> |
           <CallStmt> | <IfThenElse> | <WhileLoop> | <Block>
<Break> ::= "break"
<Continue> ::= "continue"
<Return> ::= "return" [<Expr>]
<Assignment> ::= <Name> "=" <Expr>
<VarDecl> ::= <Name> ":" <Type> ["=" <Expr>]
<CallStmt> ::= <CallExpr>
<IfThenElse> ::= "if" <Condition> <Stmt> ["else" <Stmt>]
<Condition> ::= "(" <SimpleExpr> ")"
<WhileLoop> ::= "while" <Block> <Condition> <Block>
<Block> ::= "{" <Stmt>+ "}"
```

The <Name> character means any valid Java identifier. Square brackets [...] indicate that the content of the element is optional. + means repeating one or more times; * means repeating zero or more times.

Now let's look at an example of converting Java code to HIL.

```
public class Example14 {
    static int func1(int a, int b, int c, int count) {
        for (int i = 0; i < count; i++) {
            a = a + b * c;
        }
        return a;
    }
}
```

Here we see a static method with a for loop that performs simple arithmetic calculations. HIL will look like this:

```
func Example14.func1(a: int, b: int, c: int, count: int): int {
  {
    var i: int = 0
    while {
      var t~1: boolean = i<count
    } (t~1) {
      var t~2: int = b*c
      a = a+t~2
      i = i+1
    }
  }
  return a
}
```

HIL is a simpler language than Java, so some Java constructs are translated into several simpler HIL constructs. The smaller number of constructs in the language greatly simplifies further work with it, because fewer cases will need to be considered for analysis.

In HIL, expression nesting is limited. In order to represent expressions with arbitrary nesting from Java to HIL, we use the addition of temporary variables that contain the intermediate results of the expression. In the HIL snippet above, these variables are t~1 and t~2. An example of expression conversion is the expression a+b*c. In HIL, we first create a temporary variable t~2 that stores the result b*c, and then perform the addition of a+t~2.

HIL does not have a for loop, so a while loop is used because a for loop can always be converted to a while loop. In the example above, there is a block before the loop condition. This block contains temporary variables that are used in the condition expression. The rest of the while loop has the usual parts: the loop condition and the loop body.

Another example of a conversion is the reduction of the increment i++ to the assignment i = i + 1.

One important transformation takes place over the HIL form: the addition of assert statements. These instructions are added at the beginning of the branches of conditional operations. They capture the condition expression of an if-then-else statement. For the then block, the condition itself is added, and for the else block, the negation of the condition is added.

This conversion is important because the assert statement provides information about which process makes calls to MPJ functions. For example, the

```
if (rank == 0) {
  MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
} else {
  MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, 0);
}
```

turns into

```
var t~1: boolean = rank==0
if (t~1) {
  assert t~1
  mpi.Comm.Send(mpi.MPI.COMM_WORLD, buffer, 0, 4, mpi.MPI.INT, 1, 0)
} else {
  assert !t~1
  mpi.Comm.Recv(mpi.MPI.COMM_WORLD, buffer, 0, 4, mpi.MPI.INT, 0, 0)
}
```

2.6 Low-level intermediate language

LIL (lower level intermediate language) is a low-level intermediate language. HIL and LIL forms have many similarities, but HIL is a higher level form.

In a LIL program, a function is divided into a set of basic blocks. A basic block [19] is a sequence of instructions that are always executed together. A basic block begins with a label and ends with a jump instruction.

Unlike HIL, LIL basic blocks cannot be nested. Each block gets its own name - a label. Conditional operations and loops from HIL are converted to jump instructions, also known as goto instructions.

LIL has the following transition instructions:

– unconditional jump (Jump);

– conditional jump (CondJump);

– return from a function (Return).

Below is the part of LIL grammar that differs from HIL.

```
<FuncDecl> ::= "func" <Name> "(" [<Param> ["," <Param>]+] ")" ":" <Type>
               <Block>+
<Block> ::= <Stmt>* <Transfer>
<Stmt> ::= <Assignment> | <CallStmt>
<Transfer> ::= <Jump> | <CondJump> | <Return>
<Jump> ::= "jump" <Name>
<CondJump> ::= "condJump" <SimpleExpr> <Name> <Name>
<Return> ::= "return" <SimpleExpr>
```

Here is an example of converting a fragment from Section 2.6 to LIL.

```
entry:
rank = mpi.Comm.Rank
buffer = new int[3]
t~1 = rank==0
condJump t~1 bb2 bb4
bb2:
mpi.Comm.Send
jump bb3
bb4:
t~2 = rank==1
condJump t~2 bb5 bb6
```

```
bb5:
mpi.Comm.Recv
jump bb6
bb6:
jump bb3
bb3:
jump end
end:
return
```

In this example, we see that execution starts with the basic entry block. It checks for the process rank and jumps to block bb2 if the rank is zero, or otherwise to block bb4. In the bb2 block, Send is called and then execution moves to the bb3 block, and then to the end, where the return function is called. Block bb4 is executed in a similar way.

2.7 Control flow graph

Based on the basic blocks of LIL, a control flow graph is built. The control flow graph [19] models the flow of control between the basic blocks in the program. It is a directed graph $G=(N,E)$. Each node $n\in N$ corresponds to a basic block. Each edge $e=(n_i,n_j)\in E$ corresponds to a possible transition of control from the block n_i to the block n_j.

Building a graph of the execution flow with LIL is as follows. A list of all edges in the graph is generated. To do this, information about the next blocks is obtained from each block using its jump instruction. For example, if the jump instruction is a simple jump, then an edge from the current block to the next block is added to the list. Then the graph object itself is created from the list of edges.

Since the data structure of a directed graph is used not only for the control flow graph, but also for other data, the analyzer uses the base class for directed graphs. Here is its definition.

```
class Graph[N](val edges: List[(N, N)]):
  protected val map: Map[N, List[N]] = edges.groupMap(_._1)(_._2)
  protected val reverseMap: Map[N, List[N]] = edges.groupMap(_._2)(_._1)

  val nodes: List[N] =
    edges.flatMap((n, m) => List(n, m)).distinct

  def successors(node: N): List[N] =
    map.getOrElse(node, List())

  def predecessors(node: N): List[N] =
    reverseMap.getOrElse(node, List())
```

Here, the standard parameter N is used, which is the type of nodes. In the control flow graph, nodes are represented by the String type. The graph is created from a set of arcs. Each arc is a start/end node pair. From this set, an associative array map is calculated, the key of which is a node, and the value is a list of subsequent nodes of this node. An inverse associative array is also built - from a node to its predecessor nodes. The successors() and predecessors() methods correspond to these two associative arrays.

Fig. 16 shows a control flow graph built for a program fragment:

```
if (rank == 0) {
  MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
} else {
  MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, 0);
}
```

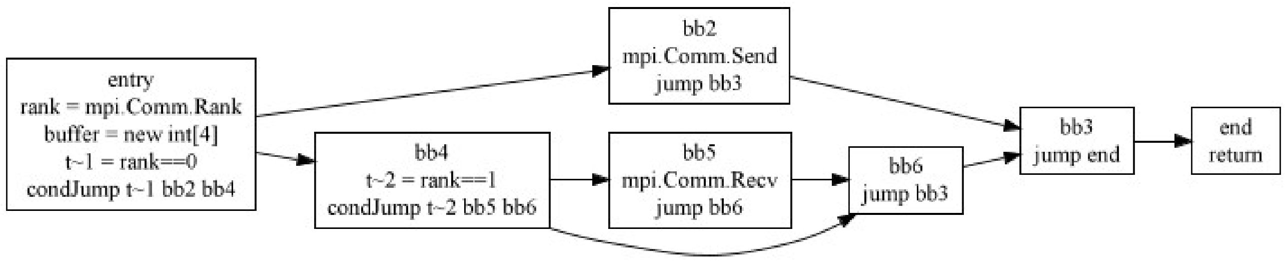

Figure 16 - Control flow graph for a simple program

2.8 Partial program evaluation

Partial evaluation [20] is a program optimization technique that consists in creating a specialized version of a program based on a part of the data provided. When the specialized program is run with another part of the data, the result will be the same as when the original program is run with the full data set (see Fig. 17).

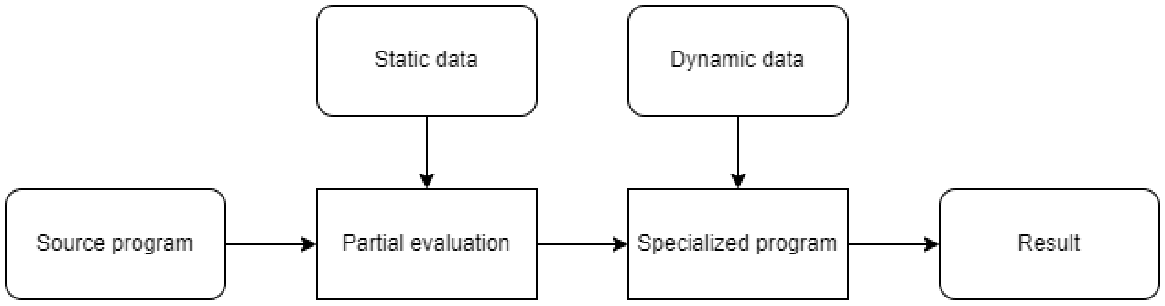

Figure 17 - Schematic of the partial calculation

In other words, partial evaluation allows to compute a part of the program at compile time using data that was known at that time. The main advantage of this optimization is that the specialized version of the program will not re-execute calculations that have already been performed at compile time. Reducing the number of calculations can significantly increase the speed of program execution.

A partial evaluation is performed on the LIL form. The purpose of this transformation in the analyzer is to process loops with a known number of iterations, that is, with the number of iterations that can be displayed when analyzing the code.

2.8.1 Separation of variables

During the analysis, it is not possible to fully compute the program because some of the data is unknown. Therefore, it is necessary to determine which part of the data is unknown. This process is called *variable* division [20] into:

– static - variables that can be determined during a partial evaluation, i.e. during analysis;
– dynamic - variables that are not static.

In other words, a dynamic variable is a variable that depends on other dynamic variables.

The process of separating variables is called binding *time analysis* because it determines at what time a variable value can be calculated, i.e., the time at which a value is associated with a variable.

Let's label all the variables in the program $x_1, \ldots, x_N$ and assume that the input variables are represented by the set $x_1, \ldots, x_n$, where $0 \leq n \leq N$. Suppose that the values of the binding times for the input variables are given $b'_1, \ldots, b'_n$where b'_j is either *S* (static variable) or *D* (dynamic variable). The task is to compute the separation for *all* variables in the program: $B = (b_1, \ldots, b_N)$ such that if $b'_i = D$. $b_i = D$. This task is achieved by the following algorithm:

1) Set the initial value for *B to be* equal to $(b'_1, \ldots, b'_n, S, \ldots, S)$that is, all variables except the input variables are considered static.
2) If the program contains an assignment $x_k = expr$assignment, and the variable x_j is used in the expression *expr* and $b_j = D$ in *B,* then set $b_k = D$ in *B*.
3) Repeat step 2 until *B* changes. Then the algorithm ends with the division of *B*.

2.8.2 Program transformation

After calculating the separation of variables, program transformation can be started. The purpose of this transformation is to make all the calculations that are possible given the known static data and change the program accordingly. For these calculations a store is used. It contains value of static variables.

To transform a program it is necessary to know which parts of the program should be changed. To do this, a list of basic blocks is used. In this case, the same block can be executed with two different storages. An example is two different iterations of a loop, during which the values of variables may be different. Therefore, the conversion algorithm uses a working list of pairs (label, store), where label is the label of the base block, store is the storage with which this base block will be calculated.

1) Set the initial value of the worklist worklist = {(entry, store0)}, where entry is the label of the initial base block, store0 = empty storage.
2) If the worklist is empty, end the algorithm.
3) Take a pair (label, store) from the list.
4) If the pair is marked, go to step 2.
5) Mark the pair.
6) Generate code for the block labeled label.
7) Add the following pairs to the worklist.
8) Go to step 2.

During the generation of new code, each instruction in the basic block is sequentially reviewed, a new instruction is created if necessary, and the repository is modified. The code generation for instructions is shown in detail in Table 4.

Table 4 - Code generation for instructions

Instruction	Action during analysis	Generated instructions
`x = exp` (if x is dynamic)	`reducedExp = reduce(exp, store)`	`x = reducedExp`
`x = exp` (if x is static)	`val = evaluate(exp, store)` `store = store[x → val]`	---
`jump label`	`newLabel = generateLabel(label, store)`	`jump newLabel`
`condJump exp l1 l2` (if the `exp` expression is static)	`reducedExp = reduce(exp, store)`	`condJump reducedExp l1 l2`
`condJump exp l1 l2` (if the exp expression is static and val = true)	`val = evaluate(exp, store)`	`jump l1`
`condJump exp l1 l2` (if the exp expression is static and val = false)	`val = evaluate(exp, store)`	`jump l2`

If an instruction is not shown in Table 4, it means that it is added to the generated code without changes. The following auxiliary functions are used:

– reduce(exp, store) - performs constant folding of the static parts of the dynamic expression exp, for example, if b = 2, then the expression b * b + a can be replaced with 4 + a;
– evaluate(exp, store) - evaluates the value of the static expression exp;
– generateLabel(label, store) - creates a new label or returns an existing one for the pair (label, store).

After generating the code for the basic block, the store can be changed if there are assignments to static variables. Next, it is necessary to determine which blocks will be processed after the current block. This choice is described in detail in Table 5. The following blocks are added to the worklist along with the updated store (denoted as store').

Table 5 - Selection of the following blocks

Transition instruction	Condition	Result
`return`	---	`{}`
`jump label`	---	`{(label, store')}`
`condJump exp l1 l2`	exp - dynamic expression	`{(l1, store'), (l2, store')}`
`condJump exp l1 l2`	exp is evaluated to true	`{(l1, store')}`
`condJump exp l1 l2`	exp is calculated as false	`{(l2, store')}`

2.9 Data flow analysis

Dataflow analysis is a type of program analysis that provides information about how data will move during program execution. Data flow problems are formulated based on a set of equations over sets that are associated with nodes and edges in the control flow graph.

There are two types of data flow analyzes:

– forward analysis - if the information propagates in the direction of the edges in the control flow graph;

– backward analysis - if the information propagates in the reverse direction.

The analyzer implements the analysis of the variables' liveness, which belongs to the family of data flow analyzes. Other examples of data flow analysis are [20]:

– constant propagation;

– common subexpression elimination;
– elimination of redundant register load operations;
– reaching definition analysis.

2.10 Variable liveness analysis

Liveness analysis is performed on the LIL form. This information will be used to convert the LIL to SSA. Liveness analysis belongs to the family of data flow analysis algorithms.

Liveness analysis is a backward analysis. The following is a formal definition of liveness for a variable, followed by a description of the algorithm.

A variable *v* is live at a program point *p* if there is a path from *p* to its use that does not assign a new value to *v*. Let us define $LiveOut(n)$ as the set of variables that are alive when exiting the basic block *n*. To calculate this set, we will use the following equation

$$\forall\ m \in successors(n) UEVar(m) \cup \left(LiveOut(m) \setminus VarKill(m)\right)$$

The following additional definitions are used here:
– $successors(n)$ - is the set of all subsequent blocks for *n*;
– $UEVar(m)$ - is the set of all variables that are used in the next block *m* before being redefined in *m*;
– $VarKill(m)$ - is a set of variables that are overridden in block *m*.

Let's take a closer look at formula (1). $LiveOut(n)$ - is the union of all the variables that are alive at the beginning of some block *m, which* immediately follows the block *n*. In other words, a variable *v* belongs to

$LiveOut(n)$if it is alive at the beginning of at least one subsequent block, not necessarily for all subsequent blocks.

Then, the contribution of each subsequent block m is determined by this expression:

$$UEVar(m) \cup (LiveOut(m) \setminus VarKill(m))$$

In order for the variable *v to* be live at the beginning of block *m, it is* necessary that

– it was either used in block *m* before being redefined in *m, i.*e., *v* belongs to the set $UEVar(m)$;
– or it is alive at the exit of block *m* and is not redefined in *m, i.*e. $v \in LiveOut(m) \setminus VarKill(m)$.

By combining these conditions using the join operator, we get the contribution for one of the following blocks *n*. Information propagates from subsequent blocks to the previous ones; that is why the liveness analysis has a backward direction.

The following algorithm is used to calculate *LiveOut* sets.

1) Initialize the sets $UEVar(n)$i $VarKill(n)$for each block *n* in the control flow graph.
2) Initialize the worklist worklist with the exit node in the control flow graph, assign $LiveOut(exit) = \varnothing$.
3) If the worklist is empty, end the algorithm.
4) Get the basic block n from the worklist.
5) Recalculate $LiveOut(n)$ based on the information for the following blocks.
6) If the value $LiveOut(n)$ value has changed, add all previous blocks

to the worklist.

7) Go to step 3.

Initialization $UEVar(n)$ i $VarKill(n)$ initialization occurs once, since this information does not change during the algorithm. To calculate these sets, the definition and use of variables are processed.

The algorithm will terminate because the *LiveOut* sets are finite, since the number of variables in the function is finite. Repeatedly calculating *LiveOut* for a block can only increase this set. Of course, formula (1) excludes the *VarKill* set, but its size does not change during the algorithm. Therefore, over time, the *LiveOut sets* will stop changing and the algorithm will terminate.

2.11 Dominators calculation

One block *dominates* another if any path from the input block to the second block passes through the first block. *A* dominator is a block that dominates a given block.

Dominators are calculated based on the control flow graph. The information about dominators is used in the subsequent stages of the analyzer during the construction of the SSA form and during the predicates propagation, which will be described in the following subsections.

Fig. 18 shows an example of a control flow graph with basic blocks A, B, C, D, E, F, and G. Table 6 shows the dominators for each block in this graph.

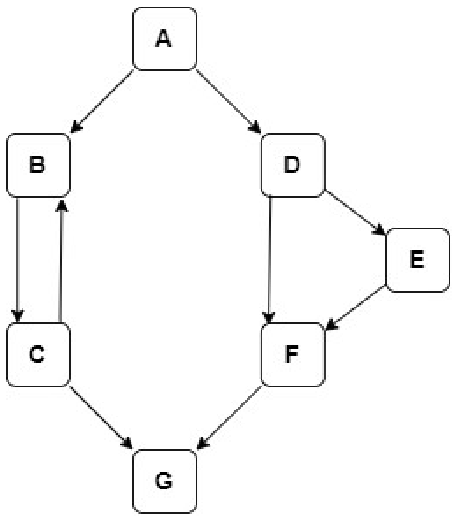

Figure 18 - Example of a control flow graph

Various algorithms have been proposed to calculate dominators. In the following, we will consider an algorithm that belongs to the family of data stream analysis. Before presenting the algorithm itself, it is necessary to define the equations of the dominator sets.

Table 6 - Blocks and their dominators

Block	Dominators
A	{A}
B	{A, B}
C	{A, B, C}
D	{A, D}
E	{A, D, E}
F	{A, D, F}
G	{A, G}

To denote the set of blocks that dominate a given block b, we use the notation $Dom(b)$. $Dom(b)$ has the following properties:

– block *n* in the control flow graph dominates over block *b* if *n* lies on every path from the input block to block *b*;

– by definition, $b \in Dom(b)$;

– $Dom(b)$ contains every block *n* that dominates *b*;

– if $x, y \in Dom(b)$, then either $x \in Dom(y)$or $y \in Dom(x)$.

Based on the above properties, let's form the data flow equations for *Dom*. In the equations below, we assume that n_0 - is the input block of the graph.

$$Dom(n_0)=\{n_0\}$$

$$Dom(n)=\{n\} \bigcup_{\forall\ m \in\ predecessors(n)} Dom(p)$$

The set $Dom(n_0)$ for an input node consists of a single element - the input node itself.

For the rest of the nodes $Dom(n)$ is calculated based on the predecessors. A node $x \in Dom(n)$ is dominant only if it is dominant for all predecessors of block *n*. This is stated by the right-hand side of the union in equation (2). The left part adds the *node n* itself to the set $Dom(n)$.

Blocks in the control graph can be numbered in several ways. The post-order is the order in which all subsequent blocks are visited, as far as possible, before a particular block is visited. The opposite of post-order, which we will refer to as RPO, is the order in which the previous blocks are visited as far as possible before the current block. Most graphs have more than one RPO numbering, but the main specified property of interest is preserved.

Table 7 - Numbering order of the control flow graph

	A	B	C	D	E	F	G
Reverse order	7	3	2	6	5	4	1
RPO	1	5	6	2	3	4	7

Both of these orders can be calculated using a depth-first search of the graph. An example of block numbering in the graph in Fig. 18 is shown in Table 7.

Based on the equations, let's define the algorithm for calculating Dom(n). This is an algorithm for analyzing the data flow in the forward direction, as information propagates from predecessors to subsequent blocks. Here N is the set of all blocks in the control flow graph.

1) Number all blocks in RPO order.
2) For each block *n*, set the initial value $Dom(n)=\{1,\dots,|N|\}$.
3) Calculate $Dom(n)$ for each block *n* in the RPO order according to formula (2).
4) If any of the sets $Dom(n)$ has changed, go to step 3.
5) Otherwise, terminate the algorithm.

The reason why blocks are visited in RPO order is that information must propagate from previous blocks to subsequent blocks. This reduces the number of iterations of the algorithm and, therefore, improves its performance. An example of the algorithm is shown in Table 8.

Two iterations are enough to calculate the dominators in this example.

After the second iteration, no changes to the sets $Dom(n)$ sets and the algorithm terminates.

Table 8 - An example of the dominator search algorithm

	A	B	C	D	E	F	G
RPO	1	5	6	2	3	4	7
After initialization	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}
Iteration 1	{1}	{1,5}	{1,5,6}	{1,2}	{1,2,3}	{1,2,4}	{1,7}
Iteration 2	{1}	{1,5}	{1,5,6}	{1,2}	{1,2,3}	{1,2,4}	{1,7}

We will show that the algorithm terminates in the general case. At the beginning of the algorithm, the sets $Dom(n)$ are initialized with the value $\{1,\dots,|N|\}$where *N is the* set of all blocks. At each iteration, for each block *n,* $Dom(n)$ can either decrease because it decreases $Dom(p)$ for one of the predecessors *p of* block *n,* or remain the same size. The length of the *Dom sets is* limited by the number of blocks in the graph, and the *Dom* sets can only change downward, so at some iteration the sets will stop changing and the algorithm will terminate.

2.12 SSA form

Static single assignment (SSA) is a form in which each variable in the program text is assigned a value no more than once. In the analyzer, SSA is used later to propagate predicates.

The SSA shape was developed in the late 1980s [21] as an intermediate representation that allowed for efficient and accurate data flow analysis. Since then, the SSA shape has rapidly gained popularity due to its intuitive nature and simple algorithm. Today, the SSA form is used in most commercial and open-source compilers, such as GCC, LLVM, HotSpot (Java virtual machine), and V8 (Javascript engine).

The advantage of the SSA form is that it significantly reduces the cost of building and using def-use chains. The reason is that in the SSA form, each variable has a single definition location. Therefore, in def-use chains, there are much fewer links between the definitions and uses of variables, and the time to find the necessary information about variables is reduced.

Another advantage of SSA is that the definition and usage information can be easily updated. This is important because when applying optimizations such as constant propagation, the program code changes, some blocks may be removed, and this information must be updated.

To achieve the uniqueness of variable definitions, SSA uses:

– insertion of φ-functions;

– renaming variables.

The φ-function is a special design of the SSA form. It allows to combine a stream of values from different blocks into one variable. φ-functions are inserted at so-called join points. Join points are blocks in the control flow graph that have more than one preceding block; they are where value streams from different blocks converge.

Fig. 19 shows an example of a situation where a φ-function should be inserted in block B3. This is necessary because in blocks B1 and B2, the variable x is assigned a value that is then used in block B3. During program execution, the control flow can enter both block B1 and block B2. In order for further analysis to take into account these two possibilities, a φ-function is added at the beginning of block B3.

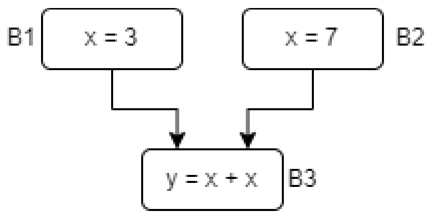

Figure 19 - A fragment of the control flow graph

An example of conversion to the SSA form is shown in Fig. 20: the φ-function is added in block B3, the variable x is renamed in the declaration and use places.

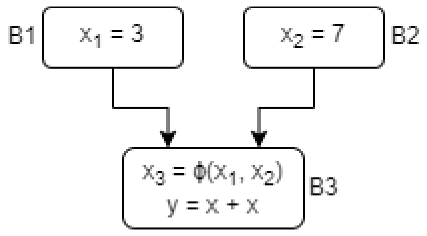

Figure 20 - Adding a φ-function

SSA can be divided into two parts, which will be discussed below:
arrangement of φ-functions;
renaming variables.

2.12.1 Placement of φ-functions

A simple solution for placing φ-functions would be to insert them at all the vanishing points. However, most φ-functions would be redundant in this case. A large number of φ-functions will only complicate and slow down the subsequent analysis, since it has to take into account all the extra φ-functions. For an effective analysis, it is necessary to determine where φ-functions are really needed.

Next, we consider the concepts that will be used in the arrangement of φ-functions. Recall that a basic block *n* dominates a basic block *m* if every path from the input block to the output block m passes through block *n*. Block *n* strictly dominates block *m* if *n is* not equal to *m*.

A block *n* is called the *direct dominator* of a block *m* if it is the closest block to *m* that strictly dominates *m*. The input block does not have a direct dominator. All other blocks in the control flow graph have exactly one direct dominator.

The *dominance frontier* for a block *n is* the set of blocks *m* such that block *m* has a preceding block *k* such that *n* strictly dominates *k,* but *n* does not strictly dominate *m*. The dominance frontier for block *n is* denoted by $DF(n)$. To calculate the dominance bounds, we use the sets $Dom(n)$which were described earlier.

Suppose that the variable *x is* defined in block *n*. Also suppose that *m* is one of the blocks that is strictly dominated by block *n*; *m* has a use of variable *x*. If there is a block *p* between blocks *n* and *m* that has a redefinition of variable *x,* then a φ-function should be added for *x*. If there is no such block *p,* then a φ-function is not needed, because block *m* will always contain the value of the variable *x* that was defined in block *n*. In addition, insertion is necessary in places where the dominance region of a given block ends, because a path can be laid to these places that does not include block *n*.

So, if in block *n* the variable *x* is assigned, then it means that a φ-function should be added at the beginning of each block from the set $DF(n)$. However, when a φ-function is added, a new definition for the variable occurs, so the process of finding the dominance frontier must be continued for the block in which the φ-function was added [22].

To arrange φ-functions, the following algorithm can be used. It is executed for each variable used in the program.

1) Let *x be the* variable for which the φ-functions will be evaluated.
2) Initialize the worklist of blocks worklist. For each block *n,* which has an assignment to the variable *x*, add to the worklist the set $DF(n)$.
3) If the worklist is empty, end the algorithm.
4) Take a block from the worklist and call it *m*.
5) If the block *m is* marked, go to step 3.
6) If there is no assignment to *x in the* block - add to worklist $DF(m)$.
7) If the variable is live at the beginning of block *m,* add a φ-function for the variable *x* at the beginning of block *m*.
8) Mark the block *m*.
9) Go to step 3.

Note that the above algorithm uses information about the variables' liveness. Starting from the blocks in which the variable is defined, the algorithm moves from blocks to their dominance bounds and adds φ-functions in the right places. Processed blocks are marked so that they do not have to be processed again.

The analyzer's implementation uses a modified version of φ-functions that retains their meaning as described above. Each block that is supposed to have φ-functions initially receives a set of variables instead - a set of parameters. At the points where transitions to such blocks occur, it is indicated which variables will be passed to the next block, i.e., a set of arguments is formed. In this case, each block is made similar to a function definition, and a transition to a block is a transition to a function without return.

2.12.2 Variable renaming

During variable renaming the name of the variable in the source program is taken as the basis and a unique number is added to this name. Thus, the new name consists of the old name and a unique number.

To rename variables the analyzer uses a special data structure called dominator tree [22]. The set of nodes in the dominator tree coincides with the set of blocks in the control flow graph. If there are blocks *n* and m connected by an edge in the dominator tree, then block *n* is the direct dominator of block *m*. The root of the dominator tree is the input block. In Fig. 21 shows an example of a control graph and the corresponding dominator tree.

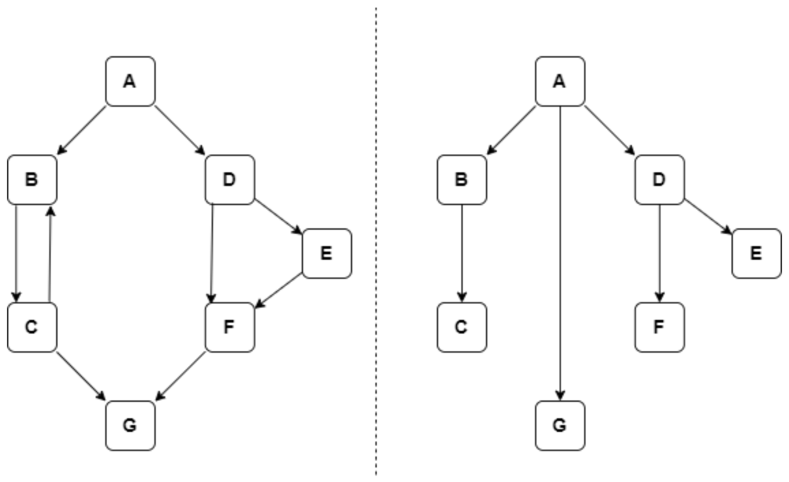

Figure 21 - Control flow graph (left) and the corresponding dominator tree

A stack and a counter are used for each variable taken from the source program. Blocks are visited in the order in which they are located in the dominator tree. When processing assignment instructions, a new name is created for the variable, the counter is incremented, and the new name is added to the stack. All uses of the variable in this block are replaced by the new name from the stack. After processing all the instructions, the algorithm proceeds to the heirs in the dominator tree. When all the heirs are processed, the name is removed from the stack.

Fig. 22 shows an example of converting LIL to SSA. All variables in the SSA form receive a unique number in addition to the name. Blocks bb2, bb6 receive parameters for the variables x, y. The variable are passed from the previous blocks.

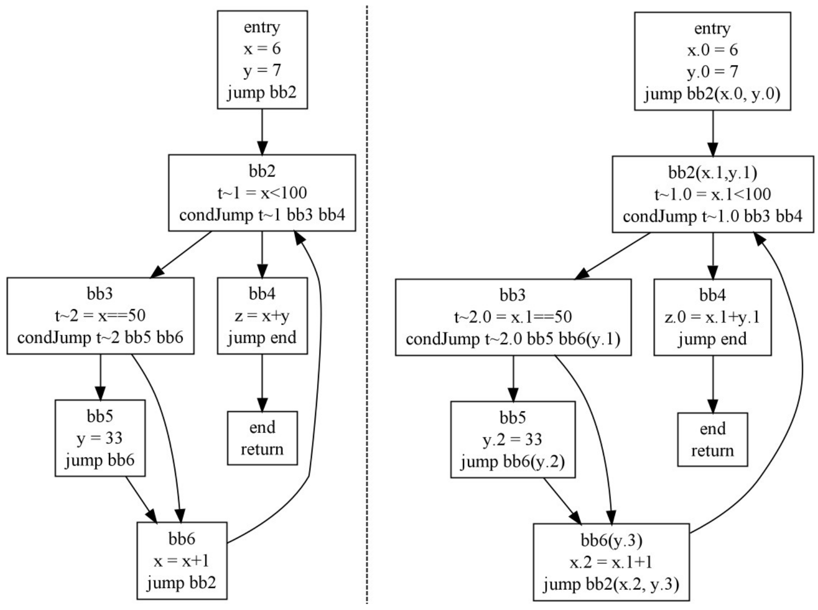

Figure 22 - Conversion of LIL (left) to SSA (right)

2.13 Predicate propagation

The next phase after conversion to SSA form is *predicate propagation*. Predicate propagation was developed during the course of this work The goal of this phase is to know which basic blocks will be executed by which process. This information is then used when building the operation graph.

Let's look at a code sample that is often used when writing MPI programs.

```
int rank = MPI.COMM_WORLD.Rank();
if (rank == 0) {
    MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
} else if (rank == 1) {
    MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, 0);
}
```

Here, the Rank method is called and the process rank is stored in the rank variable. Then the current process is checked for its rank. Thanks to the conditions of the if statements, it is possible to understand that Send will be executed by process 0, and Recv will be executed by process 1. Fig. 23 shows the corresponding control flow graph.

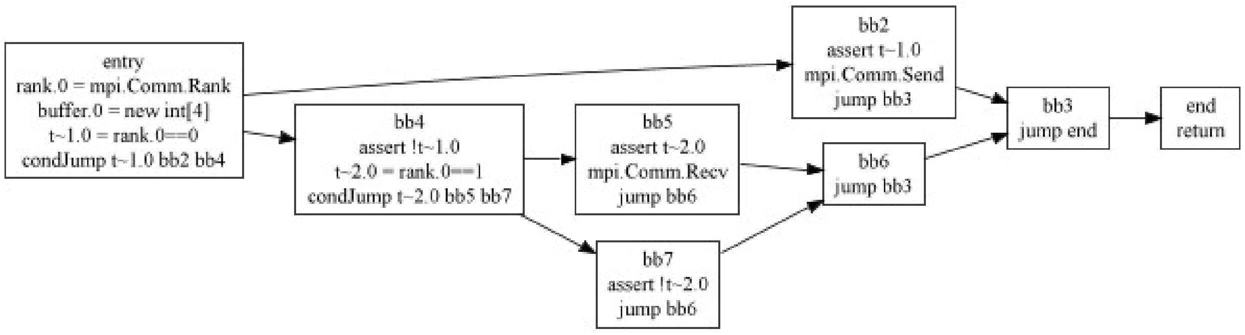

Figure 23 - Control flow graph for a code snippet with MPI operations

During predicate propagation, assert statements are used, which were added earlier. Assert statements specify the condition under which a block will be executed. We can calculate this condition and try to get information about the process rank from it. For example, block bb2 starts with the assert statement t~1.0. The variable t~1.0 contains the value true if the variable rank. 0 variable is equal to 0. The variable rank.0 is assigned the result of calling the Rank method. Hence, we can conclude that block bb2 is executed by a process with rank 0. Similarly, assert statements provide information about the rank of other processes.

The control flow graph in Fig. 23 contains blocks in the form of SSAs. This makes it easy to find what values are assigned to each variable, since each variable has only one definition location.

Here is a description of the predicate propagation algorithm.

1) For each block, its assert statement is considered.
2) Using the definition information, the process rank is calculated.
3) If information about the process rank cannot be retrieved, go to step 1 to the next block.
4) Associate the current block with the received rank.
5) Associate each block that is dominated by the current block with the resulting rank.
6) Go to step 1 to the next block.

This algorithm uses the dominators information described earlier. The reason why all blocks dominated by the current block are associated is so that the process does not change during program execution. If a block is identified by a process that executes it, then the same process will execute all the blocks it dominates.

As a result of the algorithm for the control flow graph in Fig. 23, it will be calculated that block bb2 will be executed by process 0, and block bb5 will be executed by process 1.

2.14 Operation graph

The *operation graph* is an oriented graph. This graph shows the sequence in which Send and Recv operations are run, as well as which processes run these operations. The operation graph is the last intermediate representation of the program before creating a Petri net for the program. The operation graph was also developed in the course of this work. The construction of the operation graph is based on the control flow graph.

The operation graph is a high-level representation because it ignores many program details. For example, the operation graph does not include information about variables, assignments, and conditions for executing basic blocks. The most important information is highlighted - how processes exchange messages with each other.

The operation graph consists of the following main nodes:

– send node - corresponds to a call to the Send operation;
– recv node - corresponds to a call to the Recv operation;
– intermediate node - connects the Send and Recv nodes and other intermediate nodes;
– split and join nodes - used when different operations can be called by one process under different conditions.

Here is a code snippet for which the operation graph will be built.

```
if (rank == 0) {
    MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, 0);
    MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 1, 0);
} else if (rank == 1) {
    MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, 0);
    MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 0, 0, 0);
}
```

In Fig. 24 shows the control flow graph for the fragment on the left, and the operation graph on the right.

Note that intermediate nodes are created on the basis of basic blocks. For example, the input node in the operation graph "Intermediate entry_3" is linked to the input node in the control flow graph.

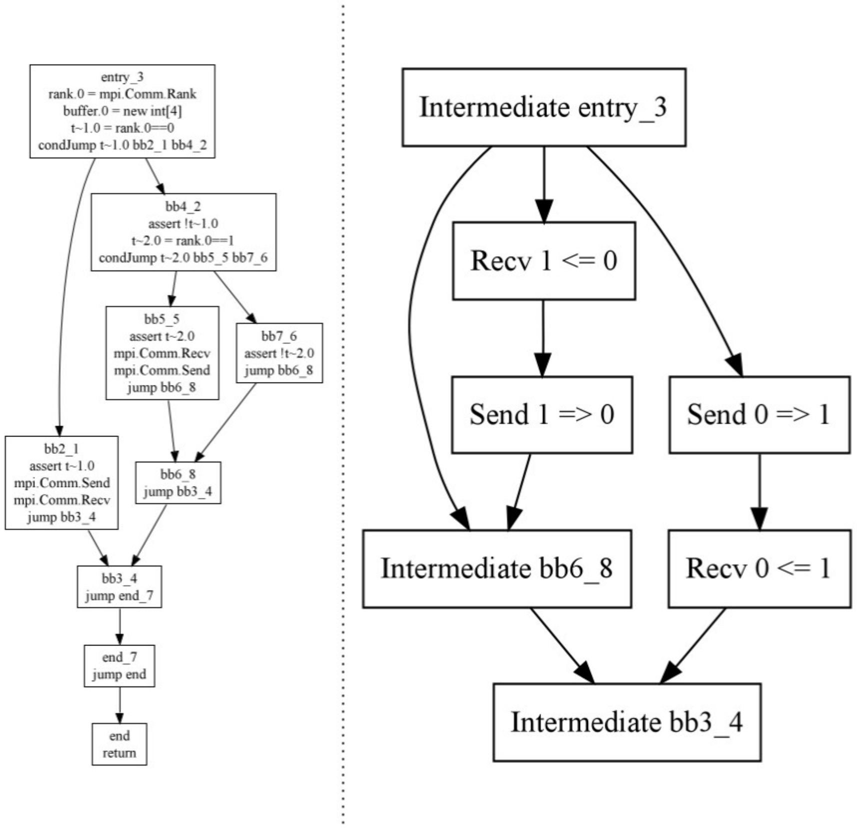

Figure 24 - Control flow graph (left) and corresponding operation graph (right)

If more than one node comes out of an intermediate node, it means that there is a split into processes. For example, the nodes "Recv 1 <= 0" and "Send 0 => 1" come out of the mentioned input node. Therefore, there is a division into processes 1 and 0; each branch will be executed by a separate process.

If more than one node converges to an intermediate node, it means that a common area for several processes begins. For example, the node "Intermediate bb6_8" is joined by the node "Send 1 => 0", which is executed by process 1. In addition, the input node is also joined by the node "Intermediate bb6_8". In this case, this means that if the process rank is not 0 and 1, then the process goes from the input node directly to the "Intermediate bb6_8" node.

Next, we consider an algorithm for transforming a control flow graph into an operations graph. It uses the association between the basic blocks and process ranks that was calculated during the predicate propagation. As in many algorithms described in this paper, a worklist is used here, each element of which is a pair (label, node), where label is the label of the basic block to be processed, and node is the node of the operation graph from which the edge to the new node will be sent.

1) Initialize the worklist with a pair (entry, node0), where entry is the label of the input basic block, node0 is the intermediate node corresponding to the input block. node0 is the starting node for the operation graph.
2) If the worklist is empty, the algorithm ends.
3) Otherwise, take the following pair (label, node) from the worklist. Let block be a basic block with a label.
4) If the block has more than one previous block, create an intermediate node new, add an edge from node to new, and assign node = new.
5) If there is no process rank for the block labeled label, then execute addNextBlocks(label, node) and go to step 2.

6) Review each instruction in the block, add Send and Recv nodes, and add edges for the created nodes. Let lastNode be the last node created, or node if no node was created. Execute addNextBlocks(label, lastNode).
7) Go to step 2.

The algorithm uses the addNextBlocks(label, node) helper function, which adds pairs (next, node) to the worklist for each subsequent block with the next label.

2.15 Building a Petri net from an operation graph

The final intermediate representation in the analyzer is a Petri net, which is eventually written to a . net file of the Tina format. The Petri net is built from the graph of operations.

Fig. 25 shows the operation graph (left) and the corresponding Petri net (right).

The Petri net contains many more elements than the operation graph. The Petri net is a lower-level representation and requires that the semantics of the program, or rather the semantics of MPI operation calls, be shown more explicitly. However, the structure of the two forms is quite similar.

To build a Petri net, an algorithm similar to the one used to build an operation graph is used. It also uses a working list of pairs (ogNode, transition), where ogNode is a node from the operation graph, and transition is the next transition in the Petri net.

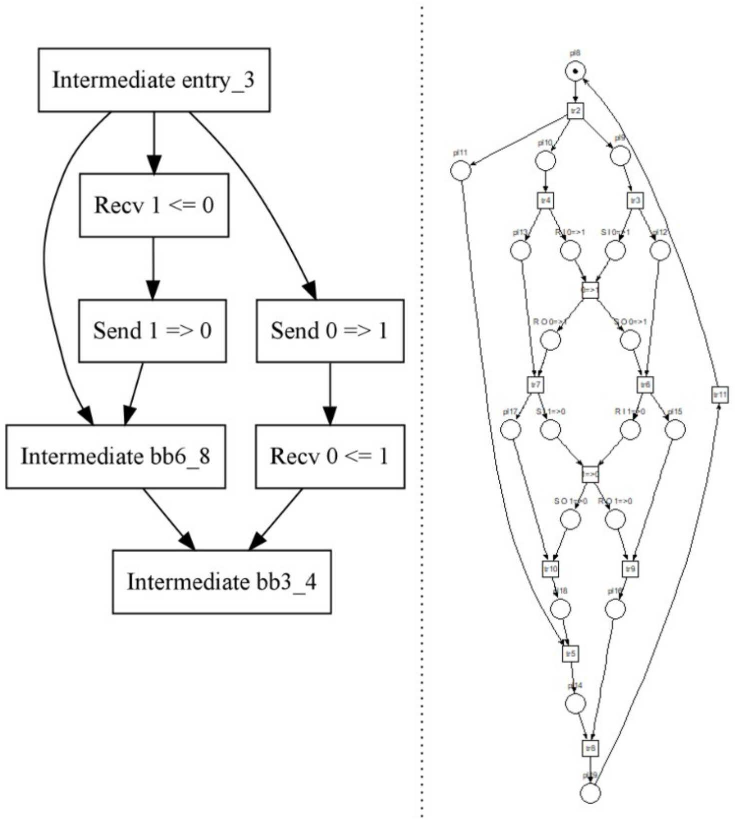

Figure 25 - Operation graph (left) and the Petri net built for it (right)

3 ANALYSIS OF THE RESULTS

3.1 A simple example of a program without deadlocks

This subsection describes how to run the analyzer and analyze the results of its launch for a simple example of a correct MPI program that has no deadlocks. The program is shown below.

```
import mpi.MPI;
public class MpiSendRecvSimple {
  public static void main(String[] args) {
    MPI.Init(args);
    int rank = MPI.COMM_WORLD.Rank();
    int[] buffer = {0, 2, 3, 4};
    int tag = 1;
    if (rank == 0) {
      MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 1, tag);
      MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 1, tag);
    } else if (rank == 1) {
      MPI.COMM_WORLD.Recv(buffer, 0, 4, MPI.INT, 0, tag);
      MPI.COMM_WORLD.Send(buffer, 0, 4, MPI.INT, 0, tag);
    }
    MPI.Finalize();
  }
}
```

In this program, deadlocks should never occur because the processes call MPI operations in the correct sequence: first, process 0 sends a message to process 1, then process 1 sends a message to process 0.

The analyzer was implemented in Scala, so a tool called sbt (simple build tool) is used to build and run it. sbt is also used to manage project dependencies, i.e., downloading the libraries required for the project. Libraries are described in the build.sbt file.

First of all, sbt shell should be started from the root directory of the project:

```
deadlock-finder > sbt
[info] welcome to sbt 1.6.2 (Azul Systems, Inc. Java 17.0.2)
[info] loading global plugins from ...
....
```

```
[info] sbt server started at local:sbt-server-8374cb68f19317221c80
[info] started sbt server
```

Next, the analyzer is launched using the run command with the path to the input Java file:

```
sbt:deadlock-finder> run examples/parallel/MpiSendRecvSimple.java
[info] running deadlockFinder.Main
examples/parallel/MpiSendRecvSimple.java
[success] Total time: 2 s, completed on Nov. 26. 2022 г., 08:25:43
```

The analyzer successfully completes its work. The built Petri net is saved at the path target\net.net. This net is shown in Fig. 26. In this simple example, the correctness of the program can be seen even visually, because there are no mutual simultaneous dependencies that cause deadlocks.

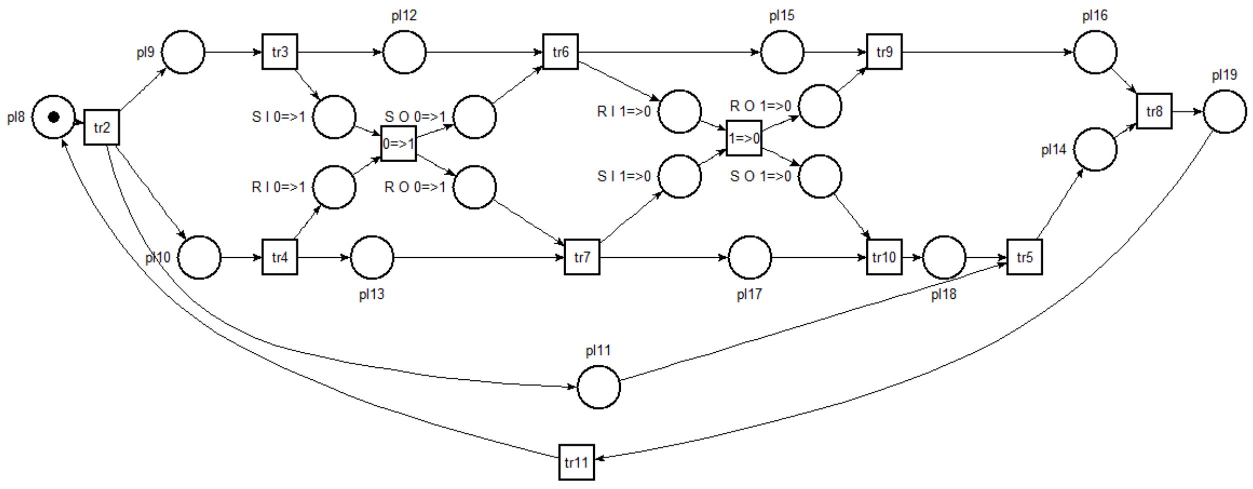

Figure 26 - An example of a Petri net for a correct program

The sift utility from the Tina package will be used to check for deadlocks. sift is a high-performance state space explorer and verifier.

```
deadlock-finder> C:\programs\tina-3.7.0\bin\sift.exe .\target\net.net -
dead
```

```
16 marking(s), 20 transitions(s)
0.000s
```

For the sift utility, a Petri net is specified and the -dead argument indicates that sift should search for dead ends in the specified net. The output of the utility does not indicate the presence of deadlocks; it only shows the number of labels (16) and the number of transitions (20) in the state space. This means that this MPI program is correct.

3.2 A simple example of a program with a deadlock

Below is a program snippet with a deadlock.

```
int rank = MPI.COMM_WORLD.Rank();
if (rank == 0) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 1, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 1, tag);
} else if (rank == 1) {
  MPI.COMM_WORLD.Recv(buffer, offset, count, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buffer, offset, count, MPI.INT, 0, tag);
}
```

The analyzer builds a Petri net for it (see Figure 27) in the same way as shown in the previous subsection.

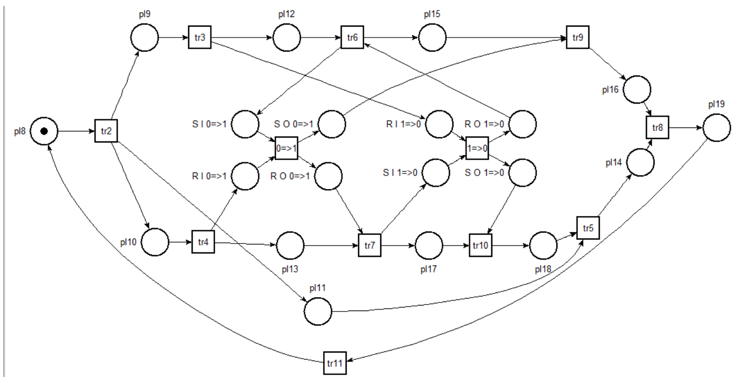

Figure 27 - Petri net for a program with a deadlock

After running sift, the following result is displayed:

```
deadlock-finder> C:\programs\tina-3.7.0\bin\sift.exe .\target\net.net -
dead
some state violates condition -f:
  {R I 0=>1} {R I 1=>0} pl11 pl12 pl13
  firable:
5 marking(s), 5 transitions(s)
0.000s
```

The results are consistent with the description in subsection 2.2. sift indicates a marking that has no allowed (firable) transitions. At the marking "{R I 0=>1} {R I 1=>0} pl11 pl12 pl13" a deadlock occurs in the net. The tokens in the places "R I 0=>1" and "R I 1=>0" indicate that Recv operations were called by both processes 0 and 1.

3.3 Example with a simple loop

This section shows an example of a program with a loop. The analyzer can process loops whose conditions can be calculated at compile time due to partial program evaluation (see Section 2.8). Below is a program snippet.

```
int rank = MPI.COMM_WORLD.Rank();
if (rank == 0) {
  int counter = 0;
  for (int i = 0; i < 2; i++) {
    MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 1, tag);
    counter++;
  }
  for (int j = 0; j < counter; j++) {
    MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, 1, tag);
  }
} else if (rank == 1) {
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 0, tag);
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 0, tag);
}
```

In this fragment, process 0 executes two loops. The first loop is executed twice, in which a Send call is made to process 1. Also in the first loop, the counter is incremented, which will be used as a condition for the second loop. Thus, the condition of the second loop depends on the first loop. In the second loop, a Recv call is made from process 1. Process 1 executes the corresponding Recv and Send calls, but without using loops.

To confirm that the cycles for process 0 are being processed correctly, Fig. 28 shows the operation graph.

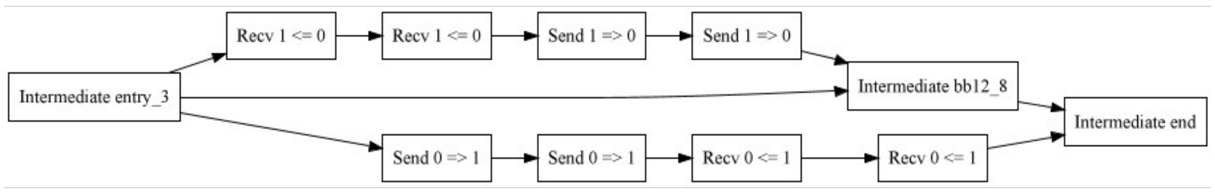

Figure 28 - Operation graph for a program with a loop

The Petri net built with the analyzer is shown in Fig. 29.

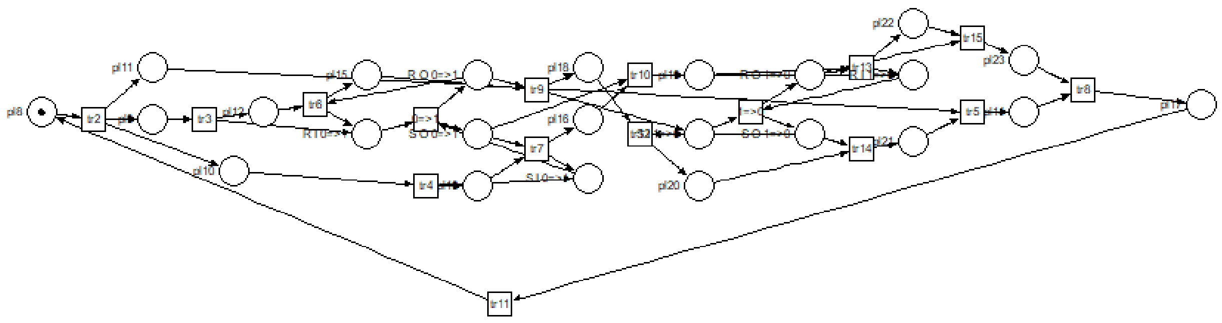

Figure 29 - Petri net for a program with a loop

The sift utility produces the following result:

```
deadlock-finder> C:\programs\tina-3.7.0\bin\sift.exe .\target\net.net -
dead
24 marking(s), 30 transitions(s)
0.000s
```

The result shows that there are no deadlocks in this program. Indeed, the processes call MPI operations in the correct sequence: first, process 0 sends two messages to process 1, then process 1 sends two messages to process 0.

3.4 A more complicated example with a dead end

As mentioned earlier, it is possible to specify an arbitrary recipient when receiving a message. In MPJ Express, the MPI.ANY_SOURCE constant is used for this purpose. In this case, the sender's rank will not be checked when the message is received. Below is a more complex program snippet that uses this constant. This example is taken from the MBI (MBI Bugs Initiative) test suite [23] and adapted to Java.

```
if (rank == 0) {
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, MPI.ANY_SOURCE, 0);
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 3, 0);
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, MPI.ANY_SOURCE, 0);
} else if (rank == 1) {
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 0, 0);
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 3, 0);
} else if (rank == 2) {
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, MPI.ANY_SOURCE, 0);
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 0, 0);
} else if (rank == 3) {
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, 1, 0);
  MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, 0, 0);
} else if (rank == 4) {
  MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 2, 0);
}
```

The analyzer will build a Petri net, which is shown in Fig. 30.

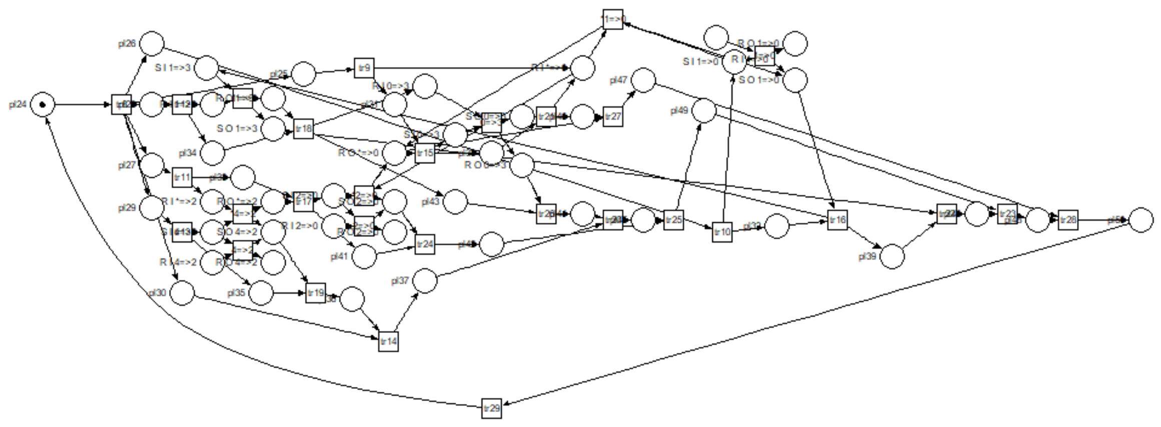

Figure 30 - Petri net for a complex example of a program with a deadlock

The result of running sift will be as follows:

```
deadlock-finder> C:\programs\tina-3.7.0\bin\sift.exe .\target\net.net -
dead
some state violates condition -f:
  {R I 1=>3} {S I 0=>3} {S I 1=>0} pl32 pl34 pl37 pl38 pl42
  firable:
302 markings(s), 833 transitions(s)
0.000s
```

sift finds the following deadlock marking: {R I 1=>3} {S I 0=>3} {S I 1=>0} pl32 pl34 pl37 pl38 pl42". From this, we can determine that a deadlock occurs when:

– process 3 receives a message from process 1 (R I 1=>3);

– process 0 sends a message to process 3 (S I 0=>3);

– process 1 sends a message to process 0 (S I 1=>0).

This deadlock cannot be resolved because in order for process 3 to receive a message from process 1, process 1 must successfully send a message to process 0. But this is impossible because process 0 is blocked from sending a message to process 3.

This situation occurred because the first Recv call in process 0 received a message from process 2. If process 0 had received a message from process 1 at this point, the deadlock would not have occurred. The deadlock in this example will not occur every time the program is run, but it will occur non-deterministically. The reason is that when Recv is called with the MPI.ANY_SOURCE constant, messages from different processes are selected randomly.

3.5 Recommendations for resolving deadlocks

After finding deadlocks with the analyzer, the dependencies of processes should be consider in detail. If necessary, the order of Send and Recv calls should be changed. For example, in Section 3.2, both processes called Recv first and therefore waited endlessly for Send. This situation can be resolved simply by setting process 0 to Send first and then Recv.

If the deadlock occurs when there are receive operations from an arbitrary sender (MPI.ANY_SOURCE) and there is more than one suitable sender, tags can be used to organize the received messages.

In addition, to resolve deadlocks when using blocking operations, non-blocking analogs can be used [24]. For example, in one of the processes, the Irecv operation can be used instead of the Recv operation. In this case, the Irecv operation will not block the process, but will immediately return. Then, after performing the necessary actions, when the received data is needed, blocking Wait operation can be called. It will wait until the data is received.

CONCLUSIONS

As a result of the work, models, methods and an algorithm for analyzing the correctness of distributed programs were developed.

A static code analyzer was implemented that automatically builds the developed model in the form of a Petri net for a given MPI program. To analyze the program, we used both traditional approaches from the field of compiler development (dominators, SSA form, data flow analysis, and others) and new approaches specialized for the current task, such as determining the ranks of processes and the graph of operations.

Despite the limitations imposed on the input programs, the developed models can serve as a foundation for further research.

The applicability of the analyzer was demonstrated on a complex example of a program in which a deadlock occurs non-deterministically. Possible directions for further research are presented below.

At the moment, the analyzer works only with the MPI implementation for Java - MPJ Express. In the future, support for the C language is possible. To do this, one just needs to connect a C parser to the analyzer and convert the syntax tree to HIL. The following phases and intermediate views will not have to be changed. C language support will allow to compare the analyzer with similar solutions and measure its performance for more complex examples.

The analyzer does not process calls to functions defined in user code. However, it is possible to solve this by embedding functions, similar to C++ inline functions. The body of the called function will be embedded at the place of its call, when the parameters are replaced by the arguments that were passed to the call. This conversion will be performed on the HIL form.

As noted, the analyzer output is a Petri net, which is checked for deadlocks by running a simulation in the Tina package. It is possible to make Tina or sift run automatically so that the analyzer calls this program and processes the results of its work. This will make it much easier to use the analyzer.

The next step is to extract information from Tina's results about the marking at which the deadlock occurred. This information will be used to inform the user at what point in their program (line, file) the deadlock is possible.

At the moment, it is possible to correctly process loops only with a condition that can be calculated at compile time. In the future, we need to find a way to process loops with arbitrary conditions. This should allow user to analyze MPI programs of a larger scale and complexity than the examples in this paper.

LIST OF SOURCES OF REFERENCE

1.Javed A., Qamar B., Jameel M., Shafi A., Carpenter B. Towards Scalable Java HPC with Hybrid and Native Communication Devices in MPJ Express. International Journal of Parallel Programming (IJPP) 2015. Springer.

2.Kirk D. B., Hwu W. W. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2016. 576 p.

3.Robey R., Zamora Y. Parallel and High Performance Computing. Manning Publications, 2021. 704 p.

4.MPI: A Message-Passing Interface Standard Version 3.1. Knoxville, Tennessee: Message Passing Interface Forum, June 4, 2015. 836 p.

5.Laguna I., Marshall R., Mohror K., Ruefenacht M., Skjellum A., Sultana N. A Large-Scale Study of MPI Usage in Open-Source HPC Applications. The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC ’19), November 17–22, 2019, Denver, CO, USA. ACM, New York, NY, USA. 12 p.

6.MPJ Express Project. URL : http://www.mpjexpress.org/index.html. (дата звернення 10.11.2022).

7.Droste, A., Kuhn, M., and Ludwig, T. MPI-checker: Static Analysis for MPI. 2nd Workshop on the LLVM Compiler Infrastructure in HPC, 2015. P. 1–10.

8.Yu H., Chen Z., Fu X., Wang J., Su Z., Sun J., Huang C., Dong W. Symbolic Verification of Message Passing Interface Programs. 42nd International Conference on Software Engineering, May 23-29, 2020, Seoul, South Korea. ACM, New York, NY, USA. 13 p.

9.King J.C. Symbolic execution and program testing. Commun. ACM, 19. 1976. P. 385-394.

10.Forejt V., Joshi S., Kroening D., Narayanaswamy G., Sharma S. Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs. ACM Transactions on Programming Languages and Systems. 39, 4, December 2017. 27 p.

11.Zaitsev D.A. Mathematical models of discrete systems: Textbook // Odessa: ONTA, 2004, 40p. In ‘.

12.Котов В.Е. Сети Петри. М. : Наука. Главная редакция физико-математической литературы, 1984. 160 с.

13.Manual Page – sift(n). (загол. з екрану). URL : https:// projects.laas.fr/tina/manuals/sift.html. (дата звернення 10.11.2022).

14.Zaitsev D.A., Shmeleva T.R., Guliak R.N. Analyzing Multidimensional Communication Lattice with Combined Cut-Through and Store-and-Forward Switching Node. Kumar R., Mishra B.K., Pattnaik P.K. (eds) Next Generation of Internet of Things. Lecture Notes in Networks and Systems, vol 201. Singapore : Springer, 2021. P. 705-715

15.MCC'2022 – Results. (загол. з екрану). URL : https://mcc.lip6.fr/ 2022/results.php. (дата звернення 11.11.2022).

16.Fokkink W. Introduction to Process Algebra. Berlin, Heidelberg : Springer, 1999. 168 p. (Texts in Theoretical Computer Science. An EATCS Series)

17.Smolka S. Strategic Directions in Concurrency Research. ACM Computing Surveys. 1996. V. 28. P. 607-625.

18.Aalst W. Three Good Reasons for Using A Petri-Net-Based Workflow Management System. Engineering and Computer Science. 1998. V. 11. P. 161-182.

19.Cooper K., Torczon L. (2011). Engineering a compiler: Second edition. Morgan Kaufmann, 2011. 824 p.

20.Jones N. D., Gomard C. K., Sestoft, P. Partial Evaluation and Automatic Program Generation. Prentice Hall, 1993. 415 p. (Prentice-hall International Series in Computer Science)

21.Rosen B., Wegman M., Zadeck K. Global value numbers and redundant computations. 15th Annual ACM Symposium on Principles of Programming Languages. 1988. P. 12-27.

22.Cytron R., Ferrante J., Rosen B., Wegman M., Zadeck K. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems. October 1991. V. 13. P. 451-490.

23.Laurent M., Saillard E., Quinson M. The MPI BUGS INITIATIVE: a Framework for MPI Verification Tools Evaluation. Correctness 2021: Fifth International Workshop on Software Correctness for HPC Applications, November 2021, St. Louis, United States. P. 1-9.

24.Рольщиков В.Б. Застосування засобів інтерфейсу передачі повідомлень при програмуванні розподілених систем мовою Java: навчальний посібник. Одеса, 2018. 209 с.