Chapter 19

Verification of MPI programs via compilation into Petri nets

Roman N. Guliak[1], Tatiana R. Shmeleva[2] and Dmitry A. Zaitsev[3]
[1]*Odessa State Environmental University, Odessa, Ukraine*, [2]*The Max Planck Institute for Software Systems (MPI-SWS), Saarbrücken-Kaiserslautern, Germany*, [3]*School of Computing, University of Derby, Derby, United Kingdom*

1 Introduction

We recommend model-driven development (MDD) of distributed software based on Petri and Sleptsov nets [1–4] for the purposes of verification directly in the process of software design. As the basic advantage of the approach, we consider applicability of formal methods [5] to analyze properties of interprocess interactions. We focus on Java programs and emphasize on the distributed software verification aspect taking into considerations Message Passing Interface (MPI) constructs only. Conventional software verification can be implemented using well-known tools, for example system KeY [6]. As denoted by the term of namely distributed software verification, we are interested mostly in the liveness property that can be simplified as absence of deadlocks.

In case of Petri net based MDD, a verified Petri net scheme of interprocess communication is subsequently implemented by the corresponding Java program extended by MPI facilities. Though, in the process of detailing an abstract Petri net specification and transforming it into software, some errors can be inserted as well. Besides, many developers use Integrated Developer Environment (IDE) without the corresponding tools that make errors to reveal during the software debugging and testing stage to find out using runtime tools.

In the present chapter, based on more than 2 decades experience of modeling MPI communications by Petri nets [7–10], we develop a technique to sketch the MPI communication scheme in the Petri net form on a given Java-MPI program. The general plot is represented in Fig. 19.1. After sketching a program communication scheme by a Petri net, we analyze Petri net using modeling system Tina [11]; for large scale Petri nets, the clan composition approach is applied [12,13]. Note that after a deadlock has been found, the program should be re-designed and its check is implemented iteratively until we obtain a deadlock-free program.

As the main contribution of the chapter, a technique for modeling MPI programs with Petri nets, that can adequately represent the behavior of distributed systems, has been developed. The developed technique has been implemented with a static code analyzer that creates a Petri net on a given Java program, which uses MPI facilities. The analyzer is based on both a traditional approach of compiler design with the corresponding toolset and some novel features adjusted for the current task.

2 Overview of message passing interface and MPJ express

MPI (Message Passing Interface) [14,15] represents a simple and powerful programming concept for distributed systems. 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 instructions, multiple data). The main content of the MPI standard is represented by procedures or functions, which are traditionally called from the FORTRAN and C programming languages.

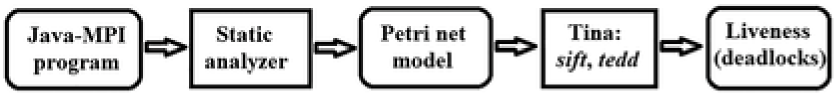

FIGURE 19.1 General scheme of the approach.

A given number of processes is started to run the same code. Initially, the only difference between the processes is an individual number, rank. Based on their ranks, processes execute different parts of the code. Consequently different processes are being executed. Though point-to-point communications of MPI implement peer-to-peer concept of interaction, in majority of applications, a hierarchy is organized with a central master process that plans and distributes subtasks among other processes called workers. For sound scalability, all the workers are implemented in the same way using their ranks to select allotted by the master subtasks.

The first version of the MPI standard 1.0 was released in 1994. The MPI 2.0 standard appeared in 1997 with the following innovations: an extended computing model, dynamic process creation, one-way communication, and parallel I/O. The MPI 3.0 standard was released in 2015; it contains nonblocking versions of collective operations and an extension of one-way communications. And finally, the latest to date is the MPI 4.0 standard released in 2021 that uses a large integer type in many procedures to overcome the limitations of a conventional integer type, split communications, and improvements for error handling.

Among MPI implementations widely used today, we mention Argonne National Laboratory MPICH and Innovative Computing Laboratory's Open MPI. Hardware vendors often have specialized versions of one of these two implementations for their platforms.

In this chapter, we consider MPJ Express [16,17]—an open-source Java messaging library that enables application developers to write and execute parallel applications for multicore processors and computing clusters/clouds.

MPI is successfully used in high-performance computing [18,19]. Applications written in MPI run on cluster computing systems with more than 100 thousand nodes [17]. As the key to MPI success its scalability is considered. We can start all the processes on a single computer, for instance a conventional laptop, for debugging. Then we allocate processes over nodes of a distributed system, a laboratory cluster or a supercomputer.

Though if our goal is a speed-up of computations acknowledged by benchmarks, the network delays of message transmission should be analyzed and minimized, for instance, through using special low latency fast interconnects. Complexity, and the corresponding time delay of solving subtasks on nodes, should justify time delays of the message exchange.

2.1 Basic operations of MPI

An MPI program consists of interacting processes divided into groups called communicators. Each process in the group is associated with an integer rank, enumeration starts from zero. The sender and recipient of a message are identified by the process rank within specified group.

There is a special default communicator *MPI_COMM_WORLD*, which consists of all local processes. In the MPJ Express package, a similar communicator is the static *COMM_WORLD* field in the *mpi.MPI class*.

General-purpose operations are not related to message transfer. One of such 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
```

This method initializes the MPI environment. Programs that use MPJ Express must start by calling *Init* that throws an *MPIException*. This is a common exception thrown by most of the methods in the MPJ Express library.

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

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

This operation clears the state of MPI. If the program terminates normally, not due to an *Abort* call or other unforeseen situation, each process must call *Finalize* before exiting.

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

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

This operation tries to abort all tasks in the communicator group. The error code *errorrcode* will be passed to the run-time.

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
```

Size() returns the number of processes in the communicator group while *Rank()* returns the rank of the process in the communicator group.

Sending and receiving messages by processes is the basic communication mechanism in MPI called *point-to-point communications* represented by operations *Send* and *Recv.* Simple example of how to use these operations follows:

```
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, 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. The first four arguments for *Recv* specify information about the buffer: the buffer pointer, offset, number of items, and type, while the last two arguments specify the sender's rank and tag.

Point-to-point operations can be either blocking when the call does not end until it is safe to use the buffer or nonblocking when the call ends immediately. In this study, we process blocking operations only.

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 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.Status* class contains information about the status of message receipt with the following 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.

Collective MPI operations are operations that involve a group or groups of processes [14]. 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. 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 Ref. [15].

2.2 Compile and run MPJ express program

Let us consider a Java 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 [16]. 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
```

2.2.1 Deadlocks within MPI programs

When using blocking calls for point-to-point operations *Send* and *Recv*, a *deadlock* can occur, 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 [15]. 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*:

```
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.

3 Overview of MPI analyzers

Analyzers that check the correctness of MPI programs can be divided into two groups: static analysis and 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 [20]. 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 [21] can also be classified as a static analyzer. MPI-SV verifies nonblocking MPI operations. It uses symbolic execution and model checking. MPI-SV is based on KLEE, a symbolic execution engine. Symbolic execution [22] 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 Ref. [23], 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.

4 Overview of Petri nets

A Petri net [1] consists of 4 elements: places, transitions, arcs, and tokens (Fig. 19.2). 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 -situated inside places and are represented by bold dots.

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 fires, the tokens are removed from the input places and added to the output places. A transition is fireable when all the input places of the transition contain tokens. An arbitrary, selected in nondeterministic way, fireable transition fires at a step.

Fig. 19.3 shows an example of firing a transition. Before firing, the transition is fireable, but after firing, it is not, because the input places do not contain tokens.

In addition to single arcs, multiple arcs are allowed in Petri nets [24]. In this case, the firing 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. 19.4 shows an example of a transition triggering on a Petri net with multiple arcs.

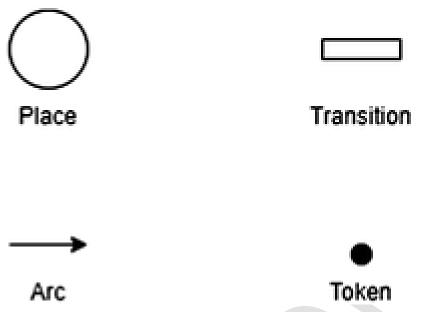

FIGURE 19.2 Elements of Petri net.

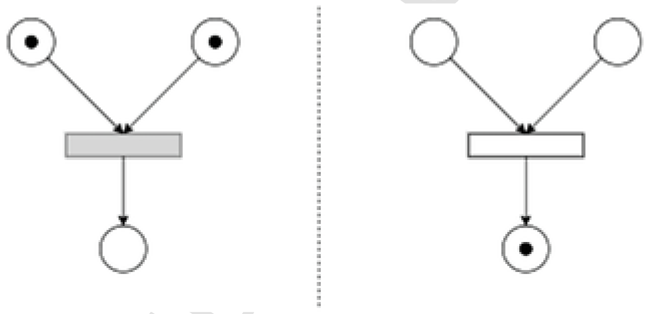

FIGURE 19.3 Firing a transition.

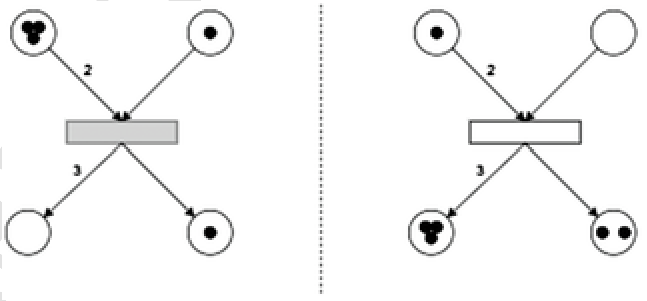

FIGURE 19.4 Firing a transition with multiple arcs.

To introduce basic properties of Petri nets, which are important in this study, we formally define a Petri net according to Ref. [24]. We assume that $\mathbb{N}$ is a set of nonnegative integers. A Petri net is a set $N = \left(P, T, F, M_0\right)$ where:

P is a finite set of places;
T is a finite set of transitions;

$F : P \times T \cup T \times P \rightarrow \mathbb{N}$ is an incident relation function that defines the arcs and their multiplicity; $M_0 : P \rightarrow \mathbb{N}$ is 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 firing sequences and the set of markings reachable 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 its all markings.

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

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

If the places are ordered, then we can associate two integer vectors of length *n*, where $n = |P|$ with each transition *t*:

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

$$t^+ = (d_1, ..., d_n), \text{where } d_i = F(t, p_i).$$

A transition *t* is fireable for some marking *M* of the net *N* if

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

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:

$$M \ge (t)$$

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

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

$$M' = M - t^- + t^+$$

Thus, firing a transition *t* changes the marking 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:

$$\begin{aligned} M &\to M' \\ &\Leftrightarrow \exists t \\ &\in T \\ &: (M \\ &\ge (t)) \wedge (M' \\ &= M - t^- + t^+). \end{aligned}$$

We use the notation $M \to_t M'$ if M' immediately follows *M* as a result of firing transition *t*. A marking M' is reachable from a marking *M* if there exists a sequence of markings $M, M_1, M_2, ..., M'$ and a sequence of transitions $t_1 t_2 ... t_k$ from the set *T* such that

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

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 $R(N) = R(N, M_0)$.

A liveness is the most valuable positive behavioral property of a Petri net. A transition *t* in the Petri net $N = (P, T, F, W, M_0)$ is called potentially live when marking $M \in R(N)$ if

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

i.e., there exists a marking reachable from *M*, namely M' at which the transition *t* is fireable. If $M = M_0$, transition *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.

Time Petri Net Analyzer Tina [11] is a set of tools for editing and analyzing Petri nets. Modeling system Tina was developed at OLC, then VerTICS, a research group of the LAAS/CNRS research laboratory. Tina contains a graphical editor for Petri nets. In MS 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. 19.5.

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. 19.6.

Fireable 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 deadlock 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* that builds various state space abstractions for Petri nets [11]. It accepts as input descriptions in text form (*.net*, *.pnml*, *.tpn* formats) or graphical form (*.ndr* files created by *nd* or *.pnml* with graphics). *sift* allows us 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 simple *Net.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)
```

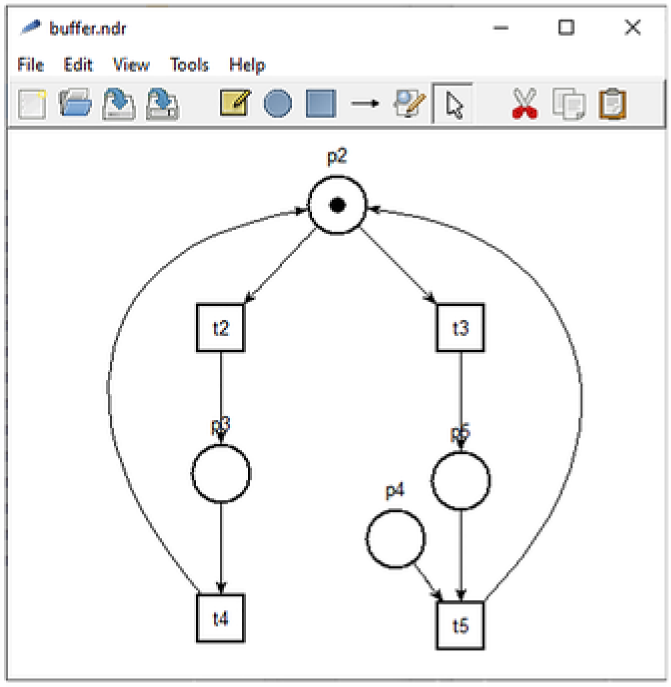

FIGURE 19.5 Tina Petri net editor *nd*.

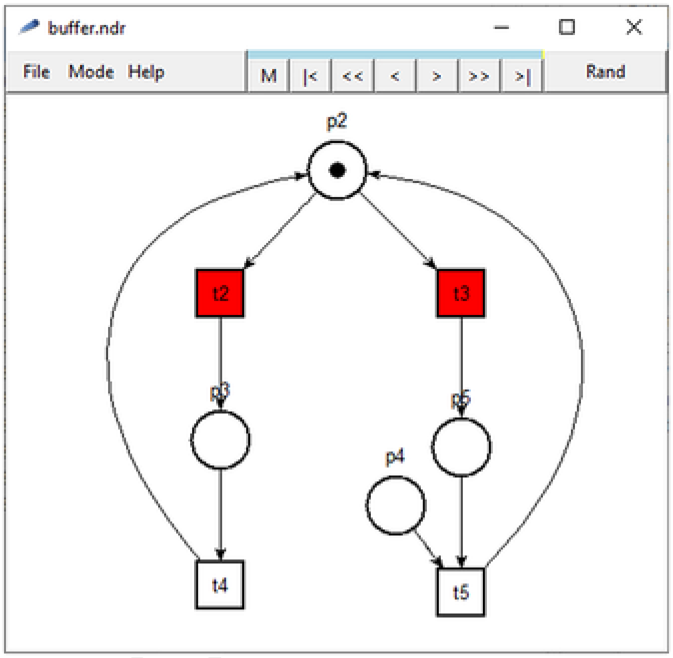

FIGURE 19.6 Petri net simulation mode.

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

5 Representation of MPI message exchange by Petri net

We develop a technique for building a model in the form of a Petri net on a given MPI program with the aim of its further check for deadlocks. Currently, the technique has the following limitations:

– a Java 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 of 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.

In Listing 1, is an example of a program where a deadlock occurs.

On this example (Listing 1), the Petri net shown in Fig. 19.7 is composed. 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.

```
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);
}
```

LISTING 1 A Java-MPI program with a deadlock.

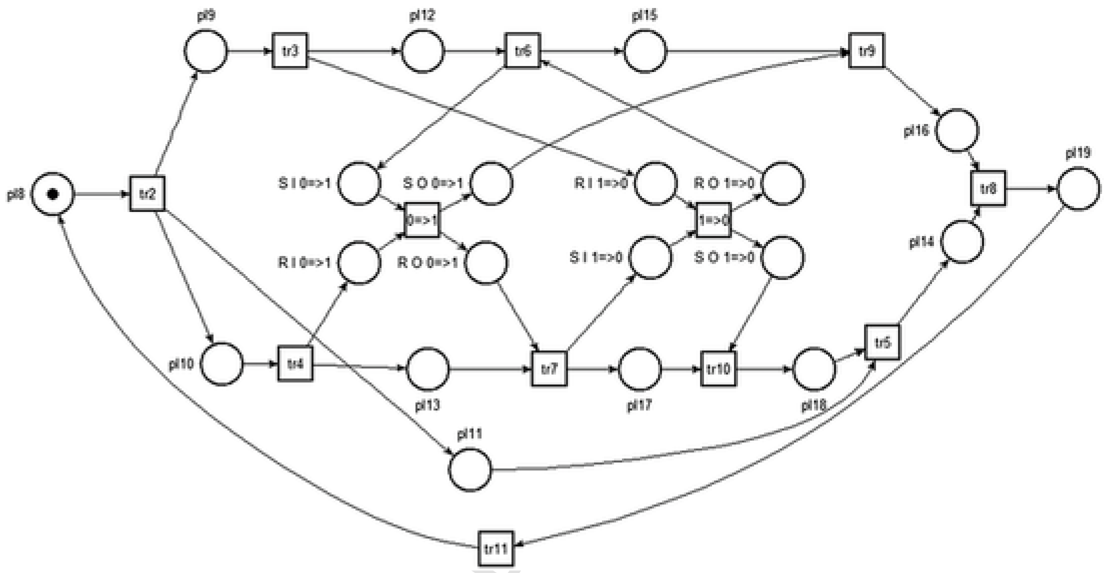

FIGURE 19.7 Petri net scheme of program in Listing 1.

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. 19.7, this place is *pl8*. In Fig. 19.8, 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.

The marking shown in Fig. 19.8 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*".

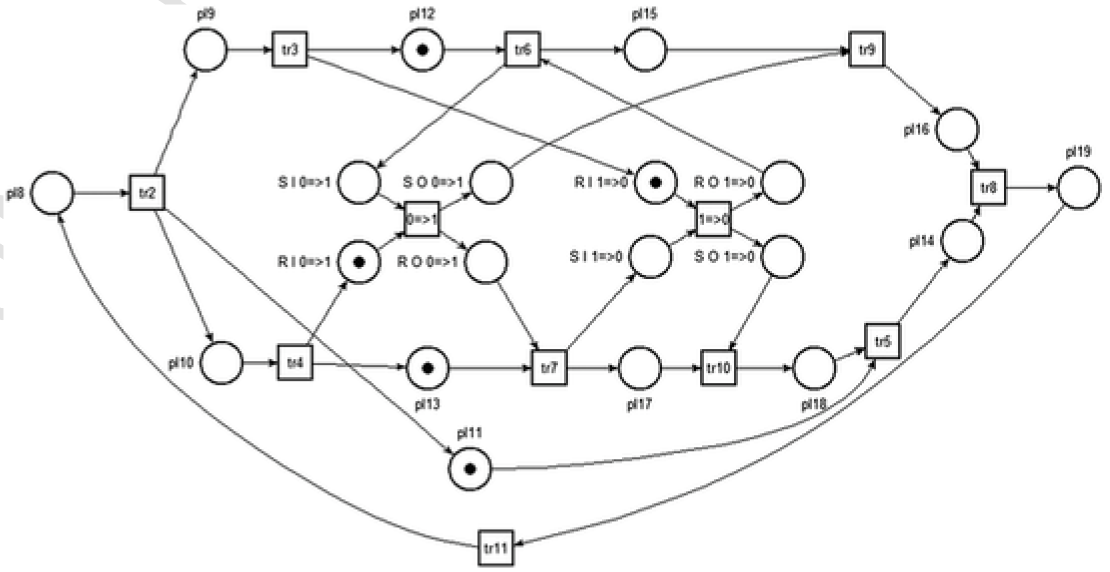

FIGURE 19.8 Deadlock marking of the Petri net in Fig. 19.7.

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 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.

Fig. 19.9 shows a Petri net pattern of a gateway for sending (and receiving) a message from process 0 to process 1. Actually, a gateway should be created for each given triple *(source_process, destination_process, message_tag)*; here for brevity we omit tags supposing they all equals to zero.

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 fires, 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.

In Listing 2, a code snippet in which process 0 sends two messages to process 1 sequentially is represented.

Fig. 19.9 shows a fragment of the corresponding Petri net. In Fig. 19.10 on the right, the gateway described above is shown. Places *pl1*, *pl2*, *pl3*, *pl4* and transitions *tr1*, *tr2*, *tr3* on the left in Fig. 19.10 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.

Thus, a certain gateway is designed to combine all pairs of send-recv functions having the same values of *source_process* and *destination_process* (in general case also the same *message_tag*). As a consequence, the number of incoming arcs of places "*S I X=>Y*" and the corresponding -outgoing arcs of places "*S -O X=>Y*" coincide. The same concerns the pair of places to process the message receiving calls "*R I X=>Y*" and "*R -O X=>Y*".

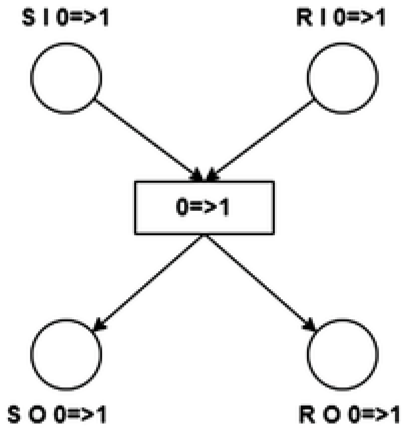

FIGURE 19.9 Gateway for sending a message from process 0 to process 1.

```
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);
}
```

LISTING 2 A code snippet for sending two sequential messages.

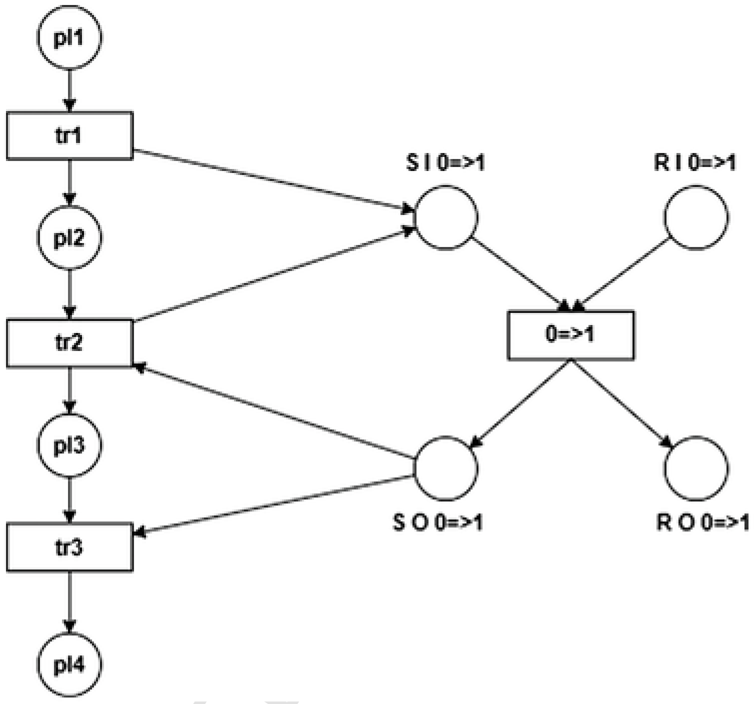

FIGURE 19.10 A Petri net modeling the sequence of operations represented in Listing 2.

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. 19.11, designed to represent the synchronization points, are also known as AND-split and AND-join [25]. AND-split (Fig. 19.11a) splits the execution thread into two parallel processes. AND-join (Fig. 19.11b) models the connection of two parallel threads.

Note that Fig. 19.11 specifies binary splitting-joining while using a given number of successor places of a transition in Fig. 19.11a, and the corresponding number of predecessor places in Fig. 19.11b, allows us to represent an arbitrary number of parallel processes. The splitting pattern models *MPI.INITIALIZE()* while joining pattern models *MPI.FINALIZE()*.

6 Compiling Java-MPI into Petri net

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.

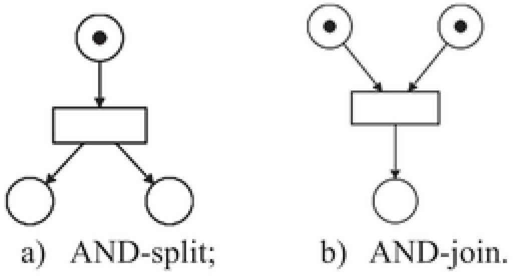

FIGURE 19.11 Splitting and joining parallel processes.

An algorithm for building a Petri net according to a given program follows, the corresponding block diagram is shown in Fig. 19.12.

- 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.

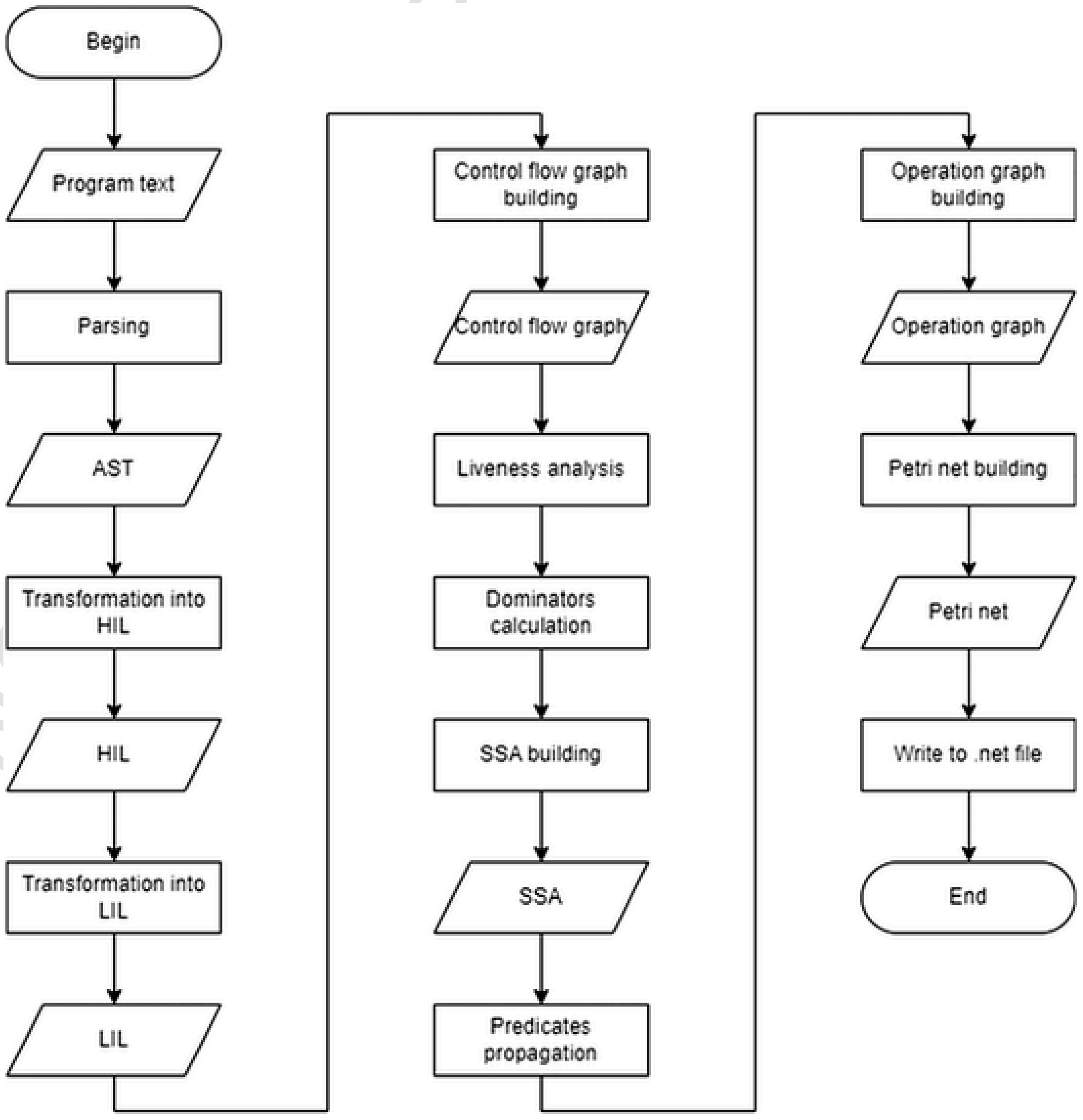

FIGURE 19.12 Flowchart of the Petri net construction algorithm.

– the Petri net is written to a file with the *.net* extension of the Tina format.

Fig. 19.13 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.

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. Detailed descriptions of the flow-chart blocks (Fig. 19.15a) and data structures (Fig. 19.15b) are provided in Ref. [26] that contains an open source prototype implementation of the analyzer. Let us illustrate the considered schemes with examples of obtained data structure.

Fig. 19.14 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);
}
```

The *operation 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 is a rather significant construct developed in this work. The construction of the operation graph is based on the control flow graph.

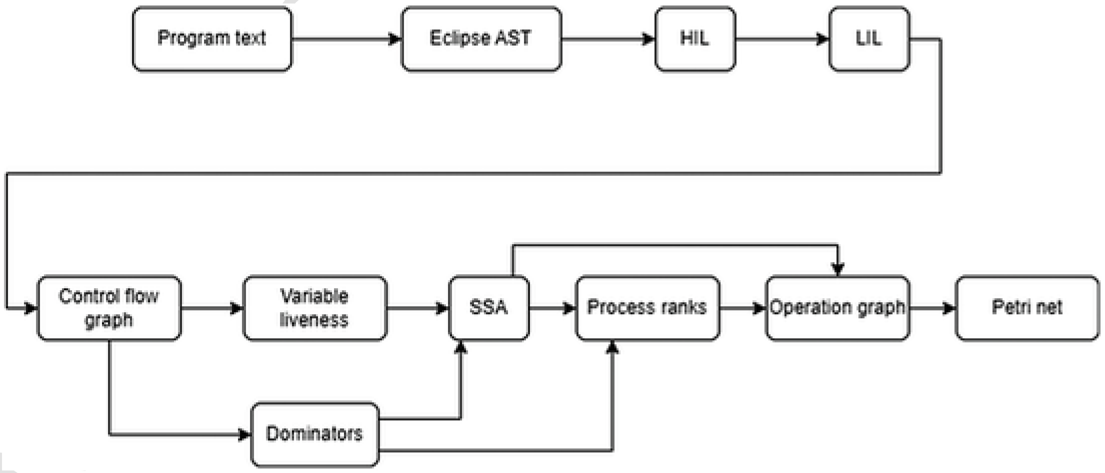

FIGURE 19.13 Dependency of intermediate forms and data structures.

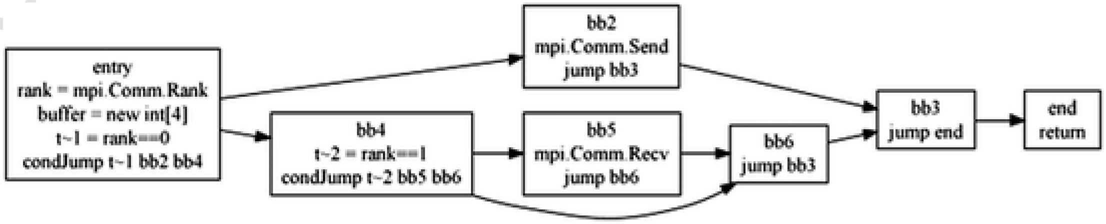

FIGURE 19.14 Control flow graph for a simple program.

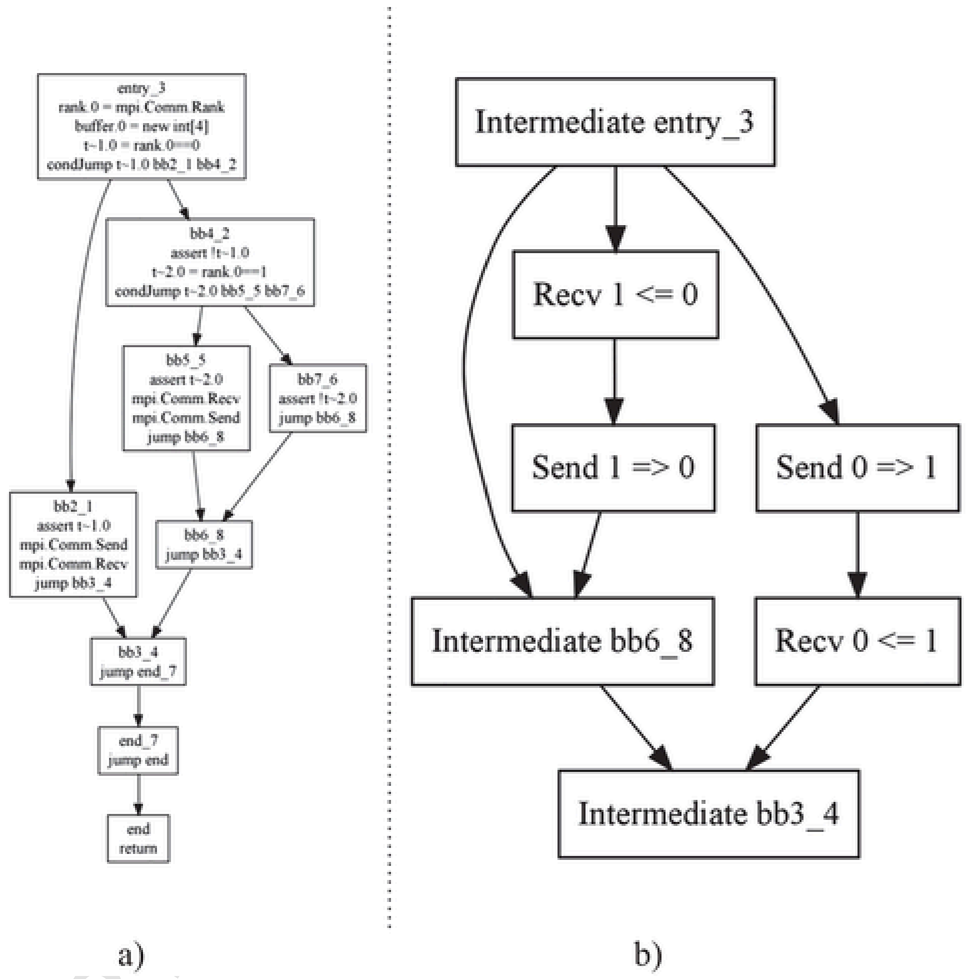

FIGURE 19.15 Control flow graph (a) and corresponding operation graph (b).

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 on 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* are used when different operations can be called by one process under different conditions.

A code snippet for which the operation graph will be built follows:

```
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. 19.15 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.

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.

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. 19.16 shows the operation graph (a) and the corresponding Petri net (b).

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 in detail. However, the structure of the two forms is quite similar.

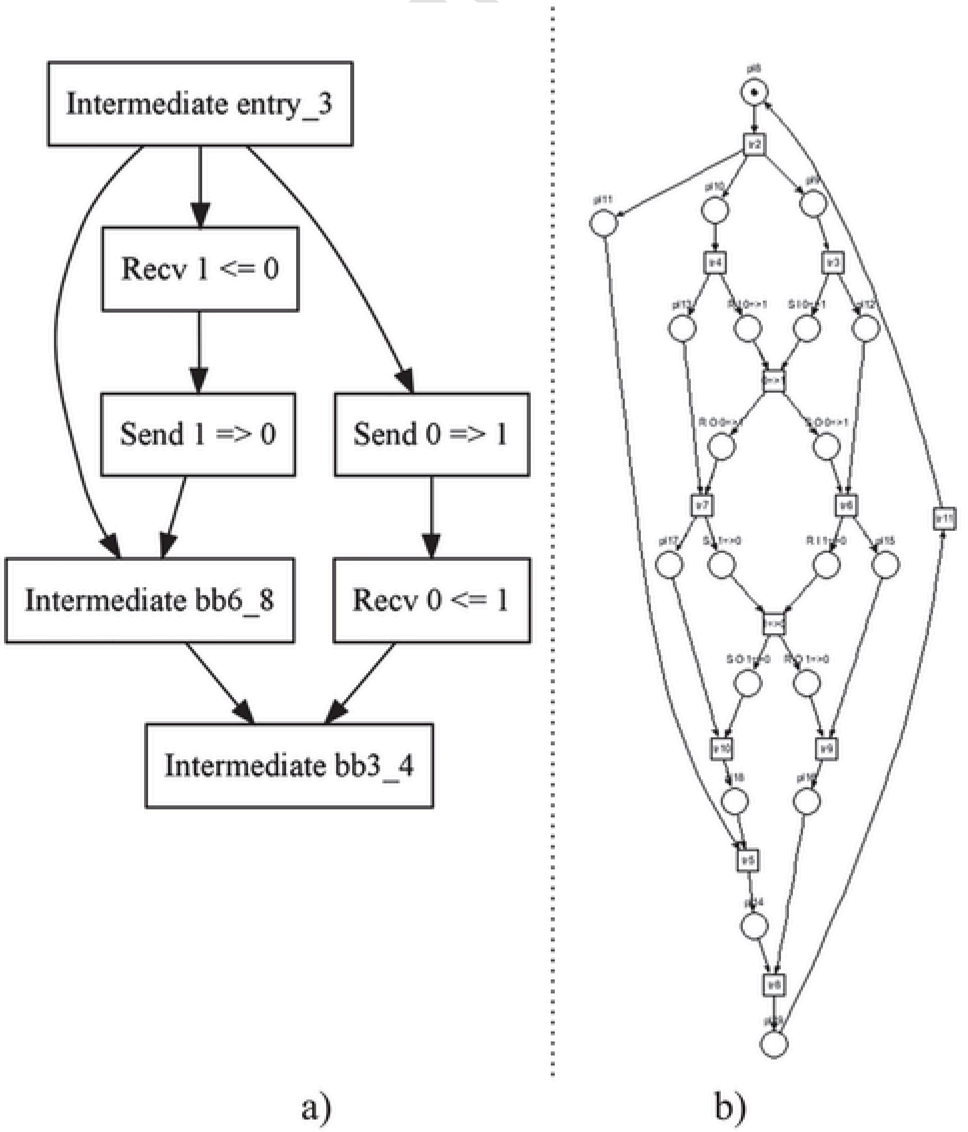

FIGURE 19.16 Operation graph (a) and the Petri net built for it (b).

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.

7 Case study: Analysis of Java-MPI programs via compilation into Petri nets

In this case-study, to illustrate how to analyze real-life Java-MPI code for deadlocks using the developed software and system Tina, we consider a series of examples with increased descriptive complexity.

7.1 A simple example of a program without deadlocks

Here we describe 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. 19.17. In this simple example, the correctness of the program can be seen even visually, because there are no mutual simultaneous dependencies that cause deadlocks.

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 deadlocks 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.

7.2 A simple example of a program with a deadlock

A program snippet with a deadlock follows:

```
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 (Fig. 19.18) in the same way as shown in the previous subsection.

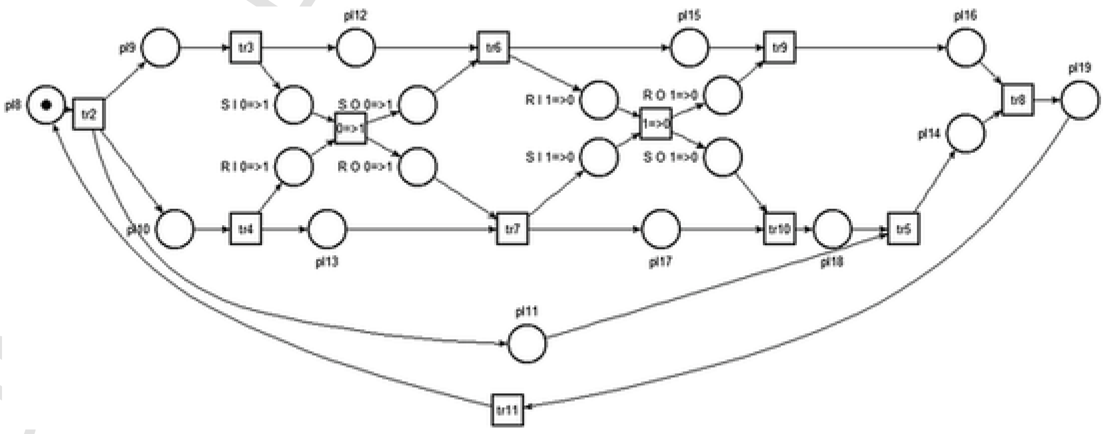

FIGURE 19.17 An example of a Petri net for a correct program.

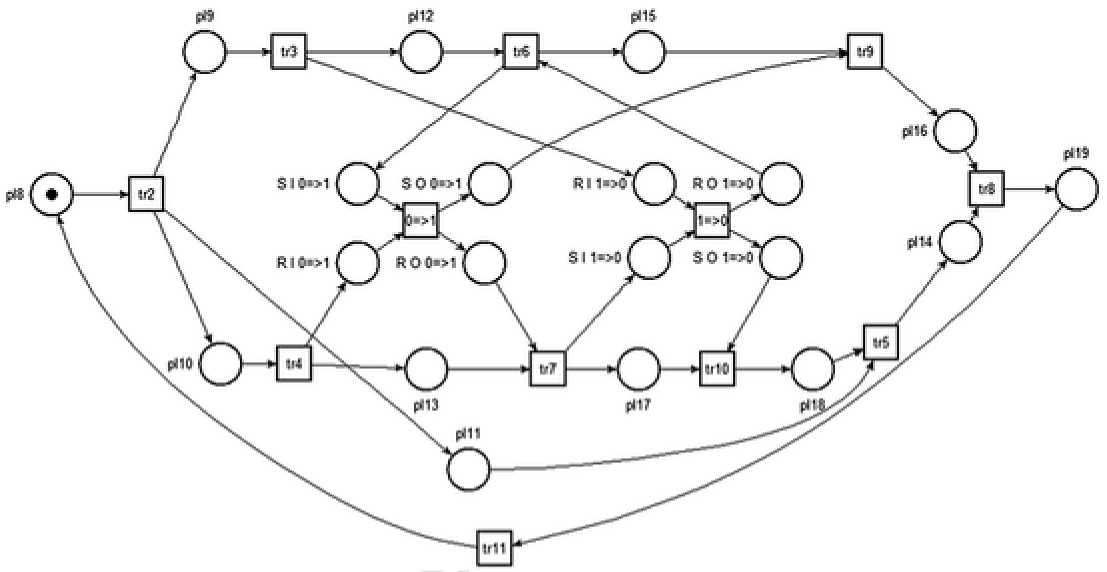

FIGURE 19.18 Petri net for a program with a deadlock.

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 (fireable) 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.

7.3 Example with a simple loop

Here we consider 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). The corresponding program snippet follows:

```
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. 19.19 shows the operation graph.

The Petri net built with the analyzer is shown in Fig. 19.20.

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.

7.4 A more complex example with a deadlock

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

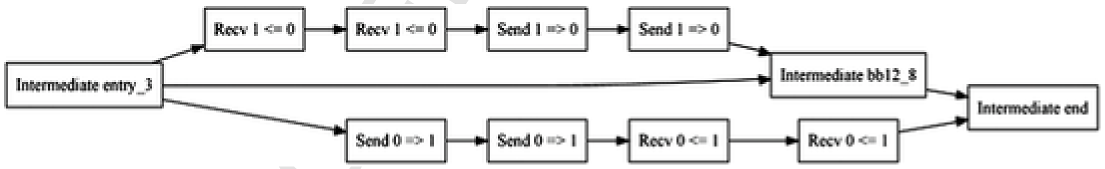

FIGURE 19.19 Operation graph for a program with a loop.

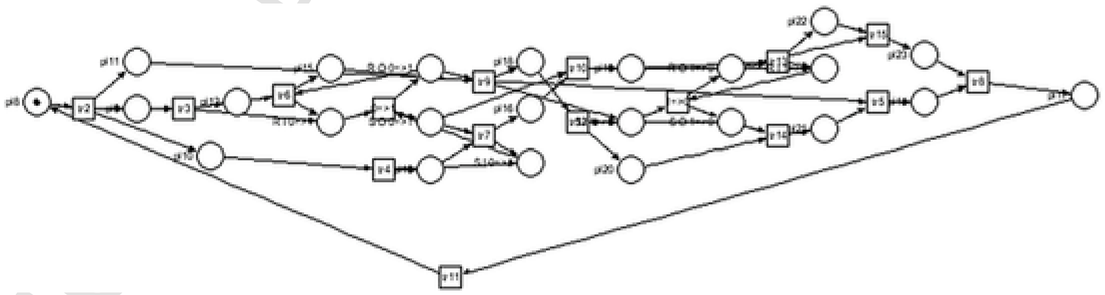

FIGURE 19.20 Petri net for a program with a loop.

(MBI Bugs Initiative) test suite [27] 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 builds a Petri net shown in Fig. 19.21.
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.

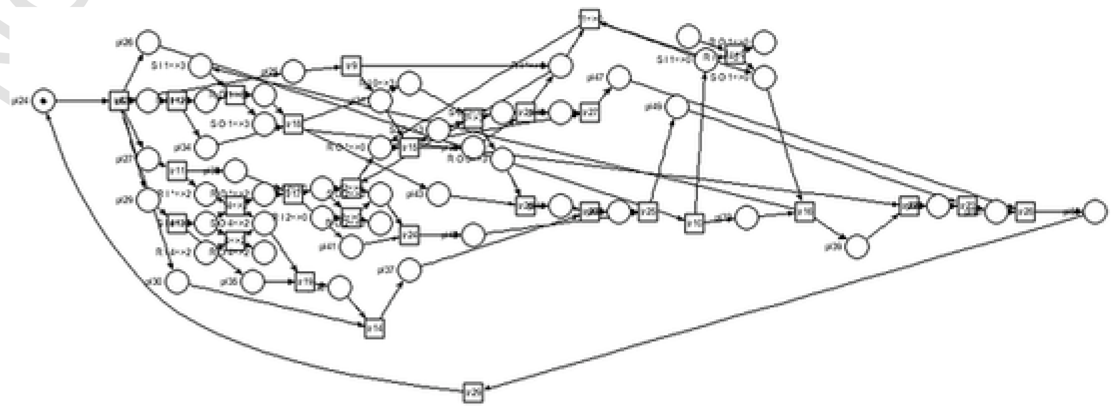

FIGURE 19.21 Petri net for a complex example of a program with a deadlock.

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 nondeterministically. The reason is that when *Recv* is called with the *MPI.ANY_SOURCE* constant, messages from different processes are selected randomly.

7.5 Recommendations for resolving deadlocks

After finding deadlocks using 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 4, 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, nonblocking analogs can be used [14]. 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, the blocking *Wait* operation can be called. It will wait until the data is received.

8 Prospective approaches to efficient analysis of big distributed software systems

There are two basic obstacles on the way for applying the developed technique and software for analyzing real-life big distributed software systems: exponential complexity of deadlock search methods for general Petri nets and loops with MPI communication operations within them. Here we offer prospective solutions for both issues based on using structurally restricted subclasses of Petri nets and a novel theory of infinite Petri nets [28–30], which represent directions for future research.

8.1 Using structurally restricted subclasses of Petri nets

Check for liveness, or even absence of deadlocks, for general Petri net are known as tasks having exponential time complexity. More precisely, it was proven that liveness is polynomially reducible to the reachability problem [31]. Recently, double exponential bound of Lipton [32] for the reachability problem has been expanded to tower complexity [33] that makes the corresponding algorithms impracticable.

Absence of deadlocks check implemented in Tina [11] is based on direct construction of the reachability tree that means applicability for rather modest example nets. The technique has been considerably optimized and sped-up with *sift* and *selt* tools of the Tina toolset based on an ad-hoc *.ktz* format of the reachability graph file. During 2020–22, Tina was a winner of Model Checking Contest [34]. Recently, supplementing the toolset with *tedd* tool that uses a symbolic state space technique, yielded considerable speed-up of computations, for some big benchmark problem instances up to hundred and even thousand times. Though, the speed-up magnitude depends much on the benchmark net structure.

As a prospective direction to gain performance scale suitable for analyses of big distributed systems, we consider study of structurally restricted classes of Petri nets. At first, the majority of nets considered within the chapter represent a marking graph. It is known that liveness of marking graphs is a polynomial time problem. According to Ref. [35], a marking graph is live if its initial marking contains at least one token for each directed circle. Thus the task is reduced to finding basis circles of a directed graph known as having a quadratic time complexity [36].

Let us reconsider studied examples. We obtain alternative processes when using MPI communication block only in case the same triple of the sender, receiver, and the message tag is used in more than one pair of connected *Send-Recv* functions. We would like to note that this case represents rather tangled style of distributed programming and recommend to avoid it.

Taking into consideration results obtained for free-choice nets [37], also leading to an efficient check of liveness though with higher time complexity, we found it rather difficult to transform the model into a free-choice net format, because after synchronization with the *Send-Recv* block namely the calling process should be continued. Process-resource structurally restricted classes of Petri nets [5] also do not correspond to the required communication pattern.

Thus, ensuring MPI programming style leading to Petri net models in the form of a marking graph represents the closest prospective direction for efficient analysis of big distributed systems.

8.2 Avoiding explicit unfolding of loops with infinite Petri nets

MPI communications within loops represent the most significant, from practical point of view, challenge for exhaustive analysis of programs. The current limitation implies considering loops with known at the compilation stage number of repetition for direct implicit unfolding of the loops. That means their straightforward transformation into linear sequences. The approach could be treated as an initial one for experimental use. Let us note that analysis of loops represents the most difficult stage for verification of conventional programs and is based on additional annotations sufficient for proving loop invariants in a formal way. Within a distributed system, we actually have a pair or more complicated system of loops, connected through Send-Recv pairs of functions that represent an even more sophisticated system.

We consider that the most proper technique to model and analyze communications within loops is a recently developed theory of Infinite Petri nets [28]. A finite specification of an infinite Petri net is represented in the form of a parametric multiset rewriting system notation [29,30] that extends format of conventional multiset rewriting systems [38]. In this case, finally we obtain an expression that specifies the model using a set of parameters. The task is to prove the property in question, say liveness, for any valid values of parameters. The corresponding techniques for Petri net invariance proof have been presented for communication lattice structures [29,30]. Actually, we resolve more general task, because we do not take into consideration specific for a given program mutual restrictions on the parameters values. Though, even a general proof can represent rather considerable breakthrough for the problem of communication within loops.

9 Conclusions

We presented a technique for compilation of MPI programs into Petri net models to prove their liveness or, at least, absence of deadlocks. The offered Petri net block to represent *Send-Recv* pair possesses certain advantages compared known works on modeling MPI communications by Petri nets. Timed complexity of the compilation does not exceed complexity of the corresponding parsers or conventional static analyzers, though subsequent analysis of the obtained mode has exponential upper bound of complexity. Modeling system Tina, which we offer to apply for Petri net analysis, contains a recent tool for symbolic state space that speeds-up the process considerably.

Besides, we offer to use restricted patterns of communication when writing programs that yield structurally restricted Petri nets, in fact, marking graphs, which liveness can be proven in a square time depending on the net size.

References

[1] D.A. Zaitsev, Clans of Petri Nets: Verification of Protocols and Performance Evaluation of Networks, LAP LAMBERT Academic Publishing, 2013, p. 292.
[2] D.A. Zaitsev, Paradigm of computations on the Petri nets, Autom. Rem. Control 75 (2014) 1369–1383, doi:10.1134/S0005117914080025.
[3] D.A. Zaitsev, Sleptsov nets run fast, IEEE Transact. Syst. Man Cybernetics Syst. 46 (5) (May 2016) 682–693, doi:10.1109/TSMC.2015.2444414.
[4] D. Zaitsev, Sleptsov net computing resolves problems of modern supercomputing revealed by Jack Dongarra in his turing Award talk in November 2022, Int. J. Parallel, Emergent Distributed Syst. 38 (4) (2023) 275–279, doi:10.1080/17445760.2023.2201002.
[5] Z.W. Li, M.C. Zhou, Deadlock Resolution in Automated Manufacturing Systems: A Novel Petri Net Approach, Springer, 2009.
[6] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P.H. Schmitt, M. Ulbrich (Eds.), Deductive Software Verification – the KeY Book, vol 10001, Springer, LNCS, 2016.
[7] W. Liu, Y. Du, H. Cui, C. Yan, Liveness analysis of parallel program's Petri net models, 2009 International Conference on Information Engineering and Computer Science, Wuhan, China, 2009, pp. 1–4, doi:10.1109/ICIECS.2009.5363769.
[8] P. Zhang, M. Qi, Modeling parallel MPI programs in Petri nets, in: T. Zhang (Ed.), Instrumentation, Measurement, Circuits and Systems, Advances in Intelligent and Soft Computing, vol 127, Springer, Berlin, Heidelberg, 2012.
[9] H. Fan, De Q. Shu, L. Zhang, Petri net automatic modeling method based on system behavior sequence, Applied Mechanics and Materials, Trans Tech Publications, Ltd., October 2014.
[10] T. Krabbe, Modelling MPI Communication Using Colored Petri Nets, Master Thesis, University of Hamburg, Germany, 2023.
[11] B. Berthomieu, O.-P. Ribet, F. Vernadat, The tool Tina – construction of abstract state space for Petri nets and time Petri nets, Int. J. Prod. Res. 42 (4) (2004) 2741–2756. http://www.laas.fr/tina.
[12] D. Zaitsev, S. Tomov, J. Dongarra, Solving linear diophantine systems on parallel architectures, IEEE Trans. Parallel Distr. Syst. 30 (5) (May 1, 2019) 1158–1169, doi:10.1109/TPDS.2018.2873354.
[13] D.A. Zaitsev, T.R. Shmeleva, P. Luszczek, Aggregation of clans to speed-up solving linear systems on parallel architectures, Int. J. Parallel Emergent Distributed Syst. 37 (2) (2022) 198–219, doi:10.1080/17445760.2021.2004412.
[14] MPI: A Message-Passing Interface Standard Version 3.1. Knoxville, Tennessee: Message Passing Interface Forum, June 4, 2015. 836 p.

[15] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, N. Sultana, 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, 2019, p. 12.
[16] MPJ Express Project. URL: http://www.mpjexpress.org/index.html. (дата зϑернення 10.11.2022).
[17] A. Javed, B. Qamar, M. Jameel, A. Shafi, B. Carpenter, Towards scalable java HPC with hybrid and native communication devices in MPJ express, Int. J. Parallel Program. (2015) Springer.
[18] D.B. Kirk, W.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, 2016, p. 576.
[19] R. Robey, Y. Zamora, Parallel and High Performance Computing, Manning Publications, 2021, p. 704.
[20] A. Droste, M. Kuhn, T. Ludwig, MPI-Checker: static analysis for MPI, 2nd Workshop on the LLVM Compiler Infrastructure in HPC, 2015, pp. 1–10.
[21] H. Yu, Z. Chen, X. Fu, J. Wang, Z. Su, J. Sun, C. Huang, W. Dong, 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, 2020, p. 13.
[22] J.C. King, Symbolic execution and program testing, Commun. ACM 19 (1976) 385–394.
[23] V. Forejt, S. Joshi, D. Kroening, G. Narayanaswamy, S. Sharma, Precise predictive analysis for discovering communication deadlocks in MPI programs, ACM Trans. Program Lang. Syst. 39 (4) (December 2017) 27.
[24] D.A. Zaitsev, Mathematical Models of Discrete Systems: Textbook, ONTA, Odessa, 2004, p. 40p.
[25] W. Aalst, Three good reasons for using A Petri-net-based workflow management system, Eng. Comput. Sci. 11 (1998) 161–182.
[26] A tool for finding deadlocks in programs using MPI library for Java (deadlock-finder). https://github.com/hanixer/deadlock-finde r.
[27] M. Laurent, E. Saillard, M. Quinson, The MPI BUGS INITIATIVE: a framework for MPI verification tools evaluation, Correctness 2021: Fifth International Workshop on Software Correctness for HPC Applications, St. Louis, United States, November 2021, pp. 1–9.
[28] D.A. Zaitsev, T.R. Shmeleva, D.E. Probert, Applying infinite Petri nets to the cybersecurity of intelligent networks, grids and clouds, Appl. Sci. 11 (24) (2021) 11870.
[29] D.A. Zaitsev, I.D. Zaitsev, T.R. Shmeleva, Infinite Petri nets: Part 2, modeling triangular, hexagonal, hypercube and hypertorus structures, Complex Syst. 26 (4) (2017) 341–371.
[30] D.A. Zaitsev, I.D. Zaitsev, T.R. Shmeleva, Infinite Petri nets: Part 1, modeling square grid structures, Complex Syst. 26 (2) (2017) 157–195.
[31] J. Petersen, Petri Net Theory and the Modeling of Systems, Prentice-Hall, Upper Saddle River, NJ, 1981.
[32] R.J. Lipton, The reachability problem requires exponential space, Tech. Rep. (1976).
[33] W. Czerwinski, S. Lasota, R. Lazic, et al., The reachability problem for Petri nets is not elementary, TCS Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC2019), vol 2433, Jun 2019.
[34] Model Checking Contest (MCC), LIP6, Paris, France. https://mcc.lip6.fr/2022/results.php.
[35] T. Murata, Petri nets: properties, analysis and applications, Proc. IEEE 77 (4) (April 1989) 541–580.
[36] S. Even, Graph Algorithms, Cambridge University Press, United States, 2011.
[37] K. Barkaoui, M. Minoux, A polynomial-time graph algorithm to decide liveness of some basic classes of bounded Petri nets, LNCS 616 (1992) 62–75.
[38] I. Cervesato, N. Durgin, J. Mitchell, et al., Relating strands and multiset rewriting for security protocol analysis, Proceedings 13th IEEE Computer Security Foundations Workshop (CSFW-13), July 2000, pp. 35–51.

Further reading

[1] D.A. Zaitsev, T.R. Shmeleva, R.N. Guliak, Analyzing multidimensional communication lattice with combined cut-through and store-and-forward switching node, in: R. Kumar, B.K. Mishra, P.K. Pattnaik (Eds.), Next Generation of Internet of Things, Lecture Notes in Networks and Systems, vol 201, Springer, Singapore, 2021, pp. 705–715.
[2] W. Fokkink, Introduction to Process Algebra, Springer, Berlin, Heidelberg, 1999, p. 168 (Texts in Theoretical Computer Science. An EATCS Series).
[3] S. Smolka, Strategic directions in concurrency research, ACM Comput. Surv. 28 (1996) 607–625.
[4] K. Cooper, L. Torczon, Engineering a Compiler, second ed., Morgan Kaufmann, 2011, p. 824.
[5] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993, p. 415 (Prentice-Hall International Series in Computer Science).
[6] B. Rosen, M. Wegman, K. Zadeck, Global value numbers and redundant computations, 15th Annual ACM Symposium on Principles of Programming Languages, 1988, pp. 12–27.
[7] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Trans. Program Lang. Syst. 13 (October 1991) 451–490.