SCIENCE CHINA
Information Sciences

• REVIEW •
Special Focus on Software Automation

October 2019, Vol. 62 200101:1–200101:24
https://doi.org/10.1007/s11432-018-9825-3

Evaluation of model checkers by verifying message passing programs

Weijiang HONG[1,2], Zhenbang CHEN[1*], Hengbiao YU[1,2] & Ji WANG[1,2*]

[1]*College of Computer, National University of Defense Technology, Changsha* 410073*, China;*
[2]*State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha* 410073*, China*

Received 30 July 2018/Revised 24 October 2018/Accepted 6 March 2019/Published online 3 September 2019

Abstract Benchmarks and evaluation are important for the development of techniques and tools. Studies regarding evaluation of model checkers by large-scale benchmarks are few. The lack of such studies is mainly because of the language difference of existing model checkers and the requirement of intensive labor in building models. In this study, we present a large-scale benchmark for evaluating model checkers whose inputs are concurrent models. The benchmark consists of 2318 models that are generated automatically from real-world message passing interface (MPI) programs. The complexities of the models have been inspected to be well distributed and suitable for evaluating model checkers. Based on the benchmark, we have evaluated five state-of-the-art model checkers, i.e., PAT, FDR, Spin, PRISM, and NuSMV, by verifying the deadlock freedom property. The evaluation results demonstrate the ability and performance difference of these model checkers in verifying message passing programs.

Keywords model checker, evaluation, benchmark, MPI, symbolic execution

Citation Hong W J, Chen Z B, Yu H B, et al. Evaluation of model checkers by verifying message passing programs. Sci China Inf Sci, 2019, 62(10): 200101, https://doi.org/10.1007/s11432-018-9825-3

1 Introduction

Model checking [1] is an effective push-button technique used for verifying concurrent systems. Until now, many model checkers have been developed and successfully applied to verify hardware or software systems. A model checker usually accepts a model $\mathcal{M}$ of the system under verification and a critical property φ of the system. Then, it tries to explore all the status of $\mathcal{M}$. If no violation of φ is detected during exploration, the model checker reports that $\mathcal{M}$ satisfies φ; otherwise, a counter-example is reported by the model checker which can be used to detect and fix bugs. In this way, model checking provides an automatic verification method. However, model checking suffers from state explosion problem, especially when applied to verify concurrent systems.

Although there are several successful studies on using model checking to verify concurrent systems, only a few of them exists for evaluating and comparing model checkers [2]. This is because different model checkers support different modeling languages. In the evaluation, much engineering effort is needed to generate the models with respect to the input languages of different model checkers. However, the existing benchmarks of model checking mainly use manually created models [3], hardware benchmark model[1)], or models of classical problems[2)]. In addition, the benchmarks for software model checkers usually consist

* Corresponding author (email: zbchen@nudt.edu.cn, wj@nudt.edu.cn)
1) Hardware Model Checking Contest Website. Http://fmv.jku.at/hwmcc17/.
2) Model Checking Contest Website. Https://mcc.lip6.fr/.

of programs instead of models3). To the best of our knowledge, large benchmarks consisting of concurrent models automatically extracted from real-world concurrent programs do not exist. Therefore, study on evaluating model checkers over a large-scale benchmark of such models is lacking.

In this study, we provide a large-scale model benchmark for evaluating existing model checkers. The models in the benchmark are automatically extracted from real-world message passing programs. Message passing is the current de-factor programming paradigm in high-performance computing (HPC). Message passing interface (MPI)4) plays a crucial role in developing HPC applications. However, due to the complexities such as non-deterministic and non-blocking communications, the development and maintenance of MPI programs are not trivial. In particular, the verification of MPI programs is extremely challenging [4] and the concurrency in MPI programs is used for evaluating the model checker. However, in some of the existing studies (e.g., [5, 6]), they either manually create the models or are difficult to support real-world MPI programs. Hence, it is not suitable for creating a large-scale model benchmark by leveraging them.

In our previous study [7], we developed a symbolic verification method, called MPI-SV, for MPI programs. MPI-SV combines symbolic execution [8] and model checking to verify MPI programs. Symbolic execution extracts path-level models from MPI programs, whereas model checking verifies path-level models with respect to critical properties, such as deadlock freedom. The two combined techniques complement each other. Symbolic execution helps to improve code coverage and handle the complex language constructs in the MPI code and extracts the communication models for model checking. By contrast, model checking helps to boost the efficiency of symbolic execution and enlarge the scope of verifiable properties. In principle, we can use the tool of MPI-SV as a benchmark generator to generate path-level models of real-world MPI programs.

Based on MPI-SV, we created a benchmark from 10 real-world MPI programs and the benchmark consists of 2318 models. Using the benchmark, we evaluated five state-of-the-art model checkers, i.e., PAT5), FDR [9], Spin6), PRISM7), and NuSMV8). The evaluation is enabled by generating the input models for different model checkers from the models in the benchmark. The main contributions of this study are as follows:

- A large-scale benchmark for evaluating model checkers is generated. It consists of 2318 models extracted from real-world MPI programs and is justified to be well distributed in complexity.
- A comprehensive evaluation of five state-of-the-art model checkers based on the benchmark.
- A discussion of lessons learned from creating the benchmark and the evaluation.

Structure. The rest of this study is organized as follows. A brief summary of the background of the study, which includes MPI, MPI-SV, and the model checkers, is presented in Section 2. Section 3 presents the design and generation as well as the the evaluation of the benchmark. Then, the translation algorithms from the models in the benchmark to the input models of model checkers are presented in Section 4. The evaluation of five model checkers and its results are presented in Section 5. Furthermore, Section 6 discusses related work. Finally, the conclusion of the study with some further research directions is presented in Section 7.

2 Preliminaries and framework

This section first provides an introduction to our framework for generating benchmark and evaluating model checkers. Then, the key MPI operations, the evaluated model checkers, and MPI-SV will be briefly introduced.

3) SV-Comp Website. Https://sv-comp.sosy-lab.org/.
4) Message Passing Interface Forum. Http://www.mpi-forum.org/docs/.
5) PAT Website. Http://pat.comp.nus.edu.sg.
6) Spin Website. Http://www.spinroot.com.
7) PRISM Website. Http://www.prismmodelchecker.org.
8) NuSMV Website. Http://nusmv.fbk.eu.

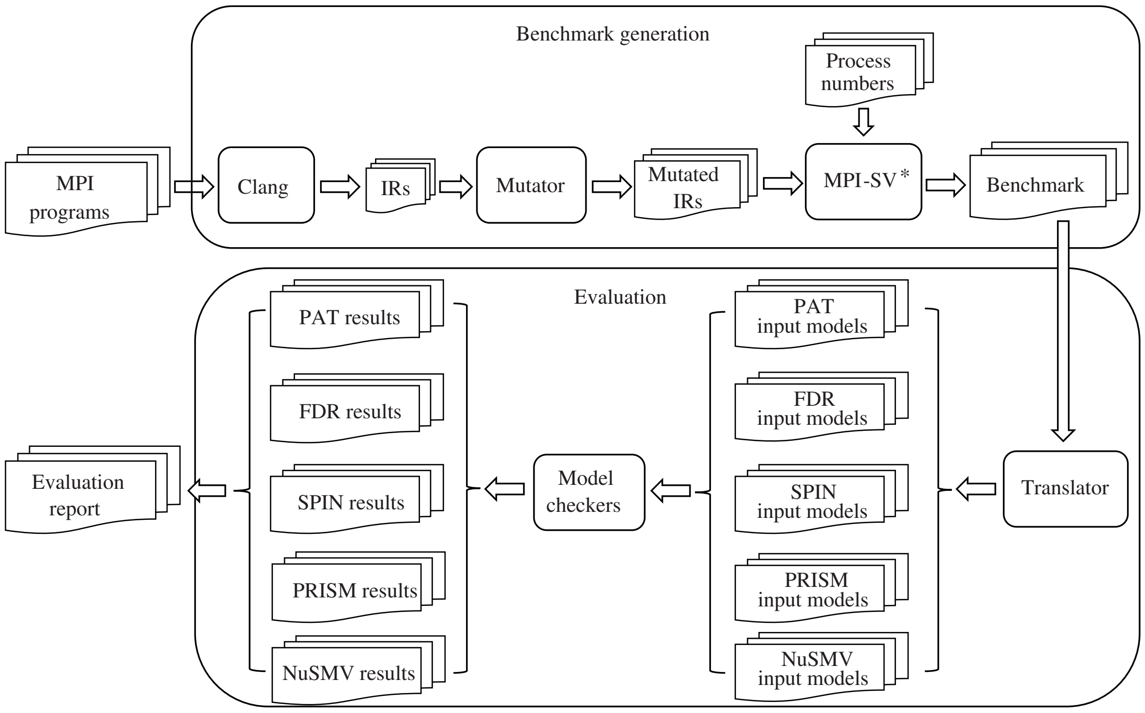

Figure 1 Framework of benchmark and evaluation.

2.1 Framework

Figure 1 shows the basic framework of our study. It involves a two-stage procedure: benchmark generation and evaluation. At the first stage, the inputs are MPI C programs, which will be compiled into an intermediate representation (IR) by Lattner [10]. Then, the IR will be mutated, and the numerous mutated IRs will be fed into a tool called MPI-SV* to generate our benchmark. MPI-SV* is a variant of MPI-SV (c.f. Subsection 2.4) for benchmark generation. Our benchmark contains many models in an unified syntax (c.f. Subsection 3.2). At the second stage, we translate the models in the benchmark to the input models for each evaluated model checker. Then, each model checker is invoked to verify the models. The verification results are analyzed for a comprehensive evaluation of the model checkers. Thus, the framework is automated and the translator component and the automation scripts are implemented. In the subsections, we will briefly introduce some key notations and components of our framework.

2.2 Message passing interface

An orthodox MPI program is written in a programming language, such as C and Fortran. The program consists of not only computation statements but also MPI operations that call for communication. The execution of the MPI program involves a fixed number of processes running on single or multiple machines. These processes communicate by message passing, performed by calling MPI operations. The MPI operations can be divided into two types: blocking and non-blocking. The key blocking MPI operations include the following:

- `Ssend(i, tag)`: sends a message with a tag to the ith process, and the sending process blocks until the message is received by the destination process.
- `Recv(i, tag)`: receives a message with the tag from the ith process, and the receiving process blocks until the message from the ith process is received.
- `Recv(*, tag)`: receives a message with the tag from any process, and the receiving process blocks until a message is received regardless of which process sends the message. A receive operation of this type is called wildcard receive, which also causes non-determinism.

Table 1 An example of an MPI program.

P_0	P_1	P_2
`IRecv(*, 1)`	`ISend(0, 1)`	`ISend(0, 1)`
`Recv(1, 1)`	`Barrier`	`Barrier`
`Barrier`		

• `Barrier`: blocks the process until all the processes have called `Barrier`, which makes a global synchronization.

• `Wait(req)`: blocks until the operation indicated by req is completed.

Non-blocking operations are frequently used in MPI programs for improving the performance. The key non-blocking operations are the following:

• `ISend(i, tag, req)`: sends a message with a tag to the ith process, and the sending process returns immediately after the operation is completed. The parameter req is used to indicate the status of the operation.

• `IRecv(i, tag, req)`: receives a message with the tag from the ith process, and the receiving process returns immediately after being completed. Similarly, `IRecv(*, tag, req)` is the non-blocking wildcard receive.

Some complex MPI operations, such as MPI_Bcast and MPI_Gather, can be implemented by composing these key operations, and MPI-SV supports a wide range of complex MPI operations.

Table 1 shows an example of an MPI program. In the example, the program runs three processes, i.e., P_0, P_1, and P_2. Both P_1 and P_2 first send a message with tag 1 to P_0. Then, they wait for the remaining processes to synchronize. P_0 first receives a message with tag 1 from any process in a non-blocking way. Afterward, P_0 blocks until a message with tag 1 is received from P_1. This program clearly has a deadlock when P_0's wildcard receive operation receives the message from P_1 first, which causes the second blocking receive operation to be blocked forever since there would be no message from P_1 anymore. If the wildcard receive operation receives the message from P_2, no deadlock will happen.

2.3 Model checkers

Model checking is one of the most effective automated techniques for verifying the correctness of software and hardware designs. It explores all possible states in a brute-force manner to prove whether a given system model truly satisfies a property. Several effective model checkers have been developed. Here, we give a short description of the five model checkers we evaluated in this study. The reason that we chose these five model checkers is that they are available, applies state-of-the-art model techniques, and are still under development or maintenance. In addition, we require the model checker to provide a command-line interface; otherwise, we do not include it, such as UPPAAL[9].

• **PAT:** Process analysis toolkit (PAT) is a self-contained framework designed to apply state-of-the-art model checking techniques for system analysis. The system model is specified by the classic process algebra language communicating sequential process (CSP) [11]. An example of using PAT is as follows.

```
(* channel_definition *)
channel C1 1;
channel C2 1;

(* process_definitions *)
P0 = C1!1->Skip; C2!->Skip;
P1 = (C1?1->Skip [] C2?1->Skip);

(* parallel operation *)
P = P0 || P1;
```

9) UPPAAL Website. Http://www.uppaal.org.

```

(* assertion *)
#assert P deadlockfree;
```

As shown in this example, we declare two different channels and processes. Then, we use the parallel composition operation defined in CSP to obtain process `P` by composing processes `P0` and `P1` in parallel. Moreover, we can write some assertions (or properties) to be verified by the model checker. In the example, we verify the deadlock freedom using the following model checkers.

- **Spin:** Spin is a model checker for verifying concurrent systems, e.g., data communication protocols. The input language of Spin is process meta language (Promela), which provides a convenient way for modeling concurrent systems. The following is an example of Spin.

```
(* variable_definitions *)
#define a1
#define b2
chan ch = [1] of {byte};

(* process_definitions *)
Proctype A()
{   ch!a
}
Proctype B()
{   ch!b
}
Proctype C()
{   if
    :: ch?a
    :: ch?b
    fi
}

(* parallel operation *)
init
{  atomic { run A(); run B(); run C() }
}
```

This example defines three processes run in parallel and one channel. Processes `A` and `B` write message `a` and `b`, respectively, into channel `ch`. The first branch in `C` is executable if and only if the channel contains message `a`. The second branch is defined in a similar manner. Considering that the channel's capacity is 1, the message that will be available is non-deterministic, which depends on the writing speeds of the processes.

- **FDR:** FDR [9] is also a model checker used for analyzing programs written in CSP, in particular, machine-readable CSP, named CSP$_M$ [12], which combines the operators of CSP with a functional programming language. The syntax of the input language of FDR is similar to that of PAT. An example is given as follows:

```
(* variable_definitions *)
channel C1
channel C2
channel D

(* process_definitions *)
P0 = C1 ->D -> SKIP
P1 = D ->C2 -> SKIP
```

```

(* parallel operation *)
Q = P0 [|{D}|] P1

(* assertion *)
assert Q : [deadlock free[F]]
```

This example illustrates the synchronization between processes. Here, the channels, i.e., C1, C2, and D, can be understood as events. P0 and P1 execute these channels in order. Q forces P0 and P1 to synchronize on event D while any other event can be performed by either process. The property Q, i.e., deadlock freedom, is written in the assert statement.

- **PRISM:** PRISM is a probabilistic model checker that supports the modeling and verification of probabilistic models, including discrete-time Markov chains, continuous-time Markov chains, Markov decision processes, and probabilistic timed automata. The following shows an example in Prism.

```
(* the type of model *)
mdp

module M
    (* variable_definitions *)
    x : [0..2] init 0;

    (* transition *)
    [] x=0 -> 0.8: (x' =0) + 0.2: (x' =1);
    [] x=1 -> (x' =2);
    [] x=2 -> 0.5: (x' =2) + 0.5: (x' =0);
end module
```

This example describes a Markov decision process. Three states are represented by x. Process M can be in one of the three states, i.e., 0, 1, 2. From state 0, the process can move to state 1 with a probability of 0.2 and remain in the same state with a probability of 0.8. From state 1, the process can only move to state 2. Finally, from state 2, the process will either remain there or move back to state 0 with an equal probability.

- **NuSMV:** NuSMV is a symbolic model checker that re-implements and extends the original binary decision diagram-based model checker SMV [13]. NuSMV's input model is specified by labeled transition system [14]. The following is an example.

```
MODULE main
VAR
  request : boolean;
  machine : {ready, busy};
ASSIGN
  init(machine) := ready;
  next(machine) :=
    case
    machine = ready & request = TRUE  :  busy;
    TRUE                              : {ready, busy};
    esac;
```

This example specifies the transition of the machine state between ready and busy. While the current state of the machine is ready and the request is TRUE, its next state is busy; otherwise, the next state is randomly chosen from {ready, busy}.

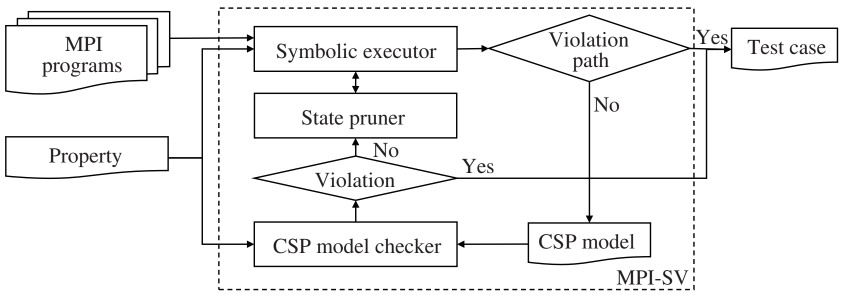

Figure 2 Framework of MPI-SV [7].

2.4 MPI-SV

Figure 2 shows the framework of MPI-SV. The inputs of MPI-SV are an MPI C program, its number of running processes, and a property (e.g., deadlock freedom) to verify.

The program will be analyzed using symbolic execution [8]. During symbolic execution, different cases of message matching in MPI communications will be explored systematically. The property will be checked simultaneously to detect property violations. If a path explored by the symbolic executor is terminated and no violation is detected, a CSP model is generated from the communication behavior of the path. The CSP model represents the equivalent communication behavior of the current path soundly and completely by changing only the message matchings of the wildcard receives in the path. Then, the CSP model will be fed into a CSP model checker for verification. In [7], PAT was used as the model checker. If PAT reports a counter-example, a violation of the property is detected and reported; otherwise, the equivalent paths of the current path are pruned because no violation exists in the equivalent behaviors.

In this way, MPI-SV combines symbolic execution and model checking to verify MPI programs. In principle, MPI-SV divides the behaviors of an MPI program into different equivalent classes. Each class is modeled by a CSP model and only one path inside the class is explored by a symbolic executor. In the high level, symbolic execution can be considered an extractor for extracting path-level models and model checking is used to verify that the models satisfy the property. Hence, we can use MPI-SV as a benchmark generator to generate the models of behavior equivalence classes of an MPI program under various running processes.

We only need to modify MPI-SV slightly to generate benchmark models during verification, i.e., generating a benchmark model in the unified syntax (c.f. Subsection 3.2) when generating a CSP model. We use MPI-SV* (c.f. Figure 1) to denote the modified version of MPI-SV for benchmark generation.

3 Benchmark

3.1 Benchmark design

Our criteria of selecting MPI programs for benchmark generation are as follows: (1) the programs must be real-world MPI programs; (2) the types and scales of the programs are diverse; and (3) the programs are analyzable to MPI-SV. Finally, we select the MPI programs in Table 2 for producing benchmark models.

All the programs, which are real-world open source MPI programs from real-world applications, are classified into several categories. In numeric calculation, we have `Integrate_mw` and `Diffusion2d` from the FEVS benchmark [15]. In addition, an MPI implementation for Gaussian elimination `Gauss_elim` [16] and a parallel solver for heat equation `Heat` [17] are typical applications in numeric calculation. For transition and communication behavior, we have `DTG` from a Ph.D. dissertation [18] and `Pingpong` that is a testing program for communication performance. In addition, we collect the MPI programs related to image processing, i.e., `Mandelbrot` and `Image_manip`, which draw the Mandelbrot set for a bitmap in

Table 2 Programs for benchmark generation

Program	Line of code	Brief descripttion
`DTG`	90	Dependence transition group
`Integrate_mw`	181	Integral computing
`Diffusion2d`	197	Simulation of diffusion equation
`Gauss_elim`	341	Gaussian elimination
`Heat`	613	Heat equation solver
`Pingpong`	220	Comm performance testing
`Mandelbrot`	268	Mandelbrot set drawing
`Image_manip`	360	Image manipulation
`Kfray`	12728	KF-Ray parallel raytracer
`ClustalW`	23265	Multiple sequence alignment
Total	38263	10 open source MPI programs

parallel and deals with image manipulations, respectively. All `Pingpong`, `Mandelbrot`, and `Image_manip` are downloaded from GitHub. There are also two large MPI programs: `Kfray`[10] is a ray tracing program that can create realistic images and `ClustalW` [19] is a popular tool for aligning multiple gene sequences.

Besides the programs, the number of processes to run in a program is important for model generation. In principle, the number of processes determines the scale of parallelism. Our aim is to have models under different scales of parallelism. Hence, we plan to use MPI-SV to analyze each program under different numbers of processes, i.e., 2, 4, 6, 8, and 10, except the programs developed with a fixed number of processes, such as `DTG` and `Pingpong`. In addition, due to the huge path space under more processes, we only analyze `Diffusion` under 4 and 6 processes.

3.2 Benchmark generation

As shown in Figure 1, the selected programs will be compiled into LLVM intermediate representation (IR) [20] for symbolic execution. Then, the IR will be fed to MPI-SV for model generation. The property we want to check is deadlock freedom, which is critical for MPI programs.

Note that we mutate the IR of each program to diversify the benchmarks [21]. Mutants are generated by rewriting a randomly selected receive operation using the following two rules:

- Replace `Recv(i)` with if $(x > a)$ `Recv(i)` else `Recv(*)`.
- Replace `Recv(*)` with if $(x > a)$ `Recv(*)` else `Recv(j)`.

Here x is an input variable, a is a randomly generated constant value, and `j` is randomly selected from the scope of the process identifiers. The mutations for `IRecv(i,r)` and `IRecv(*,r)` are similar. The goal of the first rule is to improve the program performance and simplify programming, while the second rule is to make the communication more deterministic. For each program, we generate five mutants if possible or generate as many as the number of receives.

The time limit for generating models for a program under a specific number of processes is 1 h. All the models in our benchmark are generated on a server with 32 Xeon 2.5 G cores and 256 G memory.

When MPI-SV analyzes an MPI program, as depicted in Subsection 2.4, it generates a model for a terminated path along which no violation of the critical property happens. The communications that occurred along the path are recorded in the model. Formally, if an MPI program is run under n processes, a model $\mathcal{M}$ of the program is a set of communication sequences $\{\mathsf{Proc}_i \mid 0 \leqslant i \leqslant n-1\}$, and each Proc_i is defined as the syntax in Figure 3, where $0 \leqslant d, s \leqslant n-1$ are process identities, and $t, r \in \mathbb{N}$ stand for the status of the operation.

For example, the MPI program in Table 1 produces the following model, where the statuses t and r are omitted for the sake of simplicity.

10) kf-ray. Https://code.google.com/archive/p/kf-ray/.

```
Proc ::= Comm | Proc ; Proc
Comm ::= Ssend(d, t) | Recv(s, t) | Recv(*, t) | Barrier | ISend(d,r) |
         IRecv(d,t,r) | IRecv(*,t,r) | Wait(r)
```

Figure 3 Syntax of a model in benchmark.

Table 3 Number of models extracted from different mutants

Program	o	m1	m2	m3	m4	m5	Total
DTG	1	2	1	1	1	1	7
Integrate_mw	5	5	67	–	–	–	77
Diffusion2d	2	2	18	80	3	2	107
Gauss_elim	5	6	–	–	–	–	11
Heat	5	8	6	6	6	6	37
Pingpong	0	28	28	547	28	28	659
Mandelbrot	39	330	327	–	–	–	696
Image_manip	20	5	5	–	–	–	30
Kfray	1	5	641	5	–	–	652
ClustalW	5	5	5	17	5	5	42
Total	83	396	1098	656	43	42	2318

```
Proc0 = IRecv(*,1);Recv(1,1);Barrier
Proc1 = ISend(0,1);Barrier
Proc2 = ISend(0,1);Barrier
```

In addition to the communication sequences, for each model, we record a verification result produced by MPI-SV. A verification result of a model may be deadlock, deadlock-free, or timeout. The result can be used as a reference answer during evaluation. For example, the result of the model extracted from the program in Table 1 is deadlock.

Table 3 lists the results of the benchmark generation. The first column shows the names of programs and the remaining columns indicate the number of benchmarks under different mutants, where o represents the original program. In total, 2318 models have been generated from the MPI programs in Table 2. As indicated by Table 3, the number of the models extracted from different programs varies, which is mainly due to the different input symbolizations of the programs. In principle, a model represents an equivalent class of the program's behaviors, whose control and data dependencies are same, but the matches of wildcard receives are different. From Table 3, we can see that the number of the models extracted from original programs (c.f., column o) is not large, which is also the reason why mutations are carried out. In particular, since no wildcard receives are contained in the program, the number of models extracted from original `Pingpong` is 0. In contrast, we can extract models from the mutated `Pingpong` program because the wildcard receives has been generated in the mutation.

Table 4 displays the generation results from the perspective of parallelism scales. The first column shows the names of programs and the remaining columns list the number of benchmark models generated under different processes. As shown in Table 4, for some programs, the number of models under more number of processes is smaller than that under fewer processes. The reason for this is that a large number of processes enlarge the scale of parallelism and complicate the communication along paths. In addition, MPI-SV needs more time to verify path models. Hence, the paths explored under the same time limit become fewer. For most programs, the number of extracted models is largest under 4 or 6 processes.

3.3 Benchmark evaluation

In principle, the communication complexity of each benchmark model determines the complexity of model checking-based verification. Hence, to indicate the validity of our benchmark for evaluating model checkers, we first evaluate our benchmark based on the communication complexity of the models.

Table 4 Number of models in different scales of parallelism

Program	2	4	5	6	8	10	Total
`DTG`	–	–	7	–	–	–	7
`Integrate_mw`	3	3	–	3	59	9	77
`Diffusion2d`	–	37	–	70	–	–	107
`Gauss_elim`	2	2	–	2	3	2	11
`Heat`	13	6	–	6	6	6	37
`Pingpong`	659	–	–	–	–	–	659
`Mandelbrot`	48	243	–	169	150	86	696
`Image_manip`	6	6	–	6	6	6	30
`Kfray`	292	208	–	138	11	3	652
`ClustalW`	6	9	–	15	6	6	42
Total	1029	514	7	409	241	118	2318

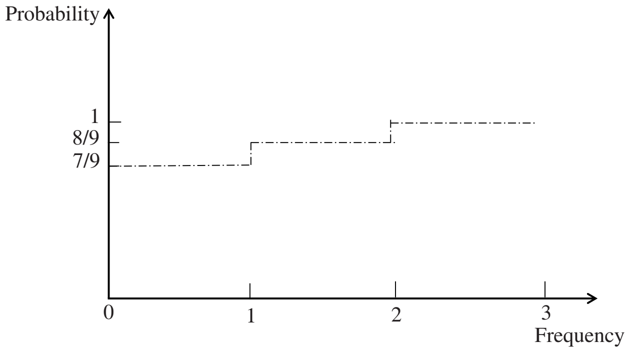

Figure 4 Cumulative distribution function.

The complexity of one process P in a model is determined by two aspects: (a) P's frequency of receiving messages, and (b) P's variety of receiving messages, i.e., the number of processes from which P receives messages. Hence, to measure the complexity of a model, we extract these static features of the model as a matrix M. Each element M_{ij} in the matrix indicates the frequency of receiving messages between the two processes, P_i and P_j, i.e., the number of receive operations of P_i that can match a send operation in P_j. For example, the matrix M of the model in Subsection 3.2 is

$$\begin{bmatrix} 0 & 2 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

M_{01} is 2 because both the receive operations of P_0, i.e., `IRecv(*,1)` and `Recv(1,1)`, can match the send operation `ISend(0,1)` of P_1. However, only `IRecv(*,1)` can match the send operation `ISend(0,1)` of P_2. As a result, M_{02} is 1. The number of non-zero values in ith line of M represents the variety of P_i. For example, the variety of P_0 is 2, whereas the variety of P_1 or P_2 is 0.

Using the matrix of a model, we calculate the complexity of the model based on the notion of cumulative distribution function (CDF), which is widely used in network complexity evaluation [22]. Given the matrix of a model M, the CDF of M is a function $F_M(x) : \mathcal{N} \to \mathcal{R}$, defined as follows:

$$F_M(x) := P(M_{ij} \leqslant x),$$

where $0 \leqslant i \leqslant n-1$, $0 \leqslant j \leqslant n-1$, and the right-hand side of this formula represents the probability that the element value in M is less than or equal to x. Taking the matrix M shown above as an example, Figure 4 shows the corresponding CDF.

Table 5 Complexity of programs

Program	2	4	6	8	10
`Integrate_mw`	3.5	15.25	35.17	113.24	131.82
`Diffusion2d`	–	48.28	110.67	–	–
`Gauss_elim`	4.38	28.65	75.39	146.30	240.37
`Heat`	12.93	42.51	85.80	141.46	211.41
`Pingpong`	23.75	–	–	–	–
`Mandelbrot`	10.26	38.10	67.59	97.49	149.37
`Image_manip`	5.87	18.58	29.65	40.24	50.61
`Kfray`	25.34	83.61	137.25	182.61	207.53
`ClustalW`	5.79	60.44	144.47	256.61	421.95

The complexity C of a model can be defined on the basis of the matrix and the CDF as follows:

$$\begin{aligned} &C := C_1 + C_2,\\ &C_1 := 1 + \int_0^{\mathrm{sum}} F_M(x)\mathrm{d}x,\\ &C_2 := \sum_{0\leqslant i,j\leqslant n-1}\left(1-\frac{|M_{ij}-\mathrm{avg}|}{\max(M_{ij},\mathrm{avg})}\right), \end{aligned} \tag{1}$$

where C_1 calculates the complexities of frequency and variety, and C_2 calculates the degree of distribution. In particular, the 1 in C_1 is the term of the Laplacian smoothing [23] in case C_1 is 0 when all the element values are zero; sum and avg stand for the summation and the average of the elements values in M, respectively. The complexity calculation ensures the following results:

• If the total frequency value sum of M is larger, i.e., there are plenty of communications in this model, the model tends to be more complex, which is ensured by C_1. The model is more complex when there are more processes in the model, which is also ensured by C_1.

• Under a fixed sum, if the distribution of the frequency values in M is more uniform, i.e., there are more process pairs participating in the communications, the model also tends to be more complex, which is ensured by C_2. For example, the complexities of the following two matrices M_1 and M_2 are 4.75 and 8, respectively.

$$M_1 = \begin{bmatrix} 0 & 4\\ 0 & 0\end{bmatrix},\quad M_2 = \begin{bmatrix} 1 & 1\\ 1 & 1\end{bmatrix}.$$

Meanwhile, the C_1 values of M_1 and M_2 are both 4.

Table 5 shows the average model complexities of each program under different numbers of processes. For each column in Table 5, the complexities between various programs show a large range of variation, especially when the number of processes is larger, e.g., 10 processes. Therefore, the models of the programs in our benchmark are different from each other with respect to the complexity; this also indicates that verification complexities of our benchmark are well distributed.

We also inspect the difference of the models from one program, which is shown in Figure 5. We can observe that the following:

• No matter what the number of processes is, the box related to `Mandelbrot` is wide enough to declare the diversity of the models from `Mandelbrot`.

• For the models of the remaining programs, the diversity only appears in a few cases, i.e., `ClustalW` under 4 and 6 processes, `Diffusion2d` under 4 and 6 processes, `Kfray` under 4, 6, and 8 processes, and `Integrate_mw` under 8 and 10 processes.

These two observations indicate that the benchmark models from a program under a fixed number of processes tend to have similar complexity.

In addition to the diversity in complexity, we also inspect the benchmark models with respect to the complexity of verification employing partial order reduction (POR) [24], which is one of the most popular

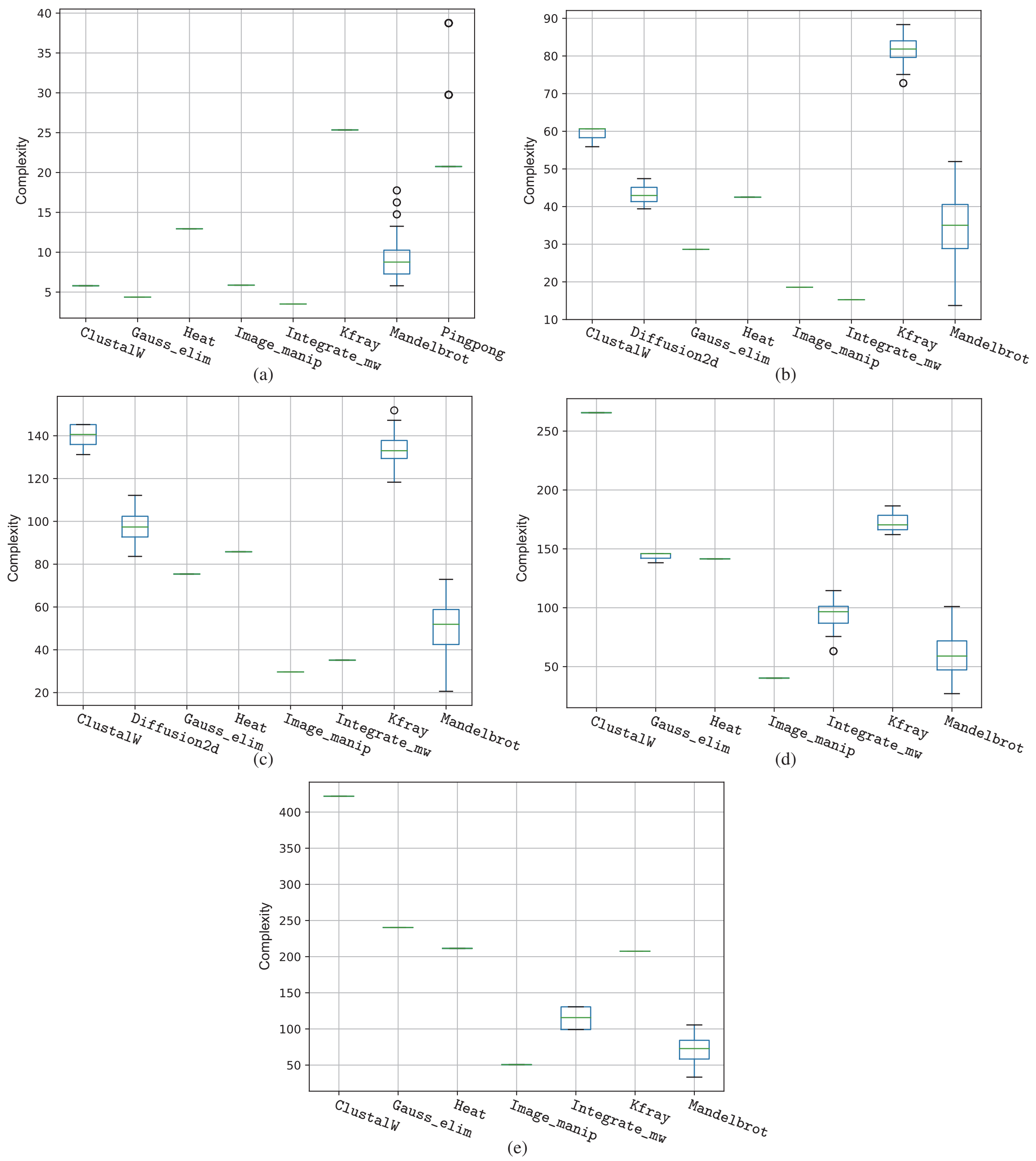

Figure 5 (Color online) Complexity in programs. Complexity under (a) 2 process; (b) 4 process; (c) 6 process; (d) 8 process; (e) 10 process.

reduction techniques applied in model checking and its application for concurrent systems improves the scalability. For MPI programs, if the processes are running independently or communicate seldom, POR has a good chance to be applied to reduce the state space during verification. If a model has many wildcard receive operations, the model tends to be difficult to apply POR in verification. Hence, we collect the average ratio of wildcard receive operations in the models of each program under different processes. Figure 6 shows the results and illustrates that the average rate of wildcard receives is less than 25% for the majority of the programs, which empirically indicates a large chance of applying POR during model checking. Therefore, our benchmark can be used to evaluate the design and implementation of POR in the model checker.

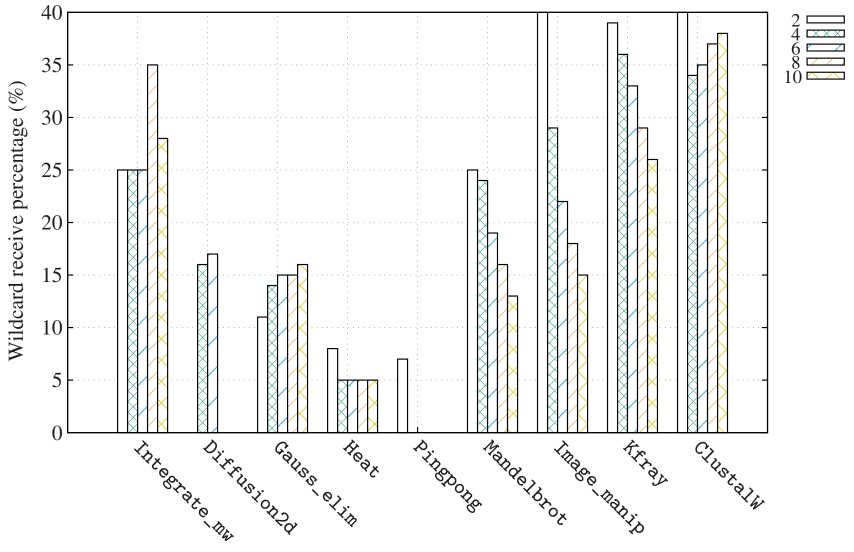

Figure 6 (Color online) Percentage of wildcard receive operations.

4 Translation algorithm

To evaluate different model checkers, we need to translate the models in our benchmark into the models in the input languages of the model checkers. Algorithm 1 shows the general framework of translation.

Algorithm 1 Benchmark translation procedure

```
Input: A benchmark model $M$, the number of processes $n$.
Output: A tool-specific model $M'$.
 1: for $i \leftarrow (0, \dots, n-1)$ do
 2:     $P'_i :=$ skip;    // the tool specific model for $P_i$.
 3:     for op $\leftarrow P_i$ do
 4:         if op = Barrier then
 5:             $P'_i := P'_i \ ; \ B$;
 6:         else if op = Wait(req) then
 7:             $P'_i := P'_i \ ; \ W_{\text{req}}?0$;
 8:         else if op = Ssend(obj, tag) then
 9:             $P'_i := P'_i \ ; \ C_{\text{obj}}!0$;
10:         else if op = ISend(obj, tag, req) then
11:             $P'_i := P'_i \ ; \ D_{\text{obj}}!0 \ ; \ W_{\text{req}}!0$;
12:         else if op = IRecv(obj, tag, req) then
13:             $P_r :=$ ROM($M$, op, $i$);
14:             $P'_i := P'_i \parallel P_r$;
15:         else if op = Recv(obj, tag) then
16:             $P_r :=$ ROM($M$, op, $i$);
17:             $P'_i := P'_i \ ; \ P_r$;
18:         end if
19:     end for
20: end for
21: $M' := \parallel \{P'_i \mid 0 \leqslant i \leqslant (n-1)\}$;    // all the $P'_i$ models synchronize on $B$.
22: return $M'$.
```

As described in Algorithm 1, given a benchmark model M, for each process P_i of M, we construct a tool-specific model of process P'_i to simulate P_i. Finally, all the models of the processes are parallel composed (line 21) to form the tool-specific model for M. For simplicity, we use channel based notations (e.g., channel read $C?x$ and channel write $C!a$) and CSP composition operators (e.g., sequential composition ; and parallel composition $\parallel$) to depict the algorithm.

Each process P_i is a sequence of MPI operations, and we handle each operation in a reverse order. For Barrier, we use an event B (i.e., special event for synchronization) composed with P'_i to indicate a synchronization (line 5). Given a Wait operation, we use a channel reading operation $W_{req}?0$, which blocks until a completion message is written to the corresponding channel $W_{\rm req}$ by the waited operation with req (line 7). When a send operation, i.e., Ssend or ISend is encountered, we compose P_i' with $C_{obj}!0$ (line 9) or $D_{\rm obj}!0$; $W_{\rm req}!0$ (line 11), where $C_{\rm obj}$ is a zero-sized channel and $D_{\rm obj}$ is a one-sized channel, and $W_{\rm req}!0$ indicates the completion of the operation. The challenging part is to model a receive operation, i.e., Recv or IRecv, especially when it is a wildcard receive. Algorithm 2 can be used to build the model of a receive operation, and then the model is composed with P_i' (lines 14 and 17 in Algorithm 1).

Algorithm 2 ROM(M, op, pid) //Receive operation modeling

```
Input: benchmark model $M$, operation op = recv(obj), and process number pid.
Output: the model for the receive operation.
 1: $\text{match}_s := \emptyset$;
 2: if $obj = *$ then
 3:     for $j \leftarrow (0, \dots, n-1)$ do
 4:         $\text{match}_s := \text{match}_s \cup \{\text{send(pid)} \mid \text{send(pid)} \in P_j\}$;   // send(pid) can be matched with op.
 5:     end for
 6: else if $\text{obj} = k$ then
 7:     $\text{match}_s := \text{match}_s \cup \{\text{send(pid)} \mid \text{send(pid)} \in P_k\}$;   // send(pid) can be matched with op.
 8: end if
 9: $P_r := \Box \{C_s?0 \mid s \in \text{match}_s\}$;
10: if op = IRecv(req) then
11:     $P_r := \text{refine}(P_r)$;
12:     $P_r := P_r \ ; \ W_{\rm req}!0$;
13: end if
14: return $P_r$.
```

The inputs of Algorithm 2 are a model M, the receive operation op, and the process identity pid. The key idea is to collect the possibly matched send operations and use channel read operations for modeling. If the receive operation is wildcard, i.e., obj is "$*$", we collect all the possibly matched send operations (`Ssend` or `ISend`) in all the processes (lines 3–5). If obj is k, we collect only those send operations in P_k (line 7). Thus, we create a channel read for each matched send operation and compose this channel read in a choice operator ($\Box$ at line 9), representing the possible communication in the processes. To satisfy the requirements in MPI standard, we need to refine the model (refine in line 11), e.g., ensuring the message receiving rules inside one process, and the detailed requirements can be referred to [7]. Similar to non-blocking send operation, if the receive operation is non-blocking, we add $W_{\rm req}!0$ in the end to indicate the completion status (line 12).

For example, according to Algorithm 1, we translate the model in Subsection 3.2 into the following CSP model:

```
Proc0 := ((D1_0?0->Skip [] D2_0?0->Skip); H0_0!0->Skip) || (H0_0?0->Skip; D1_0?0->Skip);
Proc1 := D1_0!0->Skip;
Proc2 := D2_0!0->Skip;
Model := Proc0 || Proc1 || Proc2;
```

The first three lines represent the models of the three processes in the example. Then, we combine these process models with parallel composition $\|$ to obtain the final model.

The two algorithms provide a general framework for modeling an MPI program path. The modeling method is sound and complete [7]. This means the model created by our method includes all the equivalent communication behaviors of the path by changing only the matches of the wildcard receive operations in the path. Moreover, the model is precise, i.e., the MPI program has all the communication behaviors in the model.

As indicated by the two algorithms, the channel-based and CSP-like languages are more suitable for modeling our benchmark models, which is determined by the nature of MPI programs, such as message passing and synchronization and parallel execution. For the composition operators, such as choice and synchronized parallel composition, we use the specific mechanisms in the modeling language of a model checker to implement them if the language does not directly support them in syntax.

5 Evaluation and results

This section presents the evaluation of the five state-of-the-art model checkers using the benchmark. We start with describing the experiment setup in Subsection 5.1, then the results are given in Subsections 5.2–5.4. Finally, we discuss the threats to validity in Subsection 5.5.

Based on the benchmark, we evaluate the five model checkers and try to answer the following questions.

RQ1: Which one of the model checkers is the most effective and efficient for deadlock freedom verification?

RQ2: What is the correctness of the model checkers for verifying deadlock freedom, including the consistency of the results produced by different model checkers and the runtime problems of the model checkers during verification?

RQ3: How convenient is the model checker for modeling message passing programs?

5.1 Experimental setup

We implemented the translation algorithms presented in Section 4 for the five selected model checkers. The verified property is deadlock freedom, which is supported by all the model checkers. The versions of the model checkers are PAT (v3.4.0), Spin (v6.4.8), FDR (v4.2.3), PRISM (v4.4.0), and NuSMV (v2.6.0). The time threshold for a verification task is one hour. All the verification tasks are carried out on a server with 32 Xeon 2.5 G cores and 256 G memory, and the OS is Ubuntu Linux 14.04.

5.2 Effectiveness and efficiency

Table 6 shows the main evaluation results. The first column lists the name of programs and the number of the corresponding models in our benchmark. The evaluated model checkers are displayed in the second column. The column verified models shows the number of successfully verified models by each model checker (both “Seg_fault” and “Memory_Error” mean there exists a runtime error). The last column average time lists the average time that the model checker used for verifying the models of the program (“–” means the average time is not available because all the models met the runtime error and “Timeout” means all the models failed to be verified within the time threshold). Note that the results where the corresponding model checker performs better are highlighted in bold. The criteria for better performance are verifying more models and using less time. For the 2318 models in the benchmark, PAT, Spin, FDR, PRISM, and NuSMV successfully verify 2308 (99%), 2318 (100%), 777 (34%), 656 (28%), and 489 (21%) models, respectively, in one hour. Hence, PAT and Spin have comparative effectiveness on deadlock freedom verification on our benchmark models. FDR can verify less than half of the models. Both of PRISM and NuSMV have relatively poor results, mainly because of the modeling language support for message passing programs, which will be discussed in Subsection 5.4. Figure 7 shows the percentages of verified models on each program for every model checker. Within the timeout threshold, NuSMV and PRISM fail to verify any models of three programs, i.e., `Diffusion2d`, `Pingpong`, and `Kfray`. In addition, FDR fails to verify any models of `Pingpong` and `Kfray`.

We also inspect the efficiency of the evaluated model checkers. Figure 8(a) shows the time costs (less than 15 s) of the model checkers on all verified models. Considering that NuSMV and PRISM need hundreds of seconds on average for most programs, we do not include NuSMV and PRISM in the figure. As shown in the figure, for the models that can be verified in less than 15 s, PAT and FDR have a comparative efficiency. Although Spin is less efficient than PAT and FDR, the number of exceptional points of Spin is smaller and the time costs are centralized, whereas FDR and PAT have more exception cases. These indicate that Spin has a stable performance.

To verify our conclusion further, we inspect the three model checkers on the same verified models (682) whose verification time costs are less than 15 s. Figure 8(b) shows the results. Compared with the results

Table 6 Experimental results

Program	Model checker	Verified models	Average time (s)
DTG (7)	PAT	**7**	0.29
	FDR	**7**	0.20
	SPIN	**7**	1.17
	PRISM	**7**	1.91
	NuSMV	**7**	**0.02**
Integrate_mw (77)	PAT	73	2.13
	FDR	75	8.27
	SPIN	**77**	**1.82**
	PRISM	68	126.62
	NuSMV	9	15.39
Diffusion2d (107)	PAT	**107**	**1.05**
	FDR	2	31.61
	SPIN	**107**	3.18
	PRISM	0 (Memory_Error)	–
	NuSMV	0 (Seg_fault)	–
Gauss_elim (11)	PAT	10	5.11
	FDR	9	13.12
	SPIN	**11**	5.26
	PRISM	2	**1.61**
	NuSMV	4	1468.03
Heat (37)	PAT	**37**	**0.30**
	FDR	24	9.60
	SPIN	**37**	2.24
	PRISM	13	2.76
	NuSMV	13	5.52
Pingpong (659)	PAT	**659**	**0.31**
	FDR	0	Timeout
	SPIN	**659**	3.36
	PRISM	0 (Memory_Error)	–
	NuSMV	0 (Seg_fault)	–
Mandelbrot (696)	PAT	694	**0.75**
	FDR	610	7.41
	SPIN	**696**	1.59
	PRISM	530	97.41
	NuSMV	428	265.73
Image_manip (30)	PAT	**30**	**0.41**
	FDR	25	0.67
	SPIN	**30**	1.34
	PRISM	**30**	7.00
	NuSMV	22	253.91
Kfray (652)	PAT	650	3.80
	FDR	0	Timeout
	SPIN	**652**	**3.36**
	PRISM	0	Timeout
	NuSMV	0	Timeout
ClustalW (42)	PAT	41	10.88
	FDR	25	89.68
	SPIN	**42**	5.66
	PRISM	6	1.59
	NuSMV	6	**0.05**

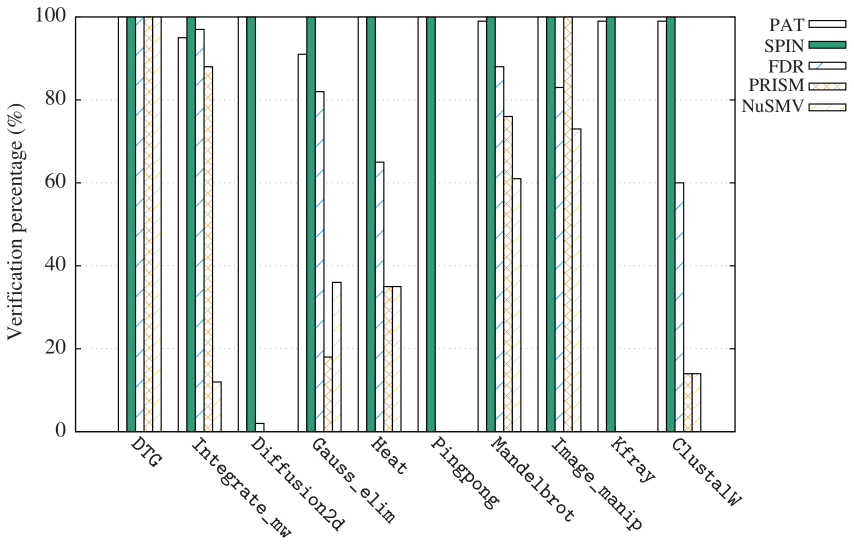

Figure 7 (Color online) Percentage of verified models.

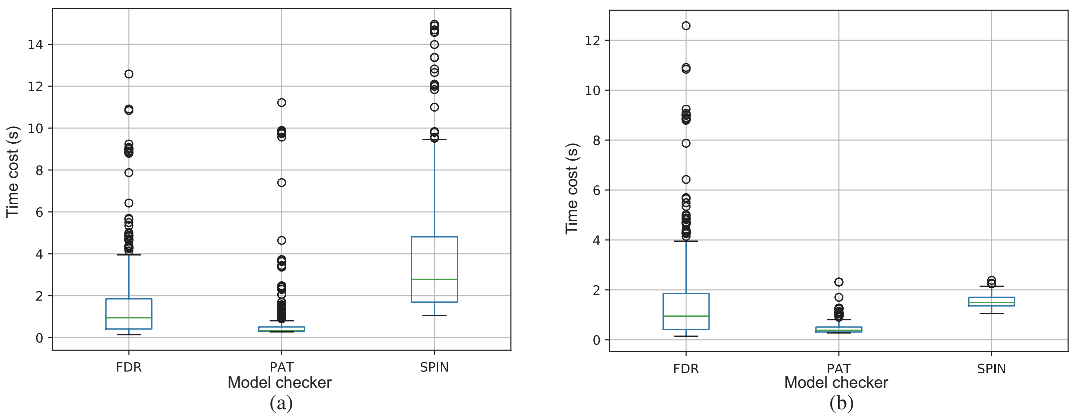

Figure 8 (Color online) Time costs of model checkers. (a) On the all models; (b) on the same models.

in Figure 8, Spin has a more comparative performance, i.e., it has less than 2 s on average on the same models. Similarly, in Figure 8, the time costs of Spin are also centralized on the same models. FDR has the most exceptional points.

Figure 9 shows the detailed information of the performance of each model checker on each MPI program except `Diffusion2d`, `Pingpong`, and `Kfray`, which are not verified successfully by all the model checkers. As shown in Figure 9, PAT, FDR, and Spin have a similar result on each program like that on all models shown in Figure 8. For example, of the models that can be verified in less than 15 s, PAT and FDR have a more efficient performance than Spin, and the data distribution of FDR is more widespread, e.g., `ClustalW` and `Heat`. In addition, in terms of the average time cost, we observe that PRISM takes more time to verify most of the models, e.g., `DTG`, `Image_manip`, and `Gauss_elim`. Furthermore, NuSMV shows a good performance when the models are simple and verifiable, e.g., `DTG` and `Integrate_mw` under 2 processes. However, NuSMV requires more time to verify the complex models, such as `Gauss_elim` and `Mandelbrot` under more than 4 processes. For these models, NuSMV requires at least 100 orders of magnitude more time costs than the first three model checkers.

Figure 10 shows the quantitative trend of each model checker for completing verification tasks under the time threshold. The X-axis is the time limit, which shows the results within 100 s only. The Y-axis shows the numbers of verified models. We do not include PRISM and NuSMV in the figure because of their poor performance within 100 s. As the figure indicates, all the three model checkers complete most of the tasks within 20 s. Spin verifies more models within the same time threshold than FDR and PAT.

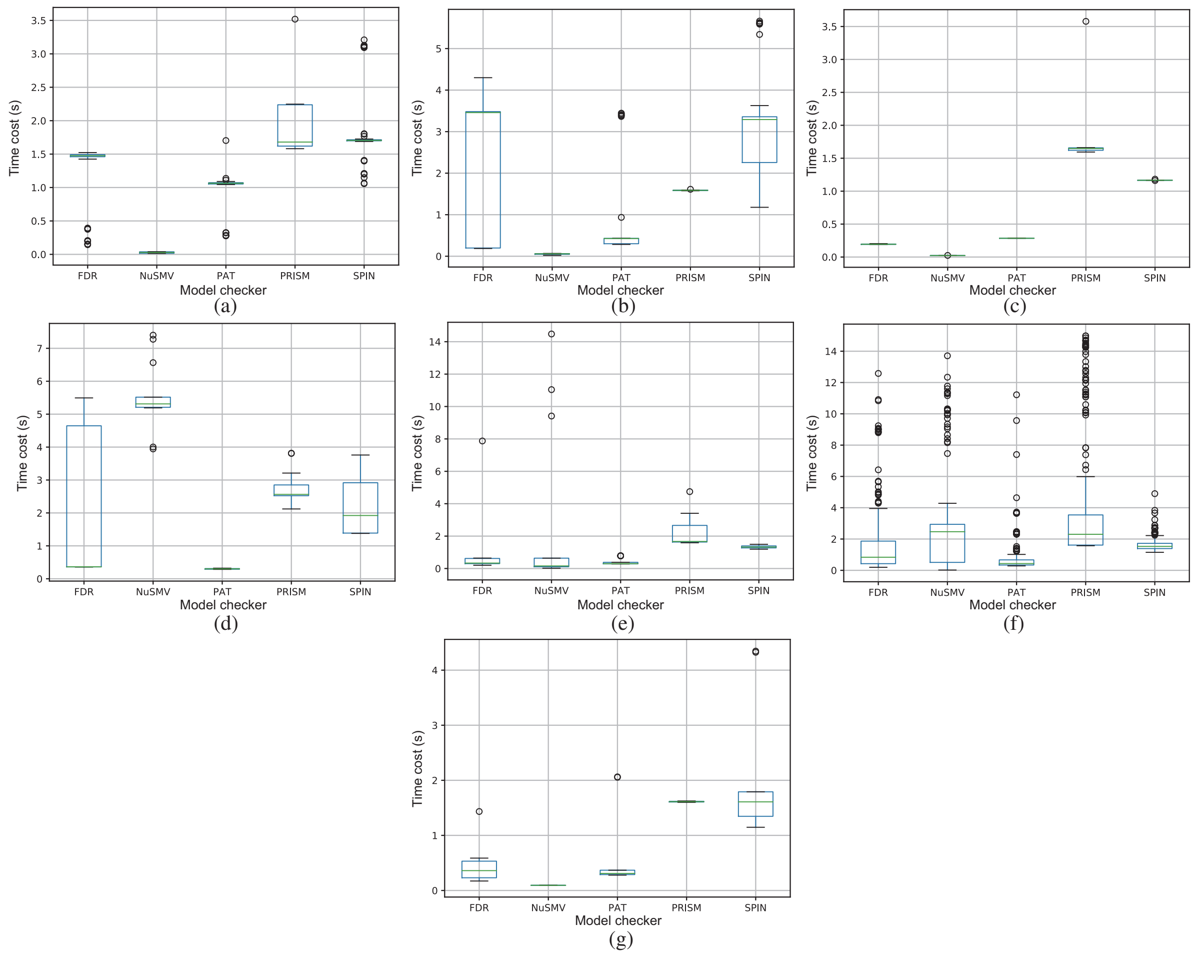

Figure 9 (Color online) Time costs of model checkers on each program. (a) Integrate_mw; (b) ClustalW; (c) DTG; (d) Heat; (e) Image_manip; (f) Mandelbrot; (g) Gauss_elim.

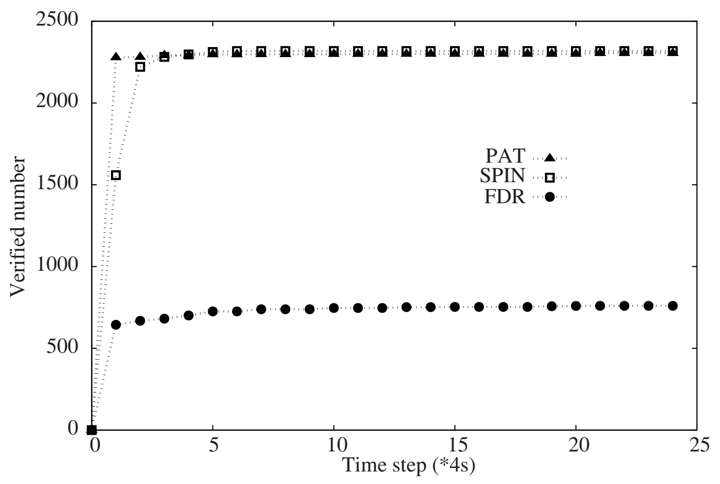

Figure 10 Trends of completion of verification tasks.

In addition, PAT and Spin verify two times more models than FDR under the time limit, indicating that PAT and Spin have better effectiveness and efficiency than the remaining two.

The number of processes determines the scale of parallelism. Models with a larger parallelism scale are more challenging for the model checkers than those with a smaller scale. We inspect the performance of each model checker on programs running under different numbers of processes. Figure 11 shows the

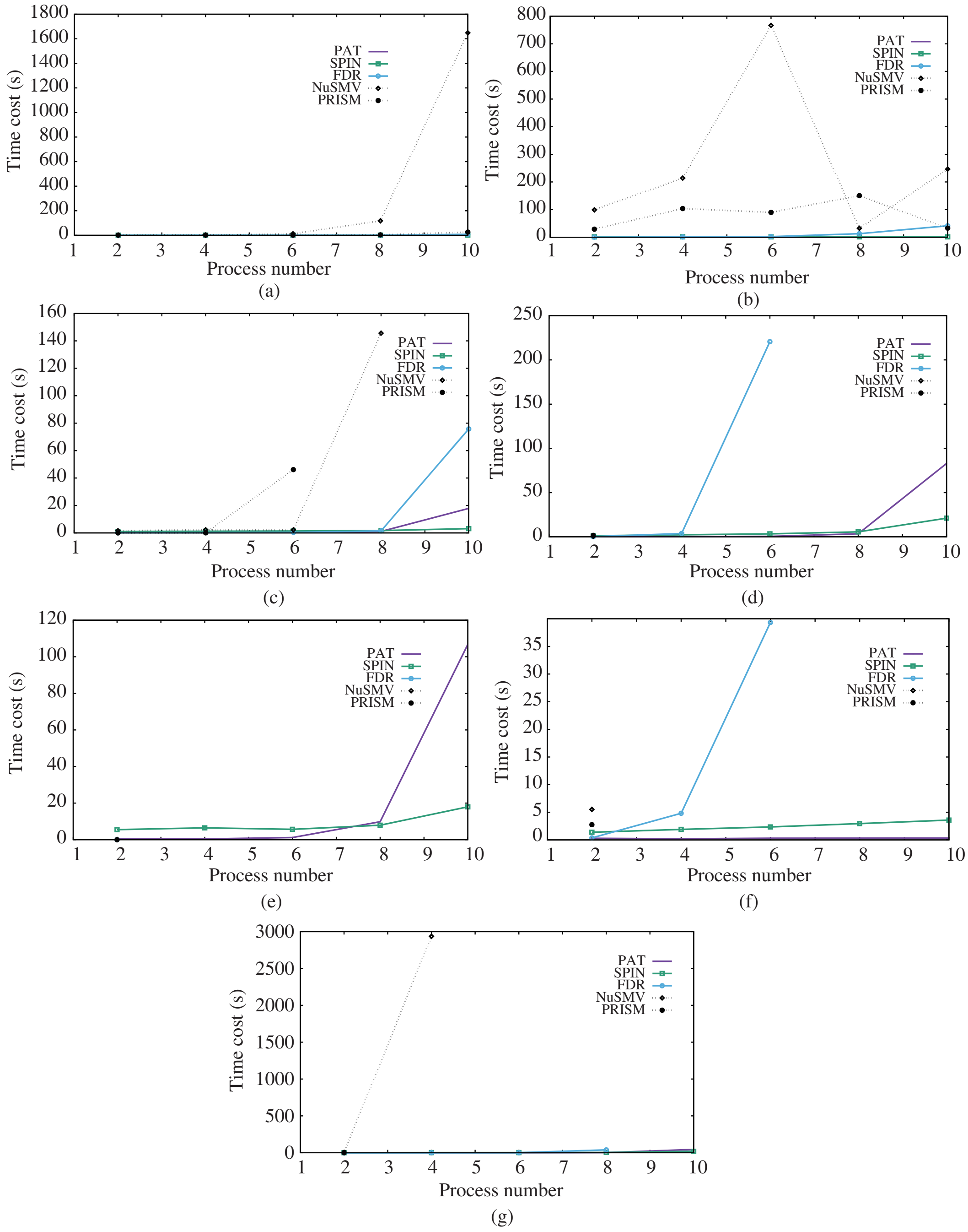

Figure 11 (Color online) Influence of parallelism scales. (a) Image_manip; (b) Mandelbrot; (c) Integrate_mw; (d) ClustalW; (e) Kfray; (f) Heat; (g) Gauss_elim.

results, which includes only seven programs, with their number of processes ranging 2, 4, 6, 8, and 10. As indicated by the figures, for most cases, a model checker needs more time when verifying models with a larger parallelism scale. Interestingly, the performance of Spin does not decrease immediately with increasing number of processes. However, the remaining model checkers, especially NuSMV, show a sharp decrease in the performance for large number of programs. This indicates that Spin has the more stable performance, consistent with what Figure 8 indicates.

As stated in Subsection 3.3 and Figure 6, our benchmark is suitable for evaluating the design and implementation of POR. The evaluation results of the five model checkers also indicate their supports for POR. Spin and PAT have good support for POR, while the support of FDR for POR is only partial and still experimental. Both PRISM and NuSMV are symbolic model checkers and they do not support POR. These facts are also consistent with the evaluation results, i.e., Spin and PAT perform better than the remaining three, and FDR performs better than PRISM and NuSMV.

With respect to the experimental results, we summarize the following answer for RQ1.

Answer to RQ1: Spin can verify all models in the benchmark within 1 h. Spin and PAT are more effective than the remaining three model checkers. In addition, they show better efficiency than the remaining three model checkers under the same time threshold. The performance of Spin is the most stable among the model checkers.

5.3 Correctness

In addition to effectiveness and efficiency, we can also check the correctness of the model checkers using the idea of differential testing [25]. If the results of the model checkers are different on any same model, there must be a problem in the implementation of at least one model checker. According to the verification results, we observe that there is no inconsistency when there are multiple model checkers that can produce a result on a model. This depicts that the implementations of the model checkers are of high quality.

During experiments, we also observed that NuSMV and PRISM had crashed many times due to memory problems. We collected the models that report "Segmentation Fault" during the NuSMV verification process, which includes 6 models in `ClustalW`, 70 models in `Diffusion2d`, and 60 models in `Pingpong`. Based on the source code of NuSMV, we located the place and found that the reason is a stack overflow. For the models that cause "Out of Memory Error" in the PRISM verification process, most of them are in the benchmark of `Diffusion2d` and `Pingpong`. In addition, FDR sometimes reports runtime errors such as "Can't Allocate Memory", "Double Free or Corruption (!Prev)", and "Corrupted Size vs. Prev_size". We collected the input models of these error cases and reported them to the developers. We expect feedback from them.

Answer to RQ2: In case of a successful verification, the model checkers are consistent on the verification results; this indicates the high quality of the implementations. NuSMV, PRISM, and FDR have runtime memory problems when verifying some models.

5.4 Convenience of modeling

The procedure of implementing the translation algorithms of Section 4 for each model checker is to use the modeling language of the model checker to model the MPI operations with respect to MPI standard. The convenience of the modeling language also influences the results of effectiveness and efficiency.

Channel-based modeling constructs, e.g., channel read, write and emptiness checking, are natural and effective for modeling message passing programs. PAT and Spin provide channel constructs in their modeling languages. Therefore, it is not hard for us to use the languages of PAT and Spin to implement the algorithms in Section 4. However, for FDR, PRISM, and NuSMV, we encountered some problems when implementing the translation algorithms.

Although both PAT and FDR verify the models in CSP, the channel constructs of PAT are more convenient than those of FDR. The input language of PAT, i.e., CSP#, has channel read, write, and emptiness checking constructs. The emptiness checking is a key to ensure that the completes-before relations [26] required by MPI standard are satisfied. However, the channel constructs of FDR regard channel operations as events, which complicates the modeling. Using more processes and events, we simulate channel operation. Hence, this channel operation adds numerous extra but necessary events, which makes models bigger and complicated. This is also the key reason why FDR performs worse than PAT and Spin.

The input language of NuSMV provides the constructs to build a labelled transition system for the system that to be verified. No channel operation construct exists in the language. Using the state-oriented language of NuSMV to model MPI operations is complicated. In addition, if the behaviour of a model is very non-deterministic, the modeling results in a large number of states and transitions. Furthermore, a choice operation is the key to modeling wildcard MPI operations, and every model in the benchmark has wildcard operations. However, we encounter a problem for modeling choice operation in NuSMV. In the beginning, we used `case` structure to model choice operation. Consider the following NuSMV model.

```
MODULE main
VAR
  pc : 1..3;
ASSIGN
  init(pc) := 1;
  next(pc) :=
    case
    pc = 1 & cons1 : 2;
    pc = 1 & cons2 : 3;
    esac;
```

In this model, `pc` denotes the current state. If `pc` is equal to 1 and `cons1` holds, the current state will be changed to 2; if `pc` is equal to 1 and `cons2` holds, the current state will be changed to 3. However, if the condition `pc` is equal to 1 holds and both cons1 and cons2 hold, then the current state will only be changed to 2. This is because the case structure is actually an `if-else` structure. Therefore, using a random selection method, we solve the problem. If several constraints are satisfied at the same time, we make a random selection. For the case structure problem demonstrated before, the following solution is adopted, where line 10 uses a random selection when both `cons1` and `cons2` hold.

```
 1  MODULE main
 2  VAR
 3    pc : 1..3;
 4  ASSIGN
 5    init(pc) := 1;
 6    next(pc) :=
 7      case
 8      pc = 1 & cons 1 & not cons2 : 2;
 9      pc = 1 & cons 2 & not cons1 : 3;
10      pc = 1 & cons 1 & cons2 :{2, 3};
11      esac;
```

However, this solution results in two choices. If there are n choices, then we have to define $2^n - 1$ transitions and states. Hence, the number of choices increases the number of states and transitions rapidly, which is also a reason that NuSMV has a poor performance. The same problem also occurs to PRISM.

Answer to RQ3: PAT and Spin provide more convenient constructs for modeling message passing programs. The convenience of modeling directly influences the effectiveness and efficiency of verification.

5.5 Threats to validity

There are external and internal threats to the validity of our results. The external threats come from the limited MPI programs used for generating benchmarks. However, we relax these external threats in the following three aspects:

- All the MPI programs are real-world MPI programs and some of them are used as the benchmark programs in the previous [27, 28];
- These programs are mixed blocking and non-blocking programs, and the scales are beyond the state-of-the-art static verification tools for MPI programs;
- Although the number of the models in our benchmark is limited, as far as we know, the scale of our benchmark is already large (c.f. related work discussion in Section 6).

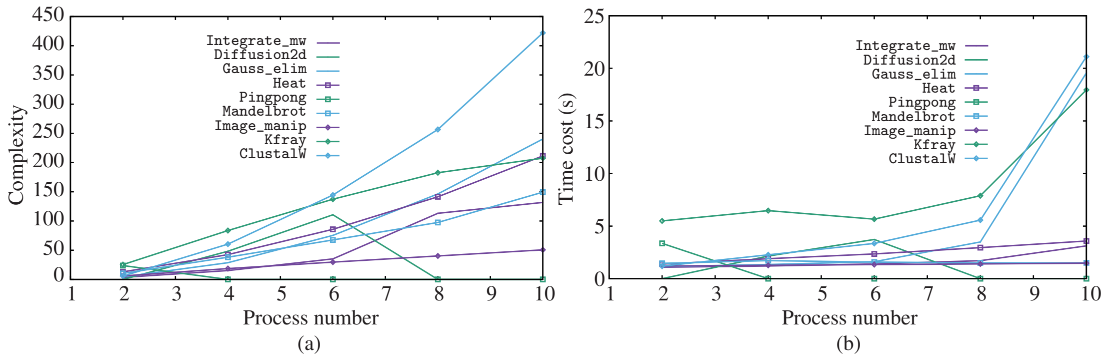

Figure 12 (Color online) Complexity comparison between model and verification. (a) The trend of model complexity; (b) the trend of verification complexity (SPIN).

In addition, we plan to address this threat further in the future by analyzing more real-world MPI programs and adding more models to the benchmark. Furthermore, the evaluated model checkers with state-based input language, i.e., NuSMV and PRISM, are naturally not good at modeling message passing programs, which may make our evaluation results biased. However, our evaluation empirically validates this insight and provides a quantitative result of how inefficient these model checkers are in verifying message passing programs. In addition, the model checker PRISM is designed for probabilistic systems; however, our benchmark does not contain probabilistic models, which may also be biased.

The internal threats come from our evaluation methods for benchmark and our implementations of the translation algorithms for model checkers. We controlled the threats by testing each implementation and drawing the following conclusion:

• The consistency of the results of different model checkers indicates the high quality of our implementation.

• To check if the model complexity reflects some trends of verification complexity, we compared the difference between model complexity and verification complexity. Figure 12(a) represents the trend of model complexity under different process numbers, while Figure 12(b) shows for the corresponding verification time by Spin. As demonstrated in Figure 12, the similarity between the trends of the lines in the two figures indicates the criterion for evaluating model complexity and the modeling are reasonable.

• We empirically validate the complexity evaluation of POR in Subsection 3.3. For Spin, under the same complexity and process number, the model with fewer wildcard receive operations is verified faster. For example, the average complexities of `Kfray` and `Heat` under 10 processes are 207.53 and 211.41, respectively. The corresponding wildcard receive percentages are 26% and 5%, respectively, whereas their average verification time costs are 17.95 and 3.58 s, respectively. These results depict the validity of our complexity evaluation of POR.

6 Related work

Our work is closely related to the existing benchmark, evaluation, and contest work of model checking. Next, we discuss the related work and make a comparison.

BEEM [3] is a benchmark for explicit model checkers. Insides BEEM, there are 50 parametrized models and their properties (safety or liveness properties) to verify. BEEM has been used by many later studies [29–31] as the benchmark for evaluating LTL model checking. Most models in BEEM are well-known examples and case studies. Unlike the models in BEEM, which are manually created, the models in our benchmark are extracted automatically from MPI programs. Moreover, our benchmark has more models. However, our benchmark is only concerned with verifying the deadlock freedom property. Model checking contest is an event for evaluating the model checkers for Petri nets. The benchmarks of the contest are Petri nets models created from representative case studies. The properties are CTL, LTL, and reachability. Similar to BEEM, the models used by the contest are created manually by different model contributors. Kwiatkowska et al. [32] presents a benchmark for probabilistic model checking. In the benchmark, thirty models covering four types of probabilistic models exist, and each model is accompanied by several probabilistic properties. There is also a benchmark for asynchronous concurrent systems [33]. In the same manner as before, the models in the benchmark are created manually.

A hardware model checking competition (HWMCC) is held annually from 2006. The benchmarks of HWMCC are unified in AIGER format[11]. The benchmarks are from hardware design, e.g., manually created or randomly generated with respect to hardware designs. Multiple tracks are designed for different types of properties, such as safety and liveness. After the development for more than 10 years, the number of models in the benchmark of HWMCC is near two thousand. Compared with the benchmark of HWMCC, we extract our benchmark from real-world MPI software, which cares more about deadlock freedom verification.

Similar to HWMCC, a software verification competition event, i.e., SV-Comp, is held each year. The competition aims to evaluate the verification tools for a software system. Most benchmarks of SV-Comp are preprocessed C programs, many of which are manually designed. Some benchmarks are also extracted from real-world programs, such as Linux driver programs. Different tracks of SV-Comp exist for different kinds of properties and programs. The concurrency safety track verifies concurrent programs. However, a message passing program is not included in this track and it has only multi-threaded programs. Compared with the benchmark of SV-Comp, our benchmark aims to evaluate the model checkers whose inputs are models instead of code. Moreover, our benchmarks are all generated from real-world MPI programs.

Furthermore, work of evaluating model checkers by applying them to verify a system. Frappier et al. [2] compared six model checkers for verifying an information system, aiming to identify the required features of model checkers for verifying information systems. Fifteen properties specified by different formalisms, e.g., LTL and CTL are verified. Similar to that of [2], Pamela compared Alloy[12] and Spin using both of them to verify a distributed hash table system. These approaches mainly care about the expressiveness of the modeling and specification languages. Compared with them, we evaluate model checkers by a large number of models in our benchmark, instead of some single system; we also compare the performances of the model checkers for deadlock freedom verification.

Our stduy is based on MPI-SV, which is related to the automatic program analysis work for MPI programs. Existing studies of automatically analyzing MPI programs can be divided into dynamic and static ones. Dynamic approaches, e.g., ISP [27] and MOPPER [28], create a model that runs on an MPI program, which depends on specific inputs. Therefore, for generating models, test inputs are needed. However, creating test inputs is also a labor-intensive work. Static approaches, e.g., TASS [15] and CIVL [6], abstract the whole program in a model for verification, and fewer ones support non-blocking MPI programs, which are frequently used in real-world MPI programs. Hence, the existing approaches are not appropriate for benchmark generation, which is also the reason that we use MPI-SV, which uses symbolic execution to tackle the problem of creating test inputs and supports the analysis of mixed blocking and non-blocking MPI programs.

7 Conclusion and future work

Benchmarks and evaluation are very important for developing model checking techniques. This study creates a benchmark based on MPI-SV for evaluating existing model checkers by verifying message passing programs. In the benchmark, 2318 models are automatically extracted from real-world MPI programs by MPI-SV. Based on the benchmark, we evaluate the five state-of-the-art model checkers in three aspects: effectiveness and efficiency, correctness and convenience of modeling. The evaluation results indicate that Spin is most effective for verifying deadlock freedom, and the modeling languages of FDR, PRISM, and NuSMV are not intuitive for modeling message passing programs. Our benchmark can be downloaded at the address[13].

11) AIGER Website. Http://fmv.jku.at/aiger/.
12) Alloy Website. Http://alloytools.org/.
13) https://github.com/mc-benchmark/mpi-benchmark.

The future work lies in two aspects: (1) enlarge the scale of the benchmark by extracting the models from more MPI programs and add more properties for verification; (2) develop the translators for more model checkers for evaluation, e.g., Petri nets model checkers.

Acknowledgements This work was supported by National Key R&D Program of China (Grant No. 2017YFB1001802) and National Natural Science Foundation of China (Garnt Nos. 61472440, 61632015, 61690203, 61532007).

References

1 Clarke E M, Grumberg O, Peled D A. Model Checking. Cambridge: MIT Press, 2001
2 Frappier M, Fraikin B, Chossart R, et al. Comparison of model checking tools for information systems. In: Proceedings of the 12th International Conference on Formal Engineering Methods, 2010. 581–596
3 Pelánek R. BEEM: benchmarks for explicit model checkers. In: Proceedings of the 14th International SPIN Workshop on Model Checking Software, 2007. 263–267
4 Gopalakrishnan G, Kirby R M, Siegel S F, et al. Formal analysis of MPI-based parallel programs. Commun ACM, 2011, 54: 82–91
5 Siegel S F. Verifying parallel programs with mpi-spin. In: Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2007. 13–14
6 Luo Z Q, Zheng M C, Siegel S F. Verification of MPI programs using CIVL. In: Proceedings of the 24th European MPI Users' Group Meeting, 2017. 6: 1–11
7 Yu H B, Chen Z B, Fu X J, et al. Combining symbolic execution and model checking to verify MPI programs. 2018. ArXiv: 1803.06300
8 King J C. Symbolic execution and program testing. Commun ACM, 1976, 19: 385–394
9 Gibson-Robinson T, Armstrong P, Boulgakov A, et al. A modern refinement checker for CSP. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 2014. 187–201
10 Lattner C. Llvm and clang: next generation compiler technology. In: Proceedings of the BSD Conference, 2008. 1–2
11 Hoare C A R. Communicating Sequential Processes. Upper Saddle River: Prentice-Hall, 1985
12 Scattergood J B. The semantics and implementation of machine-readable CSP. Dissertation for Ph.D. Degree. Oxford: University of Oxford, 1998
13 McMillan K L. Symbolic model checking. Norwell: Kluwer Academic Publishers, 1993
14 Baier C, Katoen J. Principles of Model Checking. Cambridge: MIT Press, 2008
15 Siegel S F, Zirkel T K. TASS: the toolkit for accurate scientific software. Math Comput Sci, 2011, 5: 395–426
16 Xue R N, Liu X Z, Wu M, et al. Mpiwiz: subgroup reproducible replay of mpi applications. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2009. 251–260
17 Müller M, de Supinski B, Gopalakrishnan G, et al. Dealing with mpi bugs at scale: Best practices, automatic detection, debugging, and formal verification. 2011
18 Vakkalanka S. Efficient dynamic verification algorithms for MPI applications. 2010
19 Thompson J D, Higgins D G, Gibson T J. Clustalw: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res, 1994, 22: 4673–4680
20 Lattner C, Adve V S. LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004), 2004. 75–88
21 Just R, Jalali D, Inozemtseva L, et al. Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014. 654–665
22 Newman M E J. The structure and function of complex networks. SIAM Rev, 2003, 45: 167–256
23 Hermann L R. Laplacian-isoparametric grid generation scheme. J Eng Mech Div, 1976, 102: 749–907
24 Godefroid P. Partial-order methods for the verification of concurrent systems — an approach to the state-explosion problem. In: Lecture Notes in Computer Science. Berlin: Springer, 1996
25 McKeeman W M. Differential testing for software. Digit Tech J, 1998, 10: 100–107
26 Vakkalanka S S, Gopalakrishnan G, Kirby R M. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Proceedings of the 20th International Conference on Computer Aided Verification, 2008. 66–79
27 Vakkalanka S S, Sharma S, Gopalakrishnan G, et al. ISP: a tool for model checking MPI programs. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008. 285–286
28 Forejt V, Joshi S, Kroening D, et al. Precise predictive analysis for discovering communication deadlocks in MPI programs. ACM Trans Program Lang Syst, 2017, 39: 1–27
29 Blom S, van de Pol J, Weber M. Ltsmin: distributed and symbolic reachability. In: Proceedings of the 22nd International Conference on Computer Aided Verification, 2010. 354–359
30 Lal A, Reps T W. Reducing concurrent analysis under a context bound to sequential analysis. Form Methods Syst Des, 2009, 35: 73–97
31 Laarman A, van de Pol J, Weber M. Boosting multi-core reachability performance with shared hash tables. In: Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, 2010. 247–255
32 Kwiatkowska M Z, Norman G, Parker D. The PRISM benchmark suite. In: Proceedings of the 9th International Conference on Quantitative Evaluation of Systems, 2012. 203–204
33 Atiya D A, Catano N, Lüttgen G. Towards a benchmark for model checkers of asynchronous concurrent systems. In: Proceedings of the 5th International Workshop on Automated Verification of Critical Systems (AVOCs), 2005. 98: 142–170