Specification, Design, and Implementation of Particular Classes of Systems

CLTestCheck: Measuring Test Effectiveness for GPU Kernels

Chao Peng(✉) and Ajitha Rajan

University of Edinburgh, Edinburgh, UK
{chao.peng,arajan}@ed.ac.uk

Abstract. Massive parallelism, and energy efficiency of GPUs, along with advances in their programmability with OpenCL and CUDA programming models have made them attractive for general-purpose computations across many application domains. Techniques for testing GPU kernels have emerged recently to aid the construction of correct GPU software. However, there exists no means of measuring quality and effectiveness of tests developed for GPU kernels. Traditional coverage criteria over CPU programs is not adequate over GPU kernels as it uses a completely different programming model and the faults encountered may be specific to the GPU architecture.

We address this need in this paper and present a framework, CLTestCheck, for assessing quality of test suites developed for OpenCL kernels. The framework has the following capabilities, 1. Measures kernel code coverage using three different coverage metrics that are inspired by faults found in real kernel code, 2. Seeds different types of faults in kernel code and measures fault finding capability of test suite, 3. Simulates different work-group schedules to check for potential deadlocks and data races with a given test suite. We conducted empirical evaluation of CLTestCheck on a collection of 82 publicly available GPU kernels and test suites. We found that CLTestCheck is capable of automatically measuring effectiveness of test suites, in terms of kernel code coverage, fault finding and revealing data races in real OpenCL kernels.

Keywords: Testing · Code coverage · Fault finding · Data race · Mutation testing · GPU · OpenCL

1 Introduction

Recent advances in the programmability of Graphics Processing Units (GPUs), accompanied by the advantages of massive parallelism and energy efficiency, have made them attractive for general-purpose computations across many application domains [19]. However, writing correct GPU programs is a challenge owing to many reasons [13] – a program may spawn millions of threads, which are clustered in multi-level hierarchies, making it difficult to analyse; programmer assumes responsibility for ensuring concurrently executing threads do not conflict by checking threads access disjoint parts of memory; complex striding patterns of memory accesses are hard to reason about; GPU work-group execution model and thread scheduling vary platform to platform and the assumptions are not

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 315–331, 2019.
https://doi.org/10.1007/978-3-030-16722-6_19

explicit. As a consequence of these factors, GPU programs are difficult to analyse with existing static or dynamic approaches [13]. Static techniques are thwarted by the complexity of the sharing patterns. Dynamic techniques are challenged by the combinatorial explosion of thread interleavings and space of possible data inputs. Given these difficulties, it becomes important to understand the extent to which a GPU program has been analysed and tested, and the code portions that may need further attention.

In this paper, we focus on GPU program testing and address concerns with respect to quality and adequacy of tests developed for GPU programs. We present a framework, CLTestCheck, that measures test effectiveness over GPU kernels written using OpenCL programming model [7]. The framework has three main capabilities. The first capability is a technique called *schedule amplification* to check execution of test inputs over several work-group schedules. Existing GPU architecture and simulators do not provide a means to control work-group schedules. The OpenCL specification provides no execution model for inter work-group interactions [21]. As a result, the ordering of work-groups when a kernel is launched is non-deterministic and there is, presently, no means for checking the effect of schedules on test execution. We provide this monitoring capability. For a test case T_i in test suite TS, instead of simply executing it once with an arbitrary schedule of work-groups, we execute it many times with a different work-group schedule in each execution. We build a simulator that can force work-groups in a kernel execution to execute in a certain order. This is done in an attempt to reveal test executions that produce different outputs for different work-group schedules which inevitably point to problems in inter work-group interactions.

The second capability of CLTestCheck is measuring code coverage for OpenCL kernels. The structures we chose to cover were motivated by OpenCL bugs found in public repositories like Github and research papers for GPU testing. We define and measure coverage over synchronisation statements, loop boundaries and branches in OpenCL kernels.

The final capability of the framework is creating mutations by seeding different classes of faults relevant to GPU kernels. We assess the effectiveness of test suites in uncovering the seeded faults.

We empirically evaluate CLTestCheck using 82 kernels and associated test input workloads from industry standard benchmarks. The schedule amplifier in CLTestCheck was able to detect deadlocks and inter work-group data races in benchmarks. We were able to detect barrier divergence and kernel code that requires further tests using the coverage measurement capabilities of CLTestCheck. Finally, the fault seeding capability was able to expose unnecessary barriers and unsafe accesses in loops.

The CLTestCheck framework aims to help developers assess how well the OpenCL kernels have been tested, kernel regions that require further testing, uncover bugs sensitive to work-group schedules. In summary, the main contributions in this paper are:

1. Schedule amplification to evaluate test executions using different work-group schedules.
2. Definition and measurement of kernel code coverage considering synchronisation statements, loop boundaries and branch conditions.

3. Fault seeder for OpenCL kernels that seeds faults from different classes. The seeded faults are used to assess the effectiveness of test suites with respect to fault finding.
4. Empirical evaluation on a collection of 82 publicly available GPU kernels, examining coverage, fault finding and inter work-group interactions.

The rest of this paper is organised as follows. We present background on the OpenCL programming model in Sect. 2. Related work in GPU program testing and verification is discussed in Sect. 3. CLTestCheck capabilities is discussed in Sect. 4. Experiment setup and results of our empirical evaluation is discussed in Sects. 5 and 6, respectively.

2 Background

The success of GPUs in the past few years has been due to the ease of programming using the CUDA [17] and OpenCL [7] parallel programming models, which abstract away details of the architecture. In these programming models, the developer uses a C-like programming language to implement algorithms. The parallelism in those algorithms has to be exposed explicitly. We now present a brief overview of the core concepts of OpenCL, the programming model used in this paper.

OpenCL is a programming framework and standard set from Khronos, for heterogeneous parallel computing on cross-vendor and cross-platform hardware. In the OpenCL architecture, CPU-based *Host* controls multiple *Compute Devices* (for instance CPUs and GPUs are different compute devices). Each of these coarse grained compute devices consists of multiple *Compute Units* which in turn contain one or more *processing elements* (a.k.a *streaming processors*). The processing elements execute groups of individual threads, referred to as work-groups, concurrently. The functions executed by the GPU threads are called *kernels*, parameterised by thread and group id variables. OpenCL has four types of memory regions: global and constant memory shared by all threads in all work-groups, local memory shared by threads within the same work-group and private memory for each thread. Kernels cannot write to the constant memory.

GPUs have SIMT (single instruction, multiple thread) execution model that executes batches of threads (warps) in *lock-step*, i.e all threads in a work-group execute the same instruction but on different data. If the control flow of threads within the same work-group diverges, the different execution paths are scheduled sequentially until the control flows reconverge and lock-step execution resumes. Sequential scheduling caused by divergence results in a performance penalty, slowing down execution of the kernel.

Betts et al. [2] describe two specific classes of bugs that make GPU kernels harder for verification than sequential code, data races and barrier divergence. *Inter work-group data race* is referred to as a global memory location is written by one or more threads from one work-group and accessed by one or more threads from another work-group. *Intra work-group data race* is referred to as a global or local memory location is written by one thread and accessed by another from the same work-group. Barrier is a synchronisation mechanism for threads within a work-group in OpenCL and is used to prevent intra work-group data race errors.

Barrier divergence occurs if threads in the same group reach different barriers, in which case kernel behaviour is undefined [2] and may lead to intra work-group data race.

In this paper, we focus on covering barrier functions to help detect intra work-group barrier divergence errors and revealing problems with inter work-group interactions using work-group schedule amplification.

3 Related Work

We discuss related work in the context of work-group synchronisation, verification and testing of GPU programs.

Inter Work-group Synchronisation for OpenCL Kernels. Barrier functions in the OpenCL specification [7] help synchronise threads within the same work-group. There is no mechanism, however, to synchronise threads belonging to different work-groups. One solution for this problem is to split a program into multiple kernels with the CPU executing the kernels in sequence providing implicit synchronisation. The drawback with this method is the overhead incurred in launching multiple kernels. Xiao et al. [24] proposed an implementation of inter work-group barrier that relies on information on the number of work-groups. This method is not portable as the number of launched work-groups depends on the device. Sorensen et al. [22] extended it to be portable by discovering work-group occupancy dynamically. Their implementation of inter work-group barrier synchronisation is useful when the developer knows there is interaction between work-groups that needs to be synchronised. Our contribution is in detecting undesired inter work-group interactions, not intended by the developer.

GPU Kernel Verification. Verification of GPU kernels to detect data races and barrier divergence bugs has been explored in the past. Li et al. [14] introduced a Satisfiability Modulo Theories (SMT) based approach for analysing GPU kernels and developed a tool called Prover of User GPU (PUG). The main drawback of this approach is scalability. With an increasing number of threads, the number of possible thread interleavings grows exponentially, making the analysis infeasible for large number of threads. GRace [25] and GMRace [26] were developed for CUDA programs to detect data races using both static and dynamic analysis. However, they do not support detection of inter work-group data races.

GKLEE [15] and KLEE-CL [3], based on dynamic symbolic execution, provides data race checks for CUDA and OpenCL kernels, respectively. Both tools are restricted by the need to specify a certain number of threads, and the lack of support for custom synchronisation constructs. Scalability and general applicability is a challenge with these tools.

Leung et al. [13] present a flow-based test amplification technique for verifying race freedom and determinism of CUDA kernels. For a single test input under a particular thread interleaving, they log the behaviour of the kernel and check the property. They then amplify the result of the test to hold over all the inputs that have the same values for the property integrity-inputs. The test amplification approach in [13] can check the absence of data-races, not the presence. Additionally, their approach amplifies across the space of test inputs, not work-group

schedules as done in our schedule amplifier. GPUVerify [2] is a static analysis tool that transforms a parallel GPU kernel into a two-threaded predicated program with lock-step execution and checks data races over this transformed model. The drawback of GPUVerify is that it may report false alarms and has limited support for atomic operations.

Test Effectiveness Measurement. Measuring effectiveness of tests in terms of code coverage and fault finding is common for CPU programs [6,18]. Support for GPU programs is scarce. GKLEE is the only tool that provides support for code coverage for CUDA GPU kernels. Given a kernel, it converts it into its sequential C program version (using Perl scripts) and applies the Gcov utility supplied with GCC for measuring code coverage. This form of coverage measurement disregards the GPU programming model. In our approach, we measure coverage conforming to the OpenCL programming model. With respect to fault seeder and schedule amplification, we are not aware of any existing work that provides these capabilities for GPU kernels to help measure effectiveness of test suites. The CLTestCheck framework is discussed in the next Sect. 4.

4 Our Approach

In this Section, we present the CLTestCheck framework that provides capabilities for kernel code coverage measurement, mutant generation and schedule amplification. To understand the kinds of programming bugs[1] encountered by OpenCL developers, we surveyed several publicly available OpenCL kernels and associated bug fix commits. A summary of our findings is shown in Table 1. We found bugs most commonly occur in the following OpenCL code constructs: barriers, loops, branches, global memory accesses and arithmetic computations. We seek to aid the developer in assessing quality of test suites in revealing these bug types using CLTestCheck. A detailed discussion of CLTestCheck capabilities is presented in the following sections.

4.1 Kernel Code Coverage

We define coverage over barriers, loops and branches in OpenCL code to check rigour of test suites in exercising these code structures.

Branch Coverage. GPU programs are highly parallelised, executed by numerous processing elements, each of them executing groups of threads in lock step, which is very different from parallelism in CPU programs, where each thread executes different instructions with no implicit synchronisation, as seen in lock-step execution. Kernel code for all the threads is the same, however, the threads may diverge, following different branches based on the input data they process. As seen in Table 1, uncovered branches and branch conditions are an important class of OpenCL bugs. Lidbury et al. [16] report in their work that branch coverage

[1] These are kernel bugs that violate the specification of the program or are associated with executions that lead to undefined behaviour.

Table 1. Summary of bug fixing commits we collected

#	Code Structure	Bug Type	Repository
1	Barrier	Missing barriers	Winograd-OpenCL [10], histogram [13], reduction [13], OP2 [3]
2		Removing unnecessary barriers	Winograd-OpenCL [10]
3	Loop	Incorrect condition	mcxcl [5], particles [8]
4		Incorrect boundary value	clSPARSE [1]
5		Missing loop boundary	Pannotia [21]
6	Branch	Missing else branch	liboi [11]
7		Incorrect condition	mcxcl [5], ClGaussianPyramid [4]
8	Global memory access	Inter work-group data race	Parboil-spmv [16], lonestar-bfs [21], lonestar-sssp [21]
9	Arithmetic Computations	Incorrect arithmetic operators	mcxcl [5], ClGaussianPyramid [4]

measurement is crucial for GPU programs but is currently lacking. To address this need, we define branch coverage for GPU programs as follows,

$$branch\ coverage = \frac{\#covered\ branches}{total\ \#branches} \times\ 100\% \tag{1}$$

Branch coverage measures adequacy of a test suite by checking if each branch of each control structure in GPU code has been executed by at least one thread.

Loop Boundary Coverage. In our survey of kernel bugs shown in Table 1, we found bugs related to loop boundary values and loop conditions were fairly common. For instance, bug #3 found in the `mcxcl` program allowed the loop index to access memory locations beyond the end of the array due to an erroneous loop condition. We assess adequacy of test executions with respect to loops by considering the following cases,

1. Loop body is not executed,
2. Loop body is executed exactly once,
3. Loop body is executed more than once
4. Loop boundary value is reached

$$Loop\ boundary\ coverage_{case\_i} = \frac{\#loops\ satisfying\ case\_i}{total\ \#loops} \times\ 100\% \tag{2}$$

where $case_i$ refers to one of the four loop execution cases listed above.

Barrier Coverage. Barrier divergence occurs when the number of threads within a work-group executing a barrier is not the same as the total number of threads in that work-group. Kernel behaviour with barrier divergence is undefined. Barrier

related bugs, missing barriers and unnecessary barriers, is a common class of GPU bugs according to our survey. We define barrier coverage as follows.

$$barrier\ coverage = \frac{\#covered\ barriers}{total\ \#barriers} \times\ 100\% \tag{3}$$

Barrier coverage measures adequacy of a test suite by checking if each barrier in GPU code is executed correctly. Correct execution of a barrier without barrier divergence, *covered barrier*, is when it is executed by *all* threads in any given work-group.

4.2 Fault Seeding

Mutation testing is known to be an effective means of estimating the fault finding effectiveness of test suites for CPU programs [9]. We generate mutations using traditional mutant operators, namely, arithmetic, relational, bitwise, logical and assignment operator types. In Table 1, bug fixes #3, #7 and #8 show that traditional arithmetic and relational operator mutations remain applicable to GPU programs. In addition, we define three mutations specifically for OpenCL kernels: barrier mutation, image access mutation and loop boundary mutation inspired by bug fixes #1 to #5.

The barrier mutation operator we define is deletion of an existing barrier function call, to reproduce bugs similar to #1 and #2 in Table 1. OpenCL provides 2D and 3D image data structures to facilitate access to images. Multi-dimensional arrays are not supported in OpenCL. Image structures are accessed using read and write functions that take the pixel coordinates in the image as parameter. We perform image access mutations for 2D or 3D coordinates by increasing or decreasing one of the coordinates or exchanging coordinates. Finally, we define loop boundary mutations as either (1) skipping the loop, (2) allowing n-1 iterations of the loop and (3) allowing n+1 iterations of the loop where n is the number of iterations when the loop boundary is reached. The mutant operators we use in this paper are summarised in Table 2.

Table 2. Summary of mutation operators

Type of Operator		Mutants
Arithmetic	Binary	+, −, *, /, %
	Unary	-(negation), ++, --
Relational		<, >, ==, <=, >=, ! =
Logical		&&, ||, !
Bitwise		&, |, ^, ~, <<, >>
Assignment		=, +=, −=, *=, /=, %=, <<=, >>=, &=, |=, ^=
Barrier		Delete barrier function call
Image coordinates		Change coordinates when accessing images
Loop boundary		Change the boundary value in loop condition check

4.3 Schedule Amplification

When a kernel execution is launched the GPU schedules work-groups on compute units in a certain order. Presently, there is no provision for determining this schedule or setting it in advance. The scheduler makes the decision on the fly subject to availability of compute units and readiness of work-groups for execution. The order in which work-groups are executed with the same test input can differ every time the kernel is executed. OpenCL specification has no execution model for inter work-group interactions and provides no guarantees on how work-groups are mapped to compute units. In our approach, we execute each test input over a set of schedules. In each schedule, we fix the work-group that should execute first. All other work-groups wait till it has finished execution. The work-group going first is picked so that we achieve a uniform distribution over the entire range of work-groups in the set of schedules. The order of execution for the remaining work-groups is left to the scheduler. For a test case, T over a kernel with G work-groups, we will generate N schedules, with $N < G$, such that a different work-group is executed first in each of the N schedules. The number of schedules, N, we generate is much lesser than the total number of schedules which is typically infeasible to check. The reason we only fix the first work-group in the schedule is because, most data races or deadlocks involve interactions between two work-groups. Fixing one of them and picking a different work-group each time, significantly reduces the search space of possible schedules. We cannot provide guarantees with this approach. However, with little extra cost we are able to check significantly more number of schedules than is currently possible. We believe this approach will be effective in revealing issues, if any, in inter work-group interactions.

To illustrate this, we consider a kernel co running on four work-groups. The CLTestCheck schedule amplifier will insert code on the host and GPU side, shown in Listings 1.1 and 1.2, to generate different work-group schedules.

Listing 1.1: Schedule OpenCL kernel (CPU-side)

```
// Generate a value in the range of [0,4)
int target_group = randint(4);
// Pass the value as a macro to GPU code
sprintf(clOptions,"-DTARGET_GROUP=%d", target_group);
```

Listing 1.2: Schedule OpenCL kernel (GPU-side)

```
if (my_group_id == TARGET_GROUP){
  // Original code here executed by target group
  A[(1 - buf) * 4 + tid] = A[buf * 4 + (tid + 1) % 4];
  atom_increase(num_threads_finishes);
} else {
  while (num_threads_finishes != group_size) continue;
  // Original code executed by other groups
  A[(1 - buf) * 4 + tid] = A[buf * 4 + (tid + 1) % 4];
}
```

In this example, before the GPU kernel is launched, the host side generates a random value in the range of available work-group ids. This value is the id of the selected work-group to be executed first and is passed to the kernel code using a

macro definition. On the kernel side, each thread determines if it belongs to the selected work-group. Threads in the selected work-group proceed with executing the kernel code while threads belonging to other work-groups wait. After the selected work-group completes execution, the remaining work-groups execute the original kernel in an order based on mapping to available compute units (occupancy bound execution model [22]). With different work-group schedules generated by the schedule amplifier, we were able to detect the presence of *inter* work-group data races using a *single* GPU platform. Betts et al. [2], on the other hand, focus on intra work-group data races on different GPU platforms.

4.4 Implementation

CLTestCheck is implemented using Clang LibTooling [12]. We instrument OpenCL kernel source code to measure coverage, generate mutations and multiple work-group schedules automatically. Our implementation is available at https://github.com/chao-peng/CLTestCheck.

Coverage Measurement. To record branches, loops and barriers executed within each kernel when running tests, we instrument the kernel code with data structures and statements recording the execution of these code structures. For each work-group, we introduce three local arrays, whose size is determined by the number of branches, loops and barriers accessible by threads in that work-group. To measure branch coverage, we add statements at the beginning of each then- and else-branch to record whether that branch is enabled. Similarly, statements to record the number of iterations of loops are added at the beginning of each loop body. At the end of the kernel, the information contained in the data structures is processed to compute coverage.

Fault Seeder and Mutant Execution. The CLTestCheck fault seeder generates mutants and executes them with each of the tests in the test suite to compute mutation score, as the fraction of mutants killed. The CLTestCheck fault seeder translates the target kernel source code into an intermediate form where all the applicable operators are replaced by a template string containing the original operator, its ID and type. The tool then generates mutants from this intermediate form. Once mutants are generated, the tool executes each of the mutant files and checks if the test suite kills the mutant. We term the mutant as killed if one of the following occurs: program crashes, deadlocks or produces a result different from the original kernel code.

Schedule Amplification. As mentioned earlier, we generate several schedules for each test execution by requiring a target work-group to execute the kernel code first and then allowing other work-groups to proceed. The target work-group is selected uniformly across the input space of work-group ids. To achieve coverage of this input space, we partition work-group ids into sets of 10 work-groups. Thus if we have N work-groups, we partition them into $N/10$ sets. The first set has work-group ids 0 to 9, the second set has ids 10 to 19 and so on. We then randomly pick a target work-group, W_t, from each of these sets to go first and generate a corresponding schedule of work-groups, $\{W_t, S_{N-1}\}$, where S_{N-1} refers to the schedule of remaining $N-1$ work-groups generated by the GPU execution model which is non-deterministic. For $N/10$ sets of work-groups, we will have $N/10$ schedules of the form $\{W_t, S_{N-1}\}$ (a W_t first schedule). The test input is executed using each of these $N/10$ W_t first schedules. Due to the

non-deterministic nature of S_{N-1}, we repeat the test execution with a chosen W_t first schedule 20 times. This will enable us to check if the execution model generates different S_{N-1} and evaluate executions with 20 such orderings.

5 Experiment

In our experiment, we evaluate the feasibility and effectiveness of the coverage metrics, fault seeder and work-group schedule amplifier proposed in Sect. 4 using OpenCL kernels from industry standard benchmark families and their associated test suites. We investigate the following questions:

Q1. Coverage Achieved: *What is the branch, barrier and loop coverage achieved by test suites over OpenCL kernels in our subject benchmarks?* To answer this question, we use our implementation to instrument and analyse kernel source code to record visited branches, barrier functions, loop iterations along with information on executing work-group and threads.

Q2. Fault Finding: *What is the mutation score of test suites associated with the subject programs?* For each benchmark, we generate all possible mutants by analysing the kernel source code and applying the mutation operators, discussed in Sect. 4, to eligible locations. We then assess number of mutants killed by the tests associated with each benchmark. To check if a mutant is killed, we compared execution results between the original program and mutant.

Q3. Deadlocks and Data Races: *Can the tests in the test suite give rise to unusual behaviour in the form of deadlocks or data races?* Deadlocks occur when two or more work-groups are waiting on each other for a resource. Inter work-group data races occur when test executions produce different outputs for different work-group schedules. For each test execution in each benchmark, we generate $20 * N/10$ different work-group schedules, where N is total number of work-groups for the kernel, and check if the outputs from the execution change based on work-group schedule.

Subject Programs. We used the following benchmarks for our experiments, 1. Nine scientific benchmarks with 23 OpenCL kernels from Parboil benchmark suite [23], 2. `scan` benchmark [20], with 3 kernels, that computes parallel prefix sum, 3. Five applications containing 13 kernels from Rodinia benchmark suite for heterogeneous computing, 4. 20 benchmarks from PolyBench with 43 kernels spanning linear algebra, data mining and stencil computations.

We ran our experiments on Intel CPU (i5-6500) and GPU (HD Graphics 530) using OpenCL SDK 2.0.

6 Results and Analysis

For each of the subject programs presented in Sect. 5, we ran the associated test suites and report results in terms of coverage achieved, fault finding and overhead incurred with CLTestCheck framework. We executed the test suites 20 times for each measurement. Our results in the context of the questions in Sect. 5 is presented below.

6.1 Coverage Achieved

Branch and Loop coverage (with 0, exactly 1 and >1 iterations) for each of the subject programs in the three benchmark suites[2] is shown in the plots in Fig. 1. The first row shows branch coverage, the second loop coverage. Mutation score and surviving mutation types shown in the last two rows of Fig. 1 is discussed in the next Sect. 6.2.

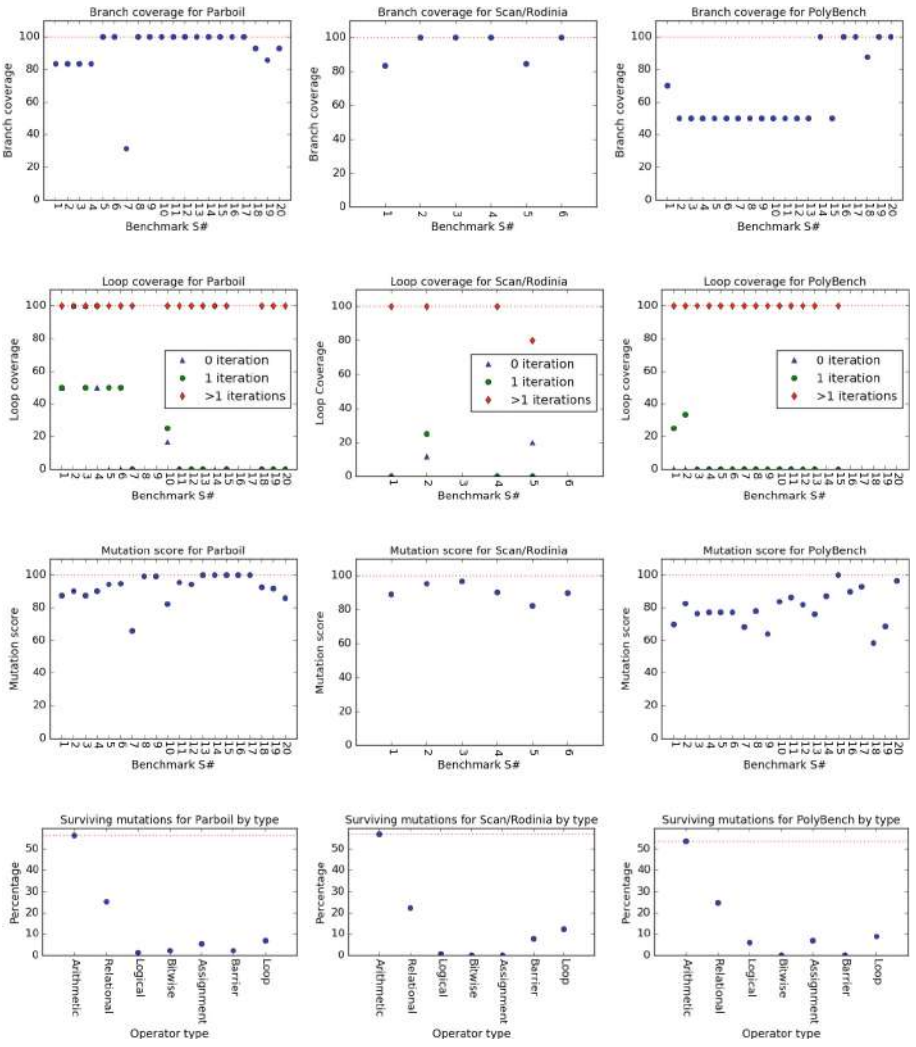


Fig. 1. Coverage achieved - Branch and Loop, mutation score and percentage of surviving mutations by type for each subject program in the 3 benchmark suites.

[2] 20 applications in Parboil counting different test suites separately, 6 in Scan/Rodinia, and 20 in PolyBench.

Barrier Coverage is not shown in the plots since for all, except one, applications with barriers, the associated test suites achieved 100% barrier coverage. The only subject program with less than 100% barrier coverage was `scan`, which had 87.5% barrier coverage. The uncovered barrier is in a loop whose condition does not allow some threads to enter the loop, resulting in barrier divergence between threads. We find that less than 100% barrier coverage is a useful indicator of barrier divergence in code.

Branch Coverage. For most subject programs in Parboil and Scan/Rodinia, test suites achieve high branch coverage (>83%). The `histo` benchmark is an outlier with a low branch coverage of 31.6%. Its kernel function, `histo_main`, contains 20 branches in a code block handling an exception condition (overflow). The test suite provided with `histo` does not raise the overflow exception, and as a result, these branches are never executed. We found uncovered branches in other applications, with >80% coverage, in Parboil and Scan/Rodinia to also result from exception handing code that is not exercised by the associated test data.

Branch coverage achieved for 13 of the 20 applications in PolyBench is at 50%. This is very low compared with other benchmark suites. Upon investigating the kernel code, we found that all the uncovered branches reside within a condition check for out of range array index. Tests associated with a majority of the applications did not check out of range array index access, resulting in low branch coverage.

Loop Coverage. Test suites for nearly all applications (with loops) execute loops more than once. Thus, coverage for >1 iterations is 100% for all but one of the applications, `srad` in Rodinia suite, that has 80%. The uncovered loop in `srad` is in an uncovered then-branch that checks exception conditions. We also checked if the boundary value in loop conditions is reached when >1 iterations is covered by test executions. We found `pathfinder` in Rodinia to be the only application to have full coverage for >1 iterations but not reach the boundary value. The unusual scenario in `pathfinder` is because one of the loops is exited using a break statement.

We find that test suites for most applications are unable to achieve any loop coverage for 0 and exactly 1 iteration. The boundary condition for most loops is based on the size of the work-groups which is typically much greater than 1. As a result, test suites have been unable skip the loop or execute it exactly once. The only exceptions were applications in the Parboil suite - `bfs, cutcp, mri-gridding, spmv`, and two applications in Rodinia - `lud, srad`, that have boundary values dependent on variables that maybe set to 0 or 1.

Overhead. For each benchmark and associated test suite, we assessed overhead introduced by our approach. We compared time needed for executing the benchmark with instrumentation and additional data structures that we introduced for coverage measurement against the original unchanged benchmark. Overhead varied greatly across benchmarks and test suites. Overhead for Parboil and Rodinia benchmarks was in the range of 2% to 118%. Overhead was lower for benchmarks that took longer to execute as the additional execution time from instrumentation is a smaller fraction of the overall time. Overhead for most programs in PolyBench ranges from 2% to 70%, which is similar to Parboil and Rodinia benchmarks. The overhead for `lu`, `fdtd-2d` and `jacobi-2d-imper` programs are >100%. The code for kernel computations in these benchmarks is

small with fast execution. Consequently, the relative increase in code size and execution time after instrumentation with CLTestCheck is high.

6.2 Fault Finding

Fault finding for the subject programs is assessed using the mutants we generate with the fault seeder, described in Sect. 4. The mutation score, percentage of mutants killed, is used to estimate fault finding capability of test suites associated with the subject programs. Each test suite associated with a benchmark is run 20 times to determine the killed mutants. A mutant is considered killed if the test suite generates different outputs on the mutant than the original program in *all* 20 repeated runs of the test suite. In addition to killed mutants, we also report results on "Undecided Mutants", that refers to mutants that are killed in at least one of the executions of the test suite, but *not all* 20 repeated executions. Changes in GPU thread scheduling between runs causes this uncertainty. We do not count the undecided mutants towards killed mutants in the mutation score. Mutation score for all subject programs in each benchmark suite is shown in the third row of plots in Fig. 1.

Mutation Score. In general, we find that test suites for subject programs achieving high branch, barrier and loop coverage also have high mutation score. For instance, for `spmv` and `stencil`, their test suites achieving 100% coverage, also achieved 100% mutation score. An instance of a program that does not follow this trend is `mri-gridding` that has 100% branch, barrier, and loop (>1 iterations) coverage but only 82% mutation score. On analysing the survived mutants, we found a significant fraction (160 out of 232) were arithmetic operator mutations within a function named *kernel_value* that contained variables defining a fourteenth-order polynomial and a cubic polynomial. Effect of mutations on the polynomials did not propagate to the output of the benchmark with the given test suite. The `histo` program with low branch coverage, 100% barrier and loop coverage has 65.9% mutation score. Nearly two thirds of the branches in `histo` cannot be reached by the input data, as a result, all the mutations in the untouched branches is not killed, resulting in a low mutation score. A few of the programs in PolyBench have mutation scores that are between 60–70%. In these programs, most surviving mutations are arithmetic operator mutations.

As seen in the last row of Fig. 1 showing surviving mutations by operator type, arithmetic operators are the dominant surviving mutations in all three benchmark suites. Control flow adequate tests can kill arithmetic operator mutations only if they propagate to a control condition or the output. Data flow coverage may be better suited for estimating these mutations. Around 20% of relational operator mutations also survive in our evaluation. Most of the surviving relational operator mutations made slight changes to operators, such as < to <=, or > to >= and vice versa. The test suites provided with the benchmarks missed such boundary mutations.

Undecided mutants occur during executions of 9, out of the 46 subject programs and test suites across all three benchmark suites. Number of undecided mutants during the 9 executions is generally small (<= 5). The only exception is `tpacf` in the Parboil benchmark suite, that resulted in 18 undecided mutants when executing one of its test suite. Undecided mutants point to non-deterministic behaviour in the kernel, that is dependent on GPU thread execu-

tion model. A large number of undecided mutants is alarming and developers should examine kernel code more closely to ensure that the behaviour observed is as intended.

Barriers were not used in all benchmarks. Only 5 out of the 9 benchmarks in Parboil, and 4 of the 6 in Scan/Rodinia had barriers. PolyBench programs did not use any barriers. Mutations removed barrier function calls in these benchmarks and we ecorded the number of mutants killed by test suites. Percentage of killed barrier mutations is generally low across all benchmarks with barriers. For instance, removing 2 out of 3 barriers in the `histo` program in Parboil, and removing all barriers in the `cutcp` program had no effect on outputs of the respective program executions. This may either mean that the test suites are inadequate with respect to the barrier mutations or it could be an indication that these barriers are superfluous with respect to program outputs, and the need for synchronisation should be further justified. For the programs in our experiment, we found barriers, whose mutations survived, to be unnecessary.

Coverage versus Mutation Score. The plots in Fig. 1 illustrate total mutation score over all types of mutations for each subject program and test suite. We also compute mutation scores specifically for branches, barriers, and loops using mutations relevant to them. We do this to compare against branch, barrier and loop coverage achieved for each of the subject programs. We found that mutation score for branches closely follows branch coverage for most subject programs. Outliers include `adi`, `nn`, `convolution-2d` and `convolution-3d`. Mutations that change $<$ to $<=$ are not killed in these kernels; these comprise one third of all branch mutations.

Mutation score for barriers is quite different from barrier coverage. This is because test suites are able to execute the barriers and achieve coverage. However, they are unable to produce different outputs when the barriers are removed. This may be a problem with the superfluous manner in which barriers are used in these programs.

Loop coverage with >1 iterations is 100% for all but one subject program (`srad` in Rodinia). Mutation score for loops on the other hand is variable. In general, tests achieving loop coverage are unable to reveal loop boundary mutations. Histo and srad are worth noting with high loop coverage but low loop mutation scores. We find that mutations to the loop boundary value in these two benchmarks survive, which implies that access to loop indices outside the boundary go unchecked in these programs. These unsafe values of loop indices should be disallowed in these kernels and loop boundary mutations in our fault seeder help reveal them.

6.3 Schedule Amplification: Deadlocks and Data Races

Kernel Deadlocks: When we used the CLTestCheck schedule amplifier on our benchmarks, we found kernel executions deadlock when the work-group ID selected to go first exceeds the number of available compute units. As there are no guarantees on how work-groups are mapped to compute units, we allow work-group IDs exceeding number of compute units to go first in some test executions using our schedule amplifier. However, it appears that the GPU makes unstated assumptions on what work-group IDs are allowed to go first. As noted by Sorenson et al. [22], "execution of large number of work-groups is in any *occupancy*

bound fashion, by delaying the scheduling of some work-groups until others have executed to completion". They observed deadlocks in kernel execution due to inter work-group barriers. However, in the benchmarks in our evaluation, there is no explicit inter work-group barrier. It may be the case that developers made implicit assumptions on inter work-group barriers using the occupancy bound model and our schedule amplification approach violates this assumption. Nevertheless, our finding exposes the need for an inter work-group execution model that explicitly states the details and assumptions related to mapping of work-groups to compute units for a given kernel on a given GPU platform.

Inter Work-group Data Races: We were able to reveal a data race in the `spmv` application from the Parboil benchmark suite. We found that when work-groups 0 or 1 are chosen to go first in our schedules, the kernels execution always produces the same result. However, when we pick other work-group ids to go first, the test output is not consistent. Among twenty executions for each schedule, the frequency of producing correct output varies from 45% to 70%.

We observe similar behaviour in the `tpacf` application in Parboil when we delete the last barrier function call in the kernel. The kernel execution produces consistent outputs when we pick work-group 0 or 1 to go first. When we pick other work-groups to go first using our schedule amplifier, the kernel execution results are non-deterministic.

We observe no unusual behaviour in any of the PolyBench programs. These programs split the computation into multiple kernels and the CPU program launches GPU kernels one by one. The transfer of control from the GPU to the CPU between kernels acts like a barrier as the CPU will wait until a kernel finishes before launching the next kernel. In addition, care has been taken in the kernel code to ensure threads do not access the same memory location. As a result, we observe no data races in PolyBench with our schedule amplifier.

7 Conclusion

We have presented the CLTestCheck framework for measuring test effectiveness over OpenCL kernels with capabilities to measure code coverage, fault seeding and mutation score measurement, and finally amplify the execution of a test input with multiple work-group schedules to check inter work-group interactions. Our empirical evaluation of CLTestCheck capabilities with 82 publicly available kernels revealed the following,

1. The schedule amplifier was able to detect deadlocks and inter work-group data races in Parboil benchmarks when higher work-group ids were forced to execute first. This finding emphasizes the need for transparency and clearly stated assumptions on how work-groups are mapped to compute units.
2. Barrier coverage served as a useful measure in identifying barrier divergence in benchmarks (`scan`).
3. Branch coverage pointed to inadequacies in existing test suites and found test inputs for exercising error handling code were missing.
4. Across all benchmark suites, we found arithmetic operator and relational operator mutations that changed $<$ to $<=$, $>$ to $>=$ or vice versa were hard to kill. More rigorous test suites to handle these mutations are needed.

5. The use of barrier mutations revealed several instances of unnecessary barrier use. Barrier usage and its implications is not well understood by developers. Barrier mutations can help reveal incorrect barrier uses.
6. Loop boundary mutations helped reveal unsafe accesses to loop indices outside the loop boundary.

In sum, the CLTestCheck framework is an automated, effective and useful tool that will help developers assess how well OpenCL kernels have been tested, kernel regions that require further testing, uncover bugs with respect to work-group schedules. In the future, we plan to add further metrics, like data flow coverage with work-group schedule, to strengthen test adequacy measurement.

References

1. AMD Inc. and Vratis Ltd.: clSPARSE: a software library containing sparse functions written in OpenCL (2016). https://github.com/clMathLibraries/clSPARSE
2. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 113–132. ACM, New York (2012)
3. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic testing of OpenCL code. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 203–218. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34188-5_18
4. Emonet, R.: Experiments on Gaussian pyramid implemented using OpenCL (2010). https://github.com/twitwi/ClGaussianPyramid
5. Fang, Q.: Monte Carlo eXtreme for OpenCL (MCXCL) (2017). https://github.com/fangq/mcxcl
6. Gay, G., Rajan, A., Staats, M., Whalen, M., Heimdahl, M.P.: The effect of program and model structure on the effectiveness of MC/DC test adequacy coverage. ACM Trans. Softw. Eng. Methodol. (TOSEM) **25**(3), 25 (2016)
7. Group, K.O.W.: The OpenCL specification version 2.2 (2017)
8. Horton, T.: Cinematic particle effects with OpenCL (2010). https://github.com/hortont424/particles
9. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
10. Kim, H.H.: Winograd-based convolution implementation in OpenCL (2017). https://github.com/csehydrogen/Winograd-OpenCL
11. Kloppenborg, B., Baron, F.: LibOI: the OpenCL interferometry library (2012). https://github.com/bkloppenborg/liboi
12. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, p. 75. IEEE Computer Society (2004)
13. Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying GPU kernels by test amplification. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, pp. 383–394. ACM, New York (2012)
14. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 187–196. ACM (2010)

15. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012, pp. 215–224. ACM, New York (2012)
16. Lidbury, C., Lascu, A., Chong, N., Donaldson, A.F.: Many-core compiler fuzzing. ACM SIGPLAN Not. **50**(6), 65–76 (2015)
17. NVIDIA Corporation: CUDA zone, September 2017. https://developer.nvidia.com/cuda-zone
18. Rajan, A., Heimdahl, M.P.: Coverage metrics for requirements-based testing. University of Minnesota (2009)
19. Rajan, A., Sharma, S., Schrammel, P., Kroening, D.: Accelerated test execution using GPUs. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 97–102. ACM (2014)
20. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Graphics Hardware, vol. 2007, pp. 97–106 (2007)
21. Sorensen, T., Donaldson, A.F.: The Hitchhiker's guide to cross-platform OpenCL application development. In: Proceedings of the 4th International Workshop on OpenCL, p. 2. ACM (2016)
22. Sorensen, T., Donaldson, A.F., Batty, M., Gopalakrishnan, G., Rakamarić, Z.: Portable inter-workgroup barrier synchronisation for GPUs. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, pp. 39–58. ACM, New York (2016)
23. Stratton, J.A., et al.: Parboil: a revised benchmark suite for scientific and commercial throughput computing. Center Reliable High-Perform. Comput. **127** (2012)
24. Xiao, S., Feng, W.C.: Inter-block GPU communication via fast barrier synchronization. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–12. IEEE (2010)
25. Zheng, M., Ravi, V., Qin, F., Agrawal, G.: GRace: a low-overhead mechanism for detecting data races in GPU programs. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, vol. 46, pp. 135–146, August 2011
26. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GMRace: detecting data races in GPU programs via a low-overhead scheme. IEEE Trans. Parallel Distrib. Syst. **25**(1), 104–115 (2014)

Implementing SOS with Active Objects: A Case Study of a Multicore Memory System

Nikolaos Bezirgiannis[1], Frank de Boer[1], Einar Broch Johnsen[2(✉)], Ka I Pun[2,3], and S. Lizeth Tapia Tarifa[2]

[1] CWI, Amsterdam, The Netherlands
{n.bezirgiannis,f.s.de.boer}@cwi.nl
[2] Department of Informatics, University of Oslo, Oslo, Norway
{einarj,violet,sltarifa}@ifi.uio.no
[3] Western Norway University of Applied Sciences, Bergen, Norway

Abstract. This paper describes the development of a parallel simulator of a multicore memory system from a model formalized as a structural operational semantics (SOS). Our implementation uses the Abstract Behavioral Specification (ABS) language, an executable, active object modelling language with a formal semantics, targeting distributed systems. We develop general design patterns in ABS for implementing SOS, and describe their application to the SOS model of multicore memory systems. We show how these patterns allow a formal correctness proof that the implementation simulates the formal operational model and discuss further parallelization and fairness of the simulator.

1 Introduction

Structural operational semantics (SOS) [1], introduced by Plotkin in 1981, describes system behavior as transition relations in a syntax-oriented, compositional way, using inference rules for local transitions and their composition. Process synchronization in SOS rules is expressed abstractly using, e.g., assertions over system states and reachability conditions over transition relations as premises, and label synchronization for parallel transitions. This high level of abstraction greatly simplifies the verification of system properties, but not the simulation of system behavior as execution quickly becomes a reachability problem with a lot of backtracking. In this paper, we study how to implement a parallel simulator with a formal correctness proof from a SOS model, in terms of a case study of a multicore memory system. Such a correctness proof requires that the implementation language is also defined formally by an operational semantics.

Supported by *SIRIUS: Centre for Scalable Data Access* (www.sirius-labs.no) and *ADAPt: Exploiting Abstract Data-Access Patterns for Better Data Locality in Parallel Processing* (www.mn.uio.no/ifi/english/research/projects/adapt/).

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 332–350, 2019.
https://doi.org/10.1007/978-3-030-16722-6_20

A major challenge in software engineering is the exploitation of the computational power of multicore (and manycore) architectures. One important aspect of this challenge is the memory systems of these architectures. These memory systems generally use caches to avoid bottlenecks in data access from main memory, but caches introduce data duplication and require protocols to ensure coherence. Although data duplication is usually not visible to the programmer, the way a program interacts with these copies largely affects performance by moving data around to maintain coherence. To develop, test and optimize software for multicore architectures, we need correct, executable models of the underlying memory systems. A SOS model of multicore memory systems with correctness proofs for cache coherency has been described in [2], together with a prototype implementation in the rewriting logic system Maude [3]. However, this fairly direct implementation of the SOS model is not well suited to simulate large systems.

This paper considers an implementation of the SOS model in ABS [4], a language tailored to the description of distributed systems based on active objects [5]. ABS is formally defined by an operational semantics and supports parallel execution on backends in Erlang, Haskell, and Java. The following features of ABS allow a high-level, coarse-grained view of the execution of different method invocations by different active objects: encapsulation of local state in active objects, communication using asynchronous method calls and futures, and cooperative scheduling of the method invocations of an active object. Our case study fully exploits these features and the resulting abstractions to correctly implement the complex process synchronization of the original SOS model.

The main contributions of this paper are as follows:

- We provide general design patterns in ABS for implementing structural operational semantics with active objects, and apply these patterns to the implementation in ABS of a structural operational semantics of multicore memory systems.
- We show how these patterns allow a formal correctness proof of this implementation by means of a simulation relation between the formal operational semantics of the ABS implementation and the operational model of multicore memory systems.
- We discuss how these ABS design patterns can be used to further parallelize the implementation while preserving correctness.
- Finally, we show how the ABS modeling concepts of symbolic time and virtual resources can be used to obtain a parallel implementation of the SOS model which abstractly ensures fairness between the progress of different parallel components, independently of the number of cores that are used in the simulation.

2 An Abstract Model of a Multicore Memory System

Design decisions for a program running on top of a multicore memory systems can be explored using simulators based on abstract models. Bijo et al. [2,6] developed a model which takes as input tasks (expressed as data access) to

be executed, the corresponding data layout in main memory (indicating where data is allocated), and a parallel architecture consisting of cores with private multi-level caches and shared memory (see Fig. 1). Additionally, the model is configurable in the number of cores, the number and size of caches, and the associativity and replacement policy. Memory is organized in blocks which move between caches and main memory. For simplicity, the model assumes that the size of cache lines and memory blocks in main memory coincide, abstracts from the data content of memory blocks, and transfers memory blocks from the caches of one core to the caches of another core via main memory.

Tasks from the program are scheduled for execution from a shared task pool. Task execution on a core requires memory blocks to be transferred from main memory to the closest cache. Each cache has a pool of fetch/flush instructions to move blocks among caches and between caches and main memory. Consistency between multiple copies of a memory block is ensured using the standard cache coherence protocol MSI (e.g., [7]), with which a cache line is either modified, shared or invalid. A *modified* cache line has the most recent value of the memory block, therefore all other copies are *invalid* (including the one in main memory). A *shared* cache line indicates that all copies of the block are consistent. The protocol's messages are broadcast to the cores. The details of the broadcast (e.g., on a mesh or a ring) can be abstracted into an *abstract communication medium*. Following standard nomenclature, *Rd* messages request *read* access and *RdX* messages *read exclusive* access to a memory block. The latter invalidates other copies of the same block in other caches to provide write access.

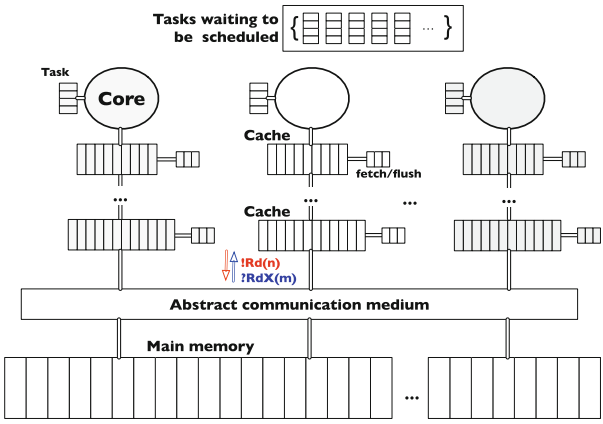


Fig. 1. Abstract model of a multicore memory system.

To access data from a block n, a core looks for n in its local caches. If n is not found in shared or modified state, a *read request* $!Rd(n)$ is broadcast to the other cores and to main memory. The cache can *fetch* the block when it is available in main memory. Eviction is required if the cache is full. Writing to block n requires n to be in shared or modified state in the local cache; if it is in shared state, an *invalidation request* $!RdX(n)$ is broadcast to obtain exclusive access. If a cache with block n in modified state receives a read request $?Rd(n)$, it *flushes* the block to main memory; if a cache with block n in shared state receives an invalidation request $?RdX(n)$, the cache line will be *invalidated*; the requests are discarded otherwise. Read and invalidation requests are broadcast instantaneously in the abstract model, reflecting that signalling on the communication medium is order of magnitude faster than moving data to or from main memory.

Syntactic categories.	*Definitions.*	
$cid \in CoreId$	$cf \in Config$	$::= M \circ \overline{dap} \circ \overline{Ca} \circ \overline{CR}$
$caid \in CacheId$	$CR \in Core$	$::= cid \bullet rst$
$n \in Address$	$Ca \in Cache$	$::= caid \bullet M \bullet dst$
$r \in Ref$	$st \in Status$	$::= \{mo, sh, inv\}$
	$dap \in AccessPtns$	$::= \varepsilon \mid dap; dap \mid \texttt{read}(r) \mid \texttt{write}(r) \mid \texttt{commit}(r)$
		$\mid \texttt{commit} \mid dap \sqcap dap \mid dap^* \mid \texttt{skip} \mid \texttt{spawn}(dap)$
	$rst \in RunLang$	$::= dap \mid rst; rst \mid \texttt{readBl}(r) \mid \texttt{writeBl}(r)$
	$dst \in DataLang$	$::= \varepsilon \mid \overline{dst} \mid \texttt{fetch}(n) \mid \texttt{flush}(n) \mid \texttt{fetchBl}(n) \mid \texttt{flush}$

Fig. 2. Syntax of runtime configurations, where over-bar denotes sets (e.g., $\overline{CR}$).

2.1 Formalization of the Multicore Memory System as an SOS Model

An operational meaning for the abstract model described above has be defined using structural operational semantics (SOS) [1] with labeled transitions to model broadcast in the abstract communication medium. The resulting formalization [2,6] is shown to guarantee standard correctness properties for data consistency and cache coherence from the literature [8,9], including the preservation of program order in each core, the absence of data races, and no access to stale data. We briefly outline the main aspects of the formal model. The runtime syntax is given in Fig. 2. A configuration cf consists of main memory M, cores $\overline{CR}$, caches $\overline{Ca}$, and tasks $\overline{dap}$ to be scheduled. (We syntactically abuse set operations for multisets, including union $\cup$ and subtraction $\setminus$.) A core $cid \bullet rst$ with identifier cid executes runtime statements rst. A cache with identifier $caid$ has a local cache memory M and data instructions dst. We assume that $caid$ encodes the cid of the core to which the cache belongs and its level in the cache hierarchy. We denote by $Status \cup \{\bot\}$ the extension of the set of status tags with the undefined value $\bot$. Thus, a memory $M : Address \to Status \cup \{\bot\}$ maps addresses n to either a status tags $Status$ or to $\bot$ if the memory block with address n is not found in M.

Data access patterns dap model tasks consisting of $\texttt{read}(r)$ and $\texttt{write}(r)$ operations to references r and control flow operations for sequential composition $dap_1; dap_2$, non-deterministic choice $dap_1 \sqcap dap_2$, repetition dap^*, task creation $\texttt{spawn}(dap)$, and **commit** which flushes the entire cache after task execution. The empty access pattern is denoted ε. Cores execute *runtime statements* rst, which extend dap with $\texttt{readBl}(r)$ and $\texttt{writeBl}(r)$ to block execution while waiting for data. Caches execute *data instructions* dst to fetch and flush the memory block with address n, here $\texttt{fetchBl}(n)$ blocks execution while waiting for data, and **flush** flushes the entire cache.

The *abstract communication medium* allows messages from one cache to be transmitted to the other caches and to main memory in a parallel instantaneous broadcast. Communication in the abstract communication medium is formalized in terms of label matching on transitions. The formal syntax for this label mechanism is as follows:

$$S ::= !Rd(n) \mid !RdX(n) \qquad R ::= ?Rd(n) \mid ?RdX(n)$$

Here, for any address n, a request of the form $!Rd(n)$ or $!RdX(n)$ is sent by one node and its dual of the form $dual(!Rd(n)) = ?Rd(n)$ or $dual(!RdX(n)) = ?RdX(n)$ is broadcast to the rest of nodes and main memory. The syntax of the model is further detailed in [2,6].

2.2 Local and Global SOS Rules

The semantics is divided into local and global rules. Local rules capture interaction inside a node containing a core and the hierarchy of caches. Global rules capture synchronization and coordination between different nodes and main memory. In an *initial* configuration cf_0, all blocks in main memory M have status *sh*, all cores are idle, all caches are empty, and the task pool in $\overline{dap}$ has a single task representing the main block of a program. Let $cf \xrightarrow{*} cf'$ denote an execution starting from cf and reaching cf' by applying global transition rules, which in turn apply local transition rules for each core and its cache hierarchy. In the rules, let the auxiliary function $addr(r)$ return the address n of the block containing reference r, $cid(caid)$ the identity of the core associated with cache $caid$, $lid(caid)$ the cache level of $caid$, and $status(M, n)$ the status of block n in map M. Let the predicate $first(caid)$ hold when $caid$ is the first level and $last(caid)$ when $caid$ is the last level cache. Note that unlabelled transitions $\rightarrow$ can be executed asynchronously, while labelled transitions $\xrightarrow{S}$ require synchronization between all the nodes and main memory (see Figs. 3 and 4). We discuss some representative rules for local and global level of the SOS model. The full SOS formalization can be found in [6].

Local semantics. The first rules of Fig. 3 involve a core and its first level cache. In PrRd_1, reading reference r succeeds if the block containing r is available. Otherwise, in PrRd_2 a **fetch**(n) instruction is added to the data instructions *dst* of the first level cache and further execution of the core is blocked by **readBl**(r). Writing to r only succeeds if the associated memory block has *mo* status in the first level cache. If the cache line is shared, the core broadcasts a $!RdX(n)$ request to acquire exclusive access, where the broadcast appears as a label on the transition in PrWr_2. Otherwise, the block must be fetched from main memory in PrWr_3 and **writeBl**(r) blocks execution.

For the remaining rules of Fig. 3, LC-Hit_1 and LC-Miss_1 capture interactions between adjacent levels of caches, and LCC-Miss_1 local state change in a cache line. If cache $caid_i$ needs a block n that is *sh* or *mo* in the next level cache, the address where block n should be placed is decided by a function $select(M_i, n)$ which reflects the cache associativity and the replacement policy. If eviction is needed, block n in $caid_j$ will be swapped with the selected block in $caid_i$ in LC-Hit_1. LC-Miss_1 shows how **fetch**(n)-instructions propagate to lower cache levels: **fetch**(n) is replaced by **fetchBl**(n) in $caid_i$ and added to the data instructions in $caid_j$. If the block cannot be found in any local cache, we have a *cache miss*: Execution is blocked by **fetchBl**(n) and a read request $!Rd(n)$ is broadcast, represented by the label in LLC-Miss_1.

$$\text{(PrRD}_1\text{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{sh, mo\}}{(caid \bullet M \bullet \overline{dst}) \circ (c \bullet \texttt{read}(r); rst) \rightarrow (caid \bullet M \bullet \overline{dst}) \circ (c \bullet rst)}$$

$$\text{(PrRD}_2\text{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{inv, \bot\}}{(caid \bullet M \bullet \overline{dst}) \circ (c \bullet \texttt{read}(r); rst) \rightarrow (caid \bullet M[n \mapsto \bot] \bullet \overline{dst} \cup \{\texttt{fetch}(n)\}) \circ (c \bullet \texttt{readBl}(r); rst)}$$

$$\text{(PrWR}_2\text{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) = sh}{(caid \bullet M \bullet \overline{dst}) \circ (c \bullet \texttt{write}(r); rst) \xrightarrow{!RdX(n)} (caid \bullet M[n \mapsto mo] \bullet \overline{dst}) \circ (c \bullet rst)}$$

$$\text{(PrWR}_3\text{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{inv, \bot\}}{(caid \bullet M \bullet \overline{dst}) \circ (c \bullet \texttt{write}(r); rst) \rightarrow (caid \bullet M[n \mapsto \bot] \bullet \overline{dst} \cup \{\texttt{fetch}(n)\}) \circ (c \bullet \texttt{writeBl}(r); rst)}$$

$$\text{(LC-Hit}_1\text{)}\quad \frac{status(M_i,n_i) = s_i \quad status(M_j,n) = s_j \quad s_j \in \{sh, mo\} \quad lid(caid_j) = lid(caid_i) + 1 \quad cid(caid_i) = cid(caid_j) \quad select(M_i,n) = n_i}{(caid_i \bullet M_i \bullet \overline{dst_i} \cup \{\texttt{fetch}(n)\}) \circ (caid_j \bullet M_j \bullet \overline{dst_j}) \rightarrow (caid_i \bullet M_i[n_i \mapsto \bot, n \mapsto s_j] \bullet \overline{dst_i}) \circ (caid_j \bullet M_j[n \mapsto \bot, n_i \mapsto s_i] \bullet \overline{dst_j})}$$

$$\text{(LC-Miss}_1\text{)}\quad \frac{lid(caid_j) = lid(caid_i) + 1 \quad cid(caid_i) = cid(caid_j) \quad status(M_j,n) \in \{inv, \bot\}}{(caid_i \bullet M_i \bullet \overline{dst_i} \cup \{\texttt{fetch}(n)\}) \circ (caid_j \bullet M_j \bullet \overline{dst_j}) \rightarrow (caid_i \bullet M_i \bullet \overline{dst_i} \cup \{\texttt{fetchBl}(n)\}) \circ (caid_j \bullet M_j[n \mapsto \bot] \bullet \overline{dst_j} \cup \{\texttt{fetch}(n)\})}$$

$$\text{(LLC-Miss}_1\text{)}\quad \frac{last(caid) = true \quad status(M,n) \in \{inv, \bot\}}{(caid \bullet M \bullet \overline{dst} \cup \{\texttt{fetch}(n)\}) \xrightarrow{!Rd(n)} (caid \bullet M[n \mapsto \bot] \bullet \overline{dst} \cup \{\texttt{fetchBl}(n)\})}$$

Fig. 3. Local transition rules.

$$\text{(Synch}_1\text{)}\quad \frac{S \neq \emptyset \quad R = dual(S) \quad M \xrightarrow{R} M' \quad \overline{Ca} \circ \overline{CR} \xrightarrow{S} \overline{Ca'} \circ \overline{CR'}}{M \circ \overline{dap} \circ \overline{Ca} \circ \overline{CR} \rightarrow M' \circ \overline{dap} \circ \overline{Ca'} \circ \overline{CR'}}$$

$$\text{(Synch}_2\text{)}\quad \frac{\overline{CR} = \{CR_1\} \uplus \overline{CR_2} \quad \overline{Ca} = \overline{Ca_1} \uplus \overline{Ca_2} \quad belongs(\overline{Ca_1}, \{CR_1\}) \quad belongs(\overline{Ca_2}, \overline{CR_2}) \quad R = dual(S) \quad \overline{Ca_1} \circ CR_1 \xrightarrow{S} \overline{Ca_1'} \circ CR_1' \quad \overline{Ca_2} \xrightarrow{R} \overline{Ca_2'} \quad \overline{CR'} = \{CR_1'\} \cup \overline{CR_2} \quad \overline{Ca'} = \overline{Ca_1'} \cup \overline{Ca_2'}}{\overline{Ca} \circ \overline{CR} \xrightarrow{S} \overline{Ca'} \circ \overline{CR'}}$$

$$\text{(Asynch)}\quad \frac{\overline{CR} = \overline{CR_1} \uplus \overline{CR_2} \uplus \overline{CR_3} \quad \overline{Ca} = \overline{Ca_1} \uplus \overline{Ca_2} \uplus \overline{Ca_3} \uplus \overline{Ca_4} \quad belongs(\overline{Ca_3}, \overline{CR_3}) \quad M \circ \overline{Ca_1} \rightarrow M' \circ \overline{Ca_1'} \quad \overline{Ca_2} \rightarrow \overline{Ca_2'} \quad \overline{dap} \circ \overline{CR_2} \rightarrow \overline{dap'} \circ \overline{CR_2'} \quad \overline{Ca_3} \circ \overline{CR_3} \rightarrow \overline{Ca_3'} \circ \overline{CR_3'} \quad \overline{CR'} = \overline{CR_1} \cup \overline{CR_2'} \cup \overline{CR_3'} \quad \overline{Ca'} = \overline{Ca_1'} \cup \overline{Ca_2'} \cup \overline{Ca_3'} \cup \overline{Ca_4}}{M \circ \overline{dap} \circ \overline{Ca} \circ \overline{CR} \rightarrow M' \circ \overline{dap'} \circ \overline{Ca'} \circ \overline{CR'}}$$

Fig. 4. Global transition rules.

Global semantics. The global rules synchronize the cache hierarchies of different cores and main memory, and ensures coherence. Selected global rules are given in Fig. 4. Rule Synch$_1$ captures a global step with synchronization on a label S, which can be either $!Rd(n)$ or $!RdX(n)$. The request will be broadcast to other caches. To maintain data consistency, these caches must process the requests at the same time. The receiving label R is the *dual* of S. For synchronization, the

transition is decomposed into a premise for main memory with label R and another premise for the caches with label S. Rule SYNCH_2 distributes the receiving label to caches $\overline{Ca}_2$, which do not belong to the cache hierarchy of the sender core CR_1. The predicate $belongs(\overline{Ca}, \overline{CR})$ expresses that any cache in $\overline{Ca}$ belongs to exactly one core in $\overline{CR}$. Rule ASYNCH captures parallel transitions without label. These transitions can be local to individual nodes and caches, parallel memory accesses, or the parallel spawning and scheduling of new tasks.

3 The ABS Model of the Multicore Memory System

In this section we outline the translation of the formal model into an executable object-oriented model using the ABS modeling language. We first briefly introduce the language and later explain the structural and behavioural correspondence between these two models, with a focus on the main challenges.

3.1 The ABS Language

ABS is a modeling language for designing, verifying, and executing concurrent software [4]. The language combines the syntax and object-oriented style of Java with the Actor model of concurrency [10] into active objects which decouple communication and synchronization using asynchronous method calls, futures and cooperative scheduling [5]. Although only one thread of control can execute in an active object at any time, cooperative scheduling allows different threads to interleave at explicitly declared points in the code. Access to an object's fields is encapsulated, so any non-local (outside of the object) read or write to fields must happen explicitly via asynchronous method calls so as to mitigate race-conditions or the need for mutual exclusion (locks).

We explain the basic mechanism of asynchronous method calls and cooperative scheduling in ABS by the simple code example of a class Bus. First, the execution of a statement res = **await** o!m(args) consists of storing a message m(args) corresponding to the asynchronous call to the message pool of the callee object o. This **await** statement *releases the control* of the caller until the return value of that method has been received. Releasing the control means that the caller can execute other messages from its own message pool in the meantime. ABS supports the shorthand o.m(args) to make an asynchronous call f=o!m(args) followed by the operation f.**get** which *blocks* the caller object (does not release control) until the future f has received the return value from the call. As a special case the statement **this**.m(args) models a self-call, which corresponds to a standard subroutine call and avoids this blocking mechanism. The code in Fig. 5 illustrates the use of the **await** statement

```
class Bus {
  Bool unlocked = True;
  Unit lock_bus{await unlocked; unlocked = False;}
  Unit release_bus{unlocked = True;} }
```

Fig. 5. Bus lock implementation in ABS using await on Booleans.

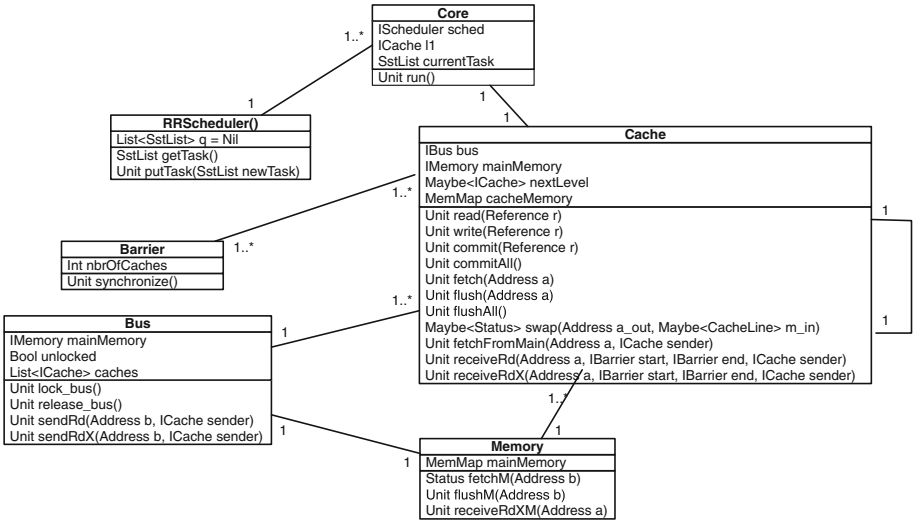


Fig. 6. Class diagram of the ABS model.

on a Boolean condition to model a binary semaphore, which is used to enforce exclusive access to a communication medium implemented as a "bus". Thus, the statement **await** bus!lock_bus() will suspend the calling method invocation (and release control in the caller object) and will be resumed when the generated invocation of the method lock_bus of the "bus" itself has been resumed when the local condition unlocked (of the "bus") has become true.

3.2 The Structural View

The runtime syntax of the SOS is represented by ABS classes, as outlined in Fig. 6. We briefly overview the translation. In ABS, object identifiers guarantee unique names and object references are used to capture how cores and caches are related. These references are encoded in a one-to-one correspondence with the naming scheme of the SOS.

A core $cid \bullet rst$ is translated into a class Core with a field currentTask representing the current task rst. Each core holds a reference to the first level cache. A cache memory $caid \bullet M \bullet dst$ is translated into a class Cache with an interface ICache and a class parameter nextLevel. In a cache, nextLevel holds a reference to the next level cache. If this reference is Nothing, it is last level cache (in the SOS, a predicate *last* is used to identify the last level). The field cacheMemory models the cache's memory M in SOS. The process pool of each cache object in ABS represents the data instruction set dst.

An ABS configuration consists of a number of cores with their corresponding cache hierarchies, the main memory, a scheduler with tasks waiting to be scheduled, and the ABS classes Bus and Barrier, which model the abstract communication medium and the global synchronization with labels $!Rd(n)$ and $!RdX(n)$

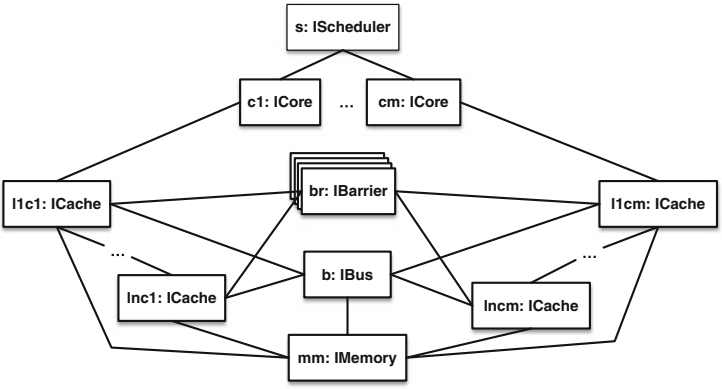


Fig. 7. Object diagram of an initial configuration.

in the SOS. The object diagram in Fig. 7 shows an initial configuration corresponding to the one depicted in Fig. 1.

3.3 The Behavioral View

We discuss in this section the design patterns in ABS that implement the synchronization inherent in the SOS model. We observe here that the combination of asynchronous method calls and cooperative scheduling is crucial because of the *multitasking* inherent in the SOS model, which requires that objects need to be able to process other requests; e.g., caches need to flush memory blocks while waiting for a fetch to succeed.

Local synchronization in the SOS model between two structural entities (e.g., two caches in rule LC-HIT$_1$ of Fig. 3), is implemented by the following synchronization pattern in ABS (see Fig. 8). Given two objects o_1 and o_2, let o_1 execute method m_1, which checks the local conditions of o_1 (highlighted as region **A** in Fig. 8). If these local conditions hold, method m_2 on o_2 is called asynchronously. Method m_2 completes when the local conditions of o_2 hold (highlighted as region **B** in Fig. 8). However, when m_2 has returned and object o_1 again schedules method m_1, the conditions on object o_2 need no longer hold. Therefore, o_1 next calls the method m_3 *synchronously* to check these conditions again. If these condition still hold, method m_3 returns successfully (in general, having updated o_2), and we can proceed to do the local changes in o_1 (highlighted

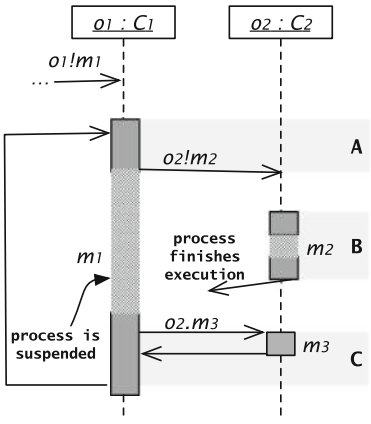


Fig. 8. Local synchronization between two ABS objects.

```
Just(nextCache) => {
  Maybe<Status> s = Nothing;
  Maybe<CacheLine> selected = Nothing;
  while (s == Nothing) {
    retValue = await nextCache!fetch(n);
    selected = select(cacheMemory, maxSize, n);
    s = nextCache.swap(n,selected,name);
  }
  case selected {
    Nothing => skip;
    Just(Pair(n1,_)) => cacheMemory = removeKey(cacheMemory,n1);
  }
  cacheMemory = put(cacheMemory, n, fromJust(s));
```

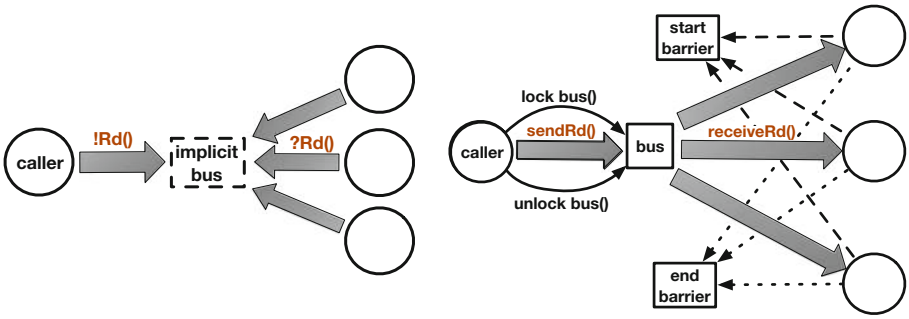
Fig. 9. Extract of ABS method fetch. When this code is reached, the requested cache line n has status invalid or it is not in the cache. The function select chooses a cache line to be swapped with n. If there is still free space in the cache, select returns Nothing. If n has either shared or modified status in the next level cache, the method swap removes the cache line with address n, inserts the selected cache line and returns the current status of n; otherwise, swap simply returns Nothing.

as region **C** in Fig. 8). Otherwise, the process needs to be repeated until we succeed. Note that method m_3 should not contain release points; because this method is called synchronously from a different object, a release point will in general have the potential of introducing deadlocks in the caller object.

To illustrate the above protocol, consider the code snippet in Fig. 9, which corresponds to part of several rules in the SOS (in particular, rule LC-HIT$_1$). Here, the current object **this** corresponds to $caid_i$ in the SOS, running method fetch, and the referenced object in nextCache corresponds to $caid_j$. When fetch from nextCache returns, all the required conditions in nextCache are *True*. However, since the call is asynchronous, (some of) the conditions may no longer hold when execution continues in **this**. This is addressed by checking the return value of method swap: If swap returns an address, it means the conditions still hold and the necessary updates are performed both locally and in nextCache; otherwise (when swap returns Nothing) fetch will be called again.

Global synchronization in the SOS (see Fig. 10a) is modelled by matching labelled transitions. To simulate this instantaneous communication in ABS, we introduced the classes Bus and Barrier. The synchronization protocol is activated by asynchronous calls to the respective methods sendRd and sendRdX of the bus. The bus subsequently asynchronously calls the corresponding methods receiveRd and receiveRdX of the caches. Two barriers start and end are used by the caches to synchronize the start, as well as the completion, of the local executions of methods receiveRd and receiveRdX.

However, observe that objects in ABS are input enabled: it is always possible to call a method on an object. In our model, this scheme may give rise



(a) State machine of the global synchronization using labels in the SOS model.

(b) State machine of the global synchronization using a bus and barriers in the ABS model.

Fig. 10. Synchronization in SOS vs ABS. In the SOS model (a), circles represent nodes in the memory system and shaded arrows labelled transitions. Note that the bus is *implicit* in the SOS model, as synchronization is captured by label matching. In the ABS model (b), circles represent the same nodes as in the SOS model, shaded arrows method invocations, solid arrows mutual access to the bus object and dotted arrows barrier synchronizations.

to inconsistent states: the local status of a memory location which triggers an asynchronous call of one of the methods sendRd and sendRdX of the bus may be invalidated by other bus synchronizations. Therefore, we add a lock to the bus (see Figs. 5 and 6), which is used to ensure exclusive access to the *message pool* of the bus when one of the methods read, write, and fetch are executed. The lock is released in case bus synchronization is not needed. The overall scheme is depicted in Fig. 10b. The exclusive access to the message pool of the bus guarantees that the message pool of the bus contains at most one call to one of the methods sendRd and sendRdX. Consequently, the triggering condition of the call cannot be invalidated before the call has been executed. This *strict* locking strategy, however, decreases concurrency in the distributed system, but reduces the complexity of the proof of equivalence between the SOS and the distributed implementation. We discuss how to further enhance the parallelization in Sect. 5.

4 Correctness

In this section we discuss the correctness of the ABS model by means of a simulation relation between the transition system describing the semantics of the ABS model of the multicore memory system and the transition system described by the SOS model.

The semantics of an ABS model can be described by a transition relation between global configurations. A global configuration is a (finite) set of object configurations. An object configuration is a tuple of the form $\langle oid, \sigma, p, Q \rangle$, where *oid* denotes the unique identity of the object, σ assigns values to the instance variables (fields) of the object, p denotes the currently executing process, and Q

denotes a set of (suspended) processes. A process is a closure (τ, S) consisting of an assignment τ of values to the local variables of the statement S.

We refer to [4] for the details of the structural operational semantics for deriving transitions $G \to G'$ between global configurations in ABS. Since in ABS concurrent objects only interact via asynchronous method calls and processes are scheduled non-deterministically (which provides an abstraction from the order in which the processes are generated by method calls), the ABS semantics satisfies the following global confluence property that allows to commute consecutive computations steps of *independent* processes which belong to *different* objects. Two processes are independent if neither one is generated by the other by an asynchronous call.

Lemma 1 (Global confluence). *For any two transitions $G \to G_1$ and $G \to G_2$ that describe execution steps of independent processes of different objects, there exists a global configuration G' such that $G_1 \to G'$ and $G_2 \to G'$.*

An object configuration is *stable* if the statement S to be executed has terminated or starts either with a **get** operation on a future or with an **await** statement on a Boolean condition or a future. A global ABS configuration is *stable* if all its object configurations are stable. Observe that our ABS model does not give rise to local divergent computations without passing through stable configurations; i.e., every local computation eventually enters a stable configuration. Together with the global confluence property in Lemma 1, this allows to restrict the semantics of the ABS model in the simulation relation to stable global configurations; i.e., transitions $G \Rightarrow G'$ between stable global configurations G and G' which result from a (non-empty) sequence of local execution steps of a *single* process from one stable configuration to a next one.

Because of the global synchronization with the bus in ABS described above, we may also represent without loss of generality the synchronization on the bus by a *single* global transition $G \Rightarrow G'$ which involves a completed execution of the method sendRd(...) (or sendRdX(...)) by the bus. This is justified because the global confluence allows for a scheduling policy such that the execution of the processes that are generated by these methods, i.e., the calls of the methods receiveRd(...) (or receiveRd(...)) are not interleaved with any other processes.

The simulation relation. The structural correspondence between a global configuration of the ABS model and a configuration of the SOS model is described in Sect. 3.2. For each method we have constructed a table which, among others, associates with some, so-called *observable*, occurrences of **await** statements (appearing in the method body) a corresponding **dst** instruction. In general, the execution of the remaining (occurrences of) **await** statements, for which there does not exist a corresponding **dst** instruction, involves some asynchronous messaging *preparing* for the corresponding synchronous exchange of information in the SOS model. In some cases, the execution of these unobservable statements (e.g., the read and write methods) also does not correspond to a change of the SOS configuration. Let α map every stable global configuration G of the ABS model to a structurally equivalent configuration $\alpha(G)$ of the SOS model, which

additionally maps every observable process (either queued or active) to the associated **dst** instruction (a process is observable if its corresponding statement is observable).

We arrive at the following theorem which expresses that the ABS model is a correct implementation of the abstract model.

Theorem 1. *Let G be a stable global configuration of the ABS model. If $G \Rightarrow G'$ then $\alpha(G) \rightarrow^* \alpha(G')$, where $\rightarrow^*$ denotes the reflexive, transitive closure of $\rightarrow$.*

Proof. The proof proceeds by a case analysis of the given transition $G \Rightarrow G'$, which, as discussed above, involves the local execution of some basic sequential code by a single object. For example, for the case of a completed execution of a method sendRd(...) (or sendRdX(...)) by the bus, a simple inspection of the sequential code of the methods that have been executed, e.g., sendRd(...) and receiveRd(...), suffices to establish the existence of a corresponding transition $\alpha(G) \rightarrow \alpha(G')$.

The remaining cases are captured by tables (as mentioned above) which provide for each method the following information. The statements in the **Location** column of each table represent for the respective method all possible processes generated by a call, i.e., a call to the method itself, and the processes which correspond to the **await** statements appearing in its body. In each row the **Next release point** statement indicates the next **await** statement or **return** statement that can be reached (statically). The **dst** instruction in each row specifies the instruction which corresponds to the **Location** statement in the simulation. Finally, **Enable condition** in each row specifies the enabling conditions (expressed in the abstract model) of the rule applications (of the abstract model) specified in **Rules**. In general these rule applications involve the sequential application of one or more rules. For unobservable statements, for which there is no corresponding **dst** instruction, the latter two columns are left unspecified.

The case analysis then consists of checking statically for each row the *local* structural correspondence between the resulting ABS process (the **Next release point**) and the resulting SOS configuration described by the specified rule applications.

5 Parallelism and Fairness of the ABS Model

This section discusses how to relax the eager locking policy of the bus implementation, without generating inconsistent states. Instead of locking the bus unconditionally when executing the read, write, and fetch methods in the ABS model, and releasing the lock when no bus synchronization is required, we only lock the bus when the triggering conditions of the bus synchronization may be invalidated. For example, an *optimistic* write implementation (see Fig. 11) tries to acquire the lock of the bus, and only after the acquisition checks if a race-condition has happened and invalidated the shared status of the address n; in this case, the write method will *backtrack* and retry (by calling itself); otherwise the write operation can safely be performed.

```
Int write(Ref r) {
 case lookup(cacheMemory, addr(r)) {
    Just(Sh) => {
      await bus!lock_bus();
      // after waking up do RACE DETECTION
      if (lookup(cacheMemory,addr(r)) == Just(Sh)) { // NO RACE
        await bus!sendRdX(addr(r),this);
        await bus!release_bus();
        cacheMemory = put(cacheMemory,addr(r),Mo);
      }
      else { // RACE CONDITION
        await bus!release_bus();
        await this!write(r); // RETRY
      } } ... } }
```

Fig. 11. Alternative, optimistic implementation of the write method to detect a bus race-condition and, in that case, retry the operation.

The strict and relaxed variations of the global synchronization bear strong resemblance respectively to conservative [11,12] and optimistic [13] algorithms in parallel and distributed discrete-event simulation (PDES) [14]. As with PDES, there is no clear winner between the strict (conservative) and relaxed (optimistic) versions of our cache simulator; certain computer programs (input-models) will be simulated faster using one version or the other, depending on the inter-dependency of the parallel components (for us, the caches). For the contrived experiment, we implemented a penalty system in the ABS model. A cache penalty is the cost (delay) incurred by failing to read or write to a particular level of cache—set here to $(L_1, L_2, L_3) =_{cost} (1, 10, 100)$ [15]. We compared the two versions for a scenario with full inter-dependency (simultaneous write instructions on the same memory block) and a scenario with minimal inter-dependency (write instructions on separate memory blocks) between 16 simulated cores. In these experiments the strict version was slightly faster up to 2% for the first case and losing out by up to 12% in the second case. The experiments were executed using the ABS-Erlang backend [16] and Erlang version 21, running on quad-socket 8-cores 16-hyperthreads Xeon®L7555, which yielded in total 64 hardware threads.

Fairness. A concern that often arises in parallel execution is fairness: the degree of variability when distributing the computing resources among different parallel components—here, the simulated cores. Fairness of parallel execution can affect the simulation's accuracy in approximating the intended (or idealized) many-core hardware. To ensure fairness of the simulation, we make use of *deployment components* [17] in ABS.

A *Deployment Component* (DC) is an ABS execution location that is created with a number of virtual resources (e.g., execution speed, memory use, network bandwidth), which are shared among its deployed objects. Any annotated statement [Cost: x] S decrements by x the resources of its DC and then completes, or

Table 1. Total cache penalties between strict/relaxed, with/without DC configurations.

	Strict with DC	Relaxed with DC	Strict	Relaxed
$\sum_{penalty}$	43068	43290	39183	24956

it will stall its computation if there are currently not enough resources remaining; the statement S may continue on the next passage of the global symbolic time where all the resources of the DCs have been renewed, and will eventually complete when its Cost has reached zero.

We make use of this resource modeling of ABS to assign equal (fair) resources of virtual execution speed to the simulated cores of the system. Each Core object is deployed onto a separate DC with fixed Speed(1) resources. The processing of each instruction has the same cost [Cost: 1]—a generalization, since common processor architectures execute different instructions in different speeds (cycles per instruction); e.g., JUMP is faster than LOAD. The result is that all Cores can execute maximum one instruction in every time interval of the global symbolic clock, and thus no Core can get too far ahead with processing its own instructions—a problem that manifests upon the parallel simulation of N number of cores using a physical machine of M cores, where N is vastly greater than M. To test this, we performed a write-congested experiment with a configuration of 20 simulated cores and 3 cache levels, comparing the strict and relaxed variations, with and without the use of deployment components. The results (shown in Table 1) were measured on a quad-core system running ABS-Erlang, counting the total cache penalties of all the cores. With respect to the strict variation, the results with and without DC have similar penalties; this can be attributed to the lock-step nature of strict bus synchronization, where no cache (and thus core) can unfairly stride forward. In the relaxed variation, however, where synchronization is less strict, we see that without the fairness imposed by DC, the penalties are almost halved, which means some cores are allowed to do multiple (successful) write operations while other cores are still waiting on the "backlog" to be simulated. This gives rise to less penalties, because of less runtime interleavings of the simulated cores and thus less competition between them.

6 Related Work

There is in general a significant gap between a formal model and its implementation [18]. SOS [1] succinctly formalizes operational models and are well-suited for proofs, but direct implementations of SOS quickly lead to very inefficient implementations. Executable semantic frameworks such as Redex [19], rewriting logic [20,21], and $\mathbb{K}$ [22] reduce this gap, and have been used to develop executable formal models of complex languages like C [23] and Java [24]. The relationship between SOS and rewriting logic semantics has been studied [25] without proposing a general solution for label matching. Bijo et al. implemented their SOS multicore memory model [26] in the rewriting logic system Maude

[3] using an orchestrator for label matching, but do not provide a correctness proof wrt. the SOS. Different semantic styles can be modeled and related inside one framework; for example, the correctness of distributed implementations of KLAIM systems in terms of simulation relations have been studied in rewriting logic [27]. Compared to these works on semantics, we implemented an SOS model in a distributed active object setting, and proved the correctness of this implementation.

Correctness-preserving compilation is related to correctness proofs for implementations, and ensures that the low-level representation of a program preserves the properties of the high-level model. Examples of this line of work include type-preserving translations into typed assembly languages [28] and formally verified compilers [29,30], which proves the semantic preservation of a compiler from C to assembler code, but leaves shared-variable concurrency for future work. In contrast to this work which studies compilation from one language to another, our work focuses on a specific model and its implementation and specifically targets parallel systems.

Simulation tools for cache coherence protocols can evaluate performance and efficiency on different architectures (e.g., gems [31] and gem5 [32]). These tools perform evaluations of, e.g., the cache hit/miss ratio and response time, by running benchmark programs written as low-level read and write instructions to memory. Advanced simulators such as Graphite [33] and Sniper [34] run programs on distributed clusters to simulate executions on multicore architectures with thousands of cores. Unlike our work, these simulators are not based on a formal semantics and correctness proofs. Our work complements these simulators by supporting the executable exploration of design choices from a programmer perspective rather from hardware design. Compared to worst-case response time analysis for concurrent programs on multicore architectures [35], our focus is on the underlying data movement rather than the response time.

7 Conclusion

We have introduced in this paper a methodology for implementing SOS models in the active object language ABS, and applied this methodology to the implementation of a SOS model of an abstraction of multicore memory systems, resulting in a parallel simulator for these systems. A challenge for this implementation is to correctly implement the synchronization patterns of the SOS rules, which may cross encapsulation barriers in the active objects, and in particular label synchronization on parallel transitions steps. We prove the correctness of this particular implementation, exploiting that the ABS model allows for a high-level coarse-grained semantics. We investigated the further parallelization and fairness of the ABS model.

The results obtained in this paper provide a promising basis for further development of the ABS model for simulating the execution of (object-oriented) programs on multicore architectures. A first such development concerns an extension of the abstract memory model with data. In particular, having the addresses of

the memory locations themselves as data allows to model and simulate different data layouts of the dynamically generated object structures.

References

1. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. **60–61**, 17–139 (2004)
2. Bijo, S., Johnsen, E.B., Pun, K.I, Tapia Tarifa, S.L.: A formal model of parallel execution on multicore architectures with multilevel caches. In: Proença, J., Lumpe, M. (eds.) FACS 2017. LNCS, vol. 10487, pp. 58–77. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68034-7_4
3. Clavel, M., et al. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
4. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
5. Boer, F.D., et al.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017)
6. Bijo, S., Johnsen, E.B., Pun, K.I, Tapia Tarifa, S.L.: A formal model of parallel execution in multicore architectures with multilevel caches (long version). Research report, Department of Informatics, University of Oslo (2018). Under revision for journal publication. http://violet.at.ifi.uio.no/papers/mc-rr.pdf
7. Solihin, Y.: Fundamentals of Parallel Multicore Architecture, 1st edn. Chapman & Hall/CRC, Boca Raton (2015)
8. Culler, D.E., Gupta, A., Singh, J.P.: Parallel Computer Architecture: A Hardware/Software Approach, 1st edn. Morgan Kaufmann Publishers Inc., Los Altos (1997)
9. Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence, 1st edn. Morgan & Claypool Publishers, San Francisco (2011)
10. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI 1973, pp. 235–245. Morgan Kaufmann Publishers Inc., San Francisco (1973)
11. Bryant, R.E.: Simulation of packet communication architecture computer systems. Technical report MIT/LCS/TR-188, MIT, Lab for Computer Science, November 1977
12. Chandy, K.M., Misra, J.: Distributed simulation: a case study in design and verification of distributed programs. IEEE Trans. Softw. Eng. **SE–5**(5), 440–452 (1979)
13. Jefferson, D.R.: Virtual time. ACM Trans. Program. Lang. Syst. **7**(3), 404–425 (1985)
14. Fujimoto, R.M.: Parallel and Distributed Simulation Systems. Wiley, Hoboken (2000)
15. Schmidl, D., Vesterkjær, A., Müller, M.S.: Evaluating OpenMP performance on thousands of cores on the numascale architecture. In: Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing (ParCo 2015). Advances in Parallel Computing, vol. 27, pp. 83–92. IOS Press (2016)

16. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. STTT **14**(5), 567–588 (2012)
17. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. J. Log. Algebr. Methods Program. **84**(1), 67–91 (2015)
18. Schlatte, R., Johnsen, E.B., Mauro, J., Tapia Tarifa, S.L., Yu, I.C.: Release the beasts: when formal methods meet real world data. In: de Boer, F., Bonsangue, M., Rutten, J. (eds.) It's All About Coordination. LNCS, vol. 10865, pp. 107–121. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90089-6_8
19. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press, Cambridge (2009)
20. Meseguer, J., Rosu, G.: The rewriting logic semantics project: a progress report. Inf. Comput. **231**, 38–69 (2013)
21. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3), 213–237 (2007)
22. Rosu, G.: $\mathbb{K}$: a semantic framework for programming languages and formal analysis tools. In: Dependable Software Systems Engineering, pp. 186–206. IOS Press (2017)
23. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012), pp. 533–544. ACM (2012)
24. Bogdanas, D., Rosu, G.: K-Java: a complete semantics of Java. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015), pp. 445–456. ACM (2015)
25. Serbanuta, T., Rosu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Inf. Comput. **207**(2), 305–340 (2009)
26. Bijo, S., Johnsen, E.B., Pun, K.I, Tapia Tarifa, S.L.: A maude framework for cache coherent multicore architectures. In: Lucanu, D. (ed.) WRLA 2016. LNCS, vol. 9942, pp. 47–63. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44802-2_3
27. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Semantics, distributed implementation, and formal analysis of KLAIM models in Maude. Sci. Comput. Program. **99**, 24–74 (2015)
28. Morrisett, J.G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Trans. Program. Lang. Syst. **21**(3), 527–568 (1999)
29. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
30. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**(4), 363–446 (2009)
31. Martin, M.M.K., et al.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. SIGARCH Comput. Arch. News **33**(4), 92–99 (2005)
32. Binkert, N., et al.: The gem5 simulator. SIGARCH Comput. Arch. News **39**(2), 1–7 (2011)
33. Miller, J.E., et al.: Graphite: a distributed parallel simulator for multicores. In: Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA), pp. 1–12. IEEE Computer Society (2010)

34. Carlson, T.E., Heirman, W., Eeckhout, L.: Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 52:1–52:12. ACM (2011)
35. Li, Y., Suhendra, V., Liang, Y., Mitra, T., Roychoudhury, A.: Timing analysis of concurrent programs running on shared cache multi-cores. In: Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS), pp. 57–67. IEEE Computer Society (2009)

Optimal and Automated Deployment for Microservices

Mario Bravetti[1], Saverio Giallorenzo[2(✉)], Jacopo Mauro[2], Iacopo Talevi[1], and Gianluigi Zavattaro[1]

[1] FOCUS Research Team, University of Bologna/Inria, Bologna, Italy
[2] University of Southern Denmark, Odense, Denmark
saverio@imada.sdu.dk

Abstract. Microservices are highly modular and scalable Service Oriented Architectures. They underpin automated deployment practices like Continuous Deployment and Autoscaling. In this paper we formalize these practices and show that automated deployment — proven undecidable in the general case — is algorithmically treatable for microservices. Our key assumption is that the configuration life-cycle of a microservice is split into two phases: (i) creation, which entails establishing initial connections with already available microservices, and (ii) subsequent binding/unbinding with other microservices. To illustrate the applicability of our approach, we implement an automatic optimal deployment tool and compute deployment plans for a realistic microservice architecture, modeled in the Abstract Behavioral Specification (ABS) language.

1 Introduction

Inspired by service-oriented computing, Microservices structure software applications as highly modular and scalable compositions of fine-grained and loosely-coupled services [18]. These features support modern software engineering practices, like continuous delivery/deployment [30] and application autoscaling [3]. Currently, these practices focus on single microservices and do not take advantage of the information on the interdependencies within an architecture. On the contrary, architecture-level deployment supports the global optimization of resource usage and avoids "domino" effects due to unstructured scaling actions that may cause cascading slowdowns or outages [27,35,39].

In this paper, we formalize the problem of automatic deployment and reconfiguration (at the architectural level) of microservice systems, proving formal properties and presenting an implemented solution.

In our work, we follow the approach taken by the *Aeolus component model* [13–15], which was used to formally define the problem of deploying component-based software systems and to prove that, in the general case, such problem is undecidable [15]. The basic idea of Aeolus is to enrich the specification of components with a finite state automaton that describes their deployment life cycle. Previous work identified decidable fragments of the Aeolus model: e.g.,

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 351–368, 2019.
https://doi.org/10.1007/978-3-030-16722-6_21

removing from Aeolus replication constraints (e.g., used to specify a minimal amount of services connected to a load balancer) makes the deployment problem decidable, but non-primitive recursive [14]; removing also conflicts (e.g., used to express the impossibility to deploy in the same system two types of components) makes the problem PSpace-complete [34] or even poly-time [15], but under the assumption that every required component can be (re)deployed from scratch.

Our intuition is that the Aeolus model can be adapted to formally reason on the deployment of microservices. To achieve our goal, we significantly revisit the formalization of the deployment problem, replacing Aeolus components with a model of *microservices*. The main difference between our model of microservices and Aeolus components lies in the specification of their deployment life cycle. Here, instead of using the full power of finite state automata (like in Aeolus and other TOSCA-compliant deployment models [10]), we assume microservices to have two states: (i) creation and (ii) binding/unbinding. Concerning creation, we use *strong* dependencies to express which microservices must be immediately connected to newly created ones. After creation, we use *weak* dependencies to indicate additional microservices that can be bound/unbound. The principle that guided this modification comes from state-of-the-art microservice deployment technologies like Docker [36] and Kubernetes [29]. In particular, the weak and strong dependencies have been inspired by Docker Compose [16] (a language for defining multi-container Docker applications) where it is possible to specify different relationships among microservices using, e.g., the depends_on (resp. external_links) modalities that force (resp. do not force) a specific startup order similarly to our strong (resp. weak) dependencies. Weak dependencies are also useful to model horizontal scaling, e.g., a load balancer that is bound to/unbound from many microservice instances during its life cycle.

In addition, w.r.t. the Aeolus model, we also consider resource/cost-aware deployments, taking inspiration from the memory and CPU resources found in Kubernetes. Microservice specifications are enriched with the amount of resources they need to run. In a deployment, a system of microservices runs within a set of computation *nodes*. Nodes represent computational units (e.g., virtual machines in an Infrastructure-as-a-Service Cloud deployment). Each node has a cost and a set of resources available to the microservices it hosts.

On the model above, we define the *optimal deployment problem* as follows: given an initial microservice system, a set of available nodes, and a new target microservice to be deployed, find a sequence of reconfiguration actions that, once applied to the initial system, leads to a new deployment that includes the target microservice. Such a deployment is expected to be *optimal*, meaning that the total cost (i.e., the sum of the costs) of the nodes used is minimal. We show that this problem is decidable by presenting an algorithm working in three phases: (1) generate a set of constraints whose solution indicates the microservices to be deployed and their distribution over the nodes; (2) generate another set of constraints whose solution indicates the connections to be established; (3) synthesize the corresponding deployment plan. The set of constraints includes optimization metrics that minimize the overall cost of the computed deployment.

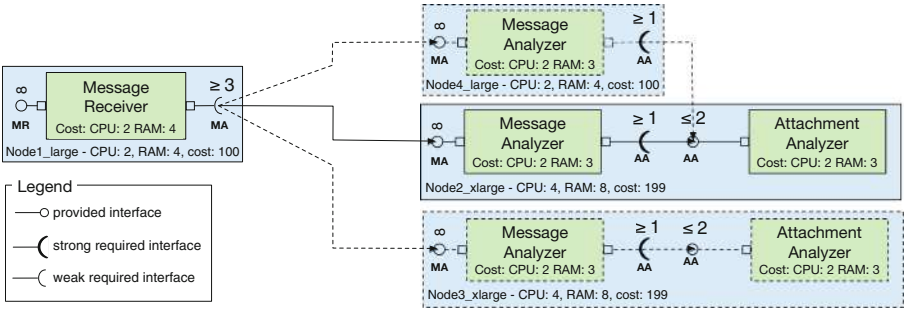


Fig. 1. Example of microservice deployment (blue boxes: nodes; green boxes: microservices; continuous lines: the initial configuration; dashed lines: full configuration). (Color figure online)

The algorithm has NEXPTIME complexity because, in the worst-case, the length of the deployment plan could be exponential in the size of the input. However, we consider this worst-case unfeasible in practice, as the number of microservices deployable on one node is limited by the available resources. Under the assumption that each node can host at most a polynomial amount of microservices, the deployment problem is NP-complete and the problem of deploying a system minimizing its total cost is an NP-optimization problem. Moreover, having reduced the deployment problem in terms of constraints, we can exploit state-of-the-art constraint solvers [12,23,24] that are frequently used in practice to cope with NP-hard problems.

To concretely evaluate our approach, we consider a real-world microservice architecture, inspired by the reference email processing pipeline from Iron.io [22]. We model that architecture in the Abstract Behavioral Specification (ABS) language, a high-level object-oriented language that supports deployment modeling [31]. We use our technique to compute two types of deployments: an initial one, with one instance for each microservice, and a set of deployments to horizontally scale the system depending on small, medium or large increments in the number of emails to be processed. The experimental results are encouraging in that we were able to compute deployment plans that add more than 30 new microservice instances, assuming availability of hundreds of machines of three different types, and guaranteeing optimality.

2 The Microservice Optimal Deployment Problem

We model microservice systems as aggregations of components with ports. Each port exposes provided and required interfaces. Interfaces describe offered and required functionalities. Microservices are connected by means of bindings indicating which port provides the functionality required by another port. As discussed in the Introduction, we consider two kinds of requirements: strong required interfaces, that need to be already fulfilled when the microservice is created, and weak required interfaces, that must be fulfilled at the end of a

deployment (or reconfiguration) plan. Microservices are enriched with the specification of the resources they need to properly run; such resources are provided to the microservices by nodes. Nodes can be seen as the unit of computation executing the tasks associated to each microservice.

As an example, in Fig. 1 we have reported the representation of the deployment of a microservice system inspired by the email processing pipeline that we will discuss in Sect. 3. Here, we consider a simplified pipeline. A Message Receiver microservice handles inbound requests, passing them to a Message Analyzer that checks the email content and sends the attachments for inspection to an Attachment Analyzer. The Message Receiver has a port with a *weak* required interface that can be fulfilled by Message Analyzer instances. This requirement is weak, meaning that the Message Receiver can be initially deployed without any connection to instances of Message Analyzer. These connections can be established afterwards and reflect the possibility to horizontally scale the application by adding/removing instances of Message Analyzer. This last microservice has instead a port with a *strong* required interface that can be fulfilled by Attachment Analyzer instances. This requirement is strong to reflect the need to immediately connect a Message Analyzer to its Attachment Analyzer.

Figure 1 presents a reconfiguration that, starting from the initial deployment depicted in continuous lines, adds the elements depicted with dashed lines. Namely, a couple of new instances of Message Analyzer and a new instance of Attachment Analyzer are deployed. This is done in order to satisfy numerical constraints associated to both required and provided interfaces. For required interfaces, the numerical constraints indicate lower bounds to the outgoing bindings, while for provided interfaces they specify upper bounds to the incoming connections. Notice that the constraint ≥ 3 associated to the weak required interface of Message Receiver is not initially satisfied; this is not problematic because constraints on weak interfaces are relevant only at the end of a reconfiguration. In the final deployment, such a constraint is satisfied thanks to the two new instances of Message Analyzer. These two instances need to be immediately connected to an Attachment Analyzer: only one of them can use the initially available Attachment Analyzer, because of the constraint ≤ 2 associated to the corresponding provided interface. Hence, a new instance of Attachment Analyzer is added.

We also model resources: each microservice has associated resources that it consumes (see the CPU and RAM quantities associated to the microservices in Fig. 1). Resources are provided by nodes, that we represent as containers for the microservice instances, providing them the resources they require. Notice that nodes have also costs: the total cost of a deployment is the sum of the costs of the used nodes (e.g., in the example the total cost is 598 cents per hour, corresponding to the cost of 4 nodes: 2 C4 large and 2 C4 xlarge virtual machine instances of the Amazon public Cloud).

We now move to the formal definitions. We assume the following disjoint sets: $\mathcal{I}$ for interfaces, $\mathcal{Z}$ for microservices, and a finite set $\mathcal{R}$ for kinds of resources. We use $\mathbb{N}$ to denote natural numbers, $\mathbb{N}^+$ for $\mathbb{N} \setminus \{0\}$, and $\mathbb{N}^+_\infty$ for $\mathbb{N}^+ \cup \{\infty\}$.

Definition 1 (Microservice type). *The set Γ of* microservice types*, ranged over by $\mathcal{T}_1, \mathcal{T}_2, \ldots$, contains 5-ples $\langle P, D_s, D_w, C, R\rangle$ where:*

- $P = (\mathcal{I} \nrightarrow \mathbb{N}^+_\infty)$ *are the provided interfaces, defined as a partial function from interfaces to corresponding numerical constraints (indicating the maximum number of connected microservices);*
- $D_s = (\mathcal{I} \nrightarrow \mathbb{N}^+)$ *are the* strong *required interfaces, defined as a partial function from interfaces to corresponding numerical constraints (indicating the minimum number of connected microservices);*
- $D_w = (\mathcal{I} \nrightarrow \mathbb{N})$ *are the* weak *required interfaces (defined as the strong ones, with the difference that also the constraint 0 can be used indicating that it is not strictly necessary to connect microservices);*
- $C \subseteq \mathcal{I}$ *are the conflicting interfaces;*
- $R = (\mathcal{R} \to \mathbb{N})$ *specifies resource consumption, defined as a total function from resources to corresponding quantities indicating the amount of required resources.*

We assume sets $\texttt{dom}(D_s)$, $\texttt{dom}(D_w)$ *and* C *to be pairwise disjoint.*[1]

Notation: given a microservice type $\mathcal{T} = \langle P, D_s, D_w, C, R\rangle$, we use the following postfix projections `.prov`, `.reqs`, `.reqw`, `.conf` and `.res` to decompose it; e.g., $\mathcal{T}$`.reqw` returns the partial function associating arities to weak required interfaces. In our example, for instance, the Message Receiver microservice type is such that Message Receiver.$\texttt{reqw}(\mathsf{MA}) = 3$ and Message Receiver.$\texttt{res}(\mathsf{RAM}) = 4$. When the numerical constraints are not explicitly indicated, we assume as default value ∞ for provided interfaces (i.e., they can satisfy an unlimited amount of ports requiring the same interface) and 1 for required interfaces (i.e., one connection with a port providing the same interface is sufficient).

Inspired by [14], we allow a microservice to specify a conflicting interface that, intuitively, forbids the deployment of other microservices providing the same interface. Conflicting interfaces can be used to express conflicts among microservices, preventing both of them to be present at the same time, or cases in which only one microservice instance can be deployed (e.g., a consistent and available microservice that can not be replicated).

Since the requirements associated with strong interfaces must be immediately satisfied, it is possible to deploy a configuration with circular dependencies only if at least one weak required interface is involved in the cycle. In fact, having a cycle with only strong required interfaces would mean to deploy all the microservices involved in the cycle simultaneously. We now formalize a well-formedness condition on microservice types to guarantee the absence of such configurations.

Definition 2 (Well-formed Universe). *Given a finite set of microservice types U (that we also call* universe*), the strong dependency graph of U is as follows: $G(U) = (U, V)$ with $V = \{(\mathcal{T}, \mathcal{T}') | \mathcal{T}, \mathcal{T}' \in U \wedge \exists p \in \mathcal{I}.p \in \texttt{dom}(\mathcal{T}.\texttt{reqs}) \cap \texttt{dom}(\mathcal{T}'.\texttt{prov})\}$. The universe U is well-formed if $G(U)$ is acyclic.*

[1] Given a partial function f, we use $\texttt{dom}(f)$ to denote the domain of f, i.e., the set $\{e \mid \exists e' : (e, e') \in f\}$.

In the following, we always assume universes to be well-formed. Well-formedness does not prevent the specification of microservice systems with circular dependencies, which are captured by cycles with at least one weak required interface.

Definition 3 (Nodes). *The set $\mathcal{N}$ of* nodes *is ranged over by $o_1, o_2, \ldots$ We assume the following information to be associated to each node o in $\mathcal{N}$.*

- *A function $R = (\mathcal{R} \to \mathbb{N})$ that specifies node* resource availability*: we use $o.\mathtt{res}$ to denote such a function.*
- *A value in $\mathbb{N}$ that specifies node* cost*: we use $o.\mathtt{cost}$ to denote such a value.*

As example, in Fig. 1, the node Node1_large is such that $\mathsf{Node1\_large}.\mathtt{res}(\mathsf{RAM}) = 4$ and $\mathsf{Node1\_large}.\mathtt{cost} = 100$.

We now define configurations that describe systems composed of microservice instances and bindings that interconnect them. A configuration, ranged over by $\mathcal{C}_1, \mathcal{C}_2, \ldots$, is given by a set of microservice types, a set of deployed microservices (with their associated type), and a set of bindings. Formally:

Definition 4 (Configuration). *A* configuration *$\mathcal{C}$ is a 4-ple $\langle Z, T, N, B\rangle$ where:*

- *$Z \subseteq \mathcal{Z}$ is the set of the currently deployed* microservices*;*
- *$T = (Z \to \mathcal{T})$ are the* microservice types*, defined as a function from deployed microservices to microservice types;*
- *$N = (Z \to \mathcal{N})$ are the* microservice nodes*, defined as a function from deployed microservices to nodes that host them;*
- *$B \subseteq \mathcal{I} \times Z \times Z$ is the set of* bindings*, namely 3-ples composed of an interface, the microservice that requires that interface, and the microservice that provides it; we assume that, for $(p, z_1, z_2) \in B$, the two microservices z_1 and z_2 are distinct and $p \in (\mathtt{dom}(T(z_1).\mathtt{req_s}) \cup \mathtt{dom}(T(z_1).\mathtt{req_w})) \cap \mathtt{dom}(T(z_2).\mathtt{prov})$.*

In our example, if we use mr to refer to the instance of Message Receiver, and ma for the initially available Message Analyzer, we will have the binding (MA, mr, ma). Moreover, concerning the microservice placement function N, we have $N(\mathsf{mr}) = \mathsf{Node1\_large}$ and $N(\mathsf{ma}) = \mathsf{Node2\_xlarge}$.

We are now ready to formalize the notion of correctness of configuration. We first define a *provisional correctness*, considering only constraints on strong required and provided interfaces, and then we define a general notion of configuration correctness, considering also weak required interfaces and conflicts. The former is intended for transient configurations traversed during the execution of a reconfiguration, while the latter for the final configuration.

Definition 5 (Provisionally correct configuration). *A configuration $\mathcal{C} = \langle Z, T, N, B\rangle$ is* provisionally correct *if, for each node $o \in \mathtt{ran}(N)$, it holds*[2]

$$\forall\, r \in \mathcal{R}.\;\; o.\mathtt{res}(r) \geq \sum_{z \in Z, N(z) = o} T(z).\mathtt{res}(r)$$

and, for each microservice $z \in Z$, both following conditions hold:

[2] Given a (partial) function f, we use $\mathtt{ran}(f)$ to denote the range of f, i.e., the function image set $\{f(e) \mid e \in \mathtt{dom}(f)\}$.

- $(p \mapsto n) \in T(z).\texttt{reqs}$ *implies that there exist* n *distinct microservices* $z_1, \ldots, z_n \in Z \setminus \{z\}$ *such that, for every* $1 \leq i \leq n$, *we have* $\langle p, z, z_i \rangle \in B$;
- $(p \mapsto n) \in T(z).\texttt{prov}$ *implies that there exist no* m *distinct microservices* $z_1, \ldots, z_m \in Z \setminus \{z\}$, *with* $m > n$, *such that, for every* $1 \leq i \leq m$, *we have* $\langle p, z_i, z \rangle \in B$.

Definition 6 (Correct configuration). *A configuration* $\mathcal{C} = \langle Z, T, N, B \rangle$ *is* correct *if* $\mathcal{C}$ *is provisionally correct and, for each microservice* $z \in Z$, *both following conditions hold:*

- $(p \mapsto n) \in T(z).\texttt{req}_\texttt{W}$ *implies that there exist* n *distinct microservices* $z_1, \ldots, z_n \in Z \setminus \{z\}$ *such that, for every* $1 \leq i \leq n$, *we have* $\langle p, z, z_i \rangle \in B$;
- $p \in T(z).\texttt{conf}$ *implies that, for each* $z' \in Z \setminus \{z\}$, *we have* $p \notin \texttt{dom}(T(z').\texttt{prov})$.

Notice that, in the example in Fig. 1, the initial configuration (in continuous lines) is only provisionally correct in that the weak required interface MA (with arity 3) of the Message Receiver is not satisfied (because there is only one outgoing binding). The full configuration — including also the elements in dotted lines — is instead correct: all the constraints associated to the interfaces are satisfied.

We now formalize how configurations evolve by means of atomic actions.

Definition 7 (Actions). *The set* $\mathcal{A}$ *contains the following actions:*

- *bind*(p, z_1, z_2) *where* $z_1, z_2 \in \mathcal{Z}$, *with* $z_1 \neq z_2$, *and* $p \in \mathcal{I}$: *add a binding between* z_1 *and* z_2 *on port* p *(which is supposed to be a weak-require port of* z_1 *and a provide port of* z_2*);*
- *unbind*(p, z_1, z_2) *where* $z_1, z_2 \in \mathcal{Z}$, *with* $z_1 \neq z_2$, *and* $p \in \mathcal{I}$: *remove the specified binding on* p *(which is supposed to be a weak required interface of* z_1 *and a provide port of* z_2*);*
- *new*$(z, \mathcal{T}, o, B_s)$ *where* $z \in \mathcal{Z}$, $\mathcal{T} \in \Gamma$, $o \in \mathcal{N}$ *and* $B_s = (\texttt{dom}(\mathcal{T}.\texttt{reqs}) \to 2^{\mathcal{Z} - \{z\}})$; *with* B_s *(representing bindings from strong required interfaces in* $\mathcal{T}$ *to sets of microservices) being such that, for each* $p \in \texttt{dom}(\mathcal{T}.\texttt{reqs})$, *it holds* $|B_s(p)| \geq \mathcal{T}.\texttt{reqs}(p)$: *add a new microservice* z *of type* $\mathcal{T}$ *hosted in* o *and bind each of its strong required interfaces to a set of microservices as described by* B_s;[3]
- *del*(z) *where* $z \in \mathcal{Z}$: *remove the microservice* z *from the configuration and all bindings involving it.*

In our example, assuming that the initially available Attachment Analyzer is named aa, we have that the action to create the initial instance of Message Analyzer is *new*(ma, MessageAnalyzer, Node2_xlarge, (AA $\mapsto$ {aa})). Notice that it is necessary to establish the binding with the Attachment Analyzer because of the corresponding strong required interface.

The execution of actions can now be formalized using a labeled transition system on configurations, which uses actions as labels.

[3] Given sets S and S' we use: 2^S to denote the power set of S, i.e., the set $\{S' \mid S' \subseteq S\}$; $S - S'$ to denote set difference; and $|S|$ to denote the cardinality of S.

Definition 8 (Reconfigurations). *Reconfigurations are denoted by transitions* $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ *meaning that the execution of* $\alpha \in \mathcal{A}$ *on the configuration* $\mathcal{C}$ *produces a new configuration* $\mathcal{C}'$*. The transitions from a configuration* $\mathcal{C} = \langle Z, T, N, B \rangle$ *are defined as follows:*

$$\mathcal{C} \xrightarrow{bind(p,z_1,z_2)} \langle Z, T, N, B \cup \langle p, z_1, z_2 \rangle \rangle$$
if $\langle p, z_1, z_2 \rangle \notin B$ *and*
$p \in \mathtt{dom}(T(z_1).\mathtt{req_W}) \cap \mathtt{dom}(T(z_2).\mathtt{prov})$

$$\mathcal{C} \xrightarrow{unbind(p,z_1,z_2)} \langle Z, T, N, B \setminus \langle p, z_1, z_2 \rangle \rangle$$
if $\langle p, z_1, z_2 \rangle \in B$ *and*
$p \in \mathtt{dom}(T(z_1).\mathtt{req_W}) \cap \mathtt{dom}(T(z_2).\mathtt{prov})$

$$\mathcal{C} \xrightarrow{new(z,\mathcal{T},o,B_s)} \langle Z \cup \{z\}, T', N', B' \rangle$$
if $z \notin Z$ *and*
$\forall\, p \in \mathtt{dom}(\mathcal{T}.\mathtt{req_S}).\ \forall z' \in B_s(p).$
$\quad p \in \mathtt{dom}(T(z').\mathtt{prov})$ *and*
$T' = T \cup \{(z \mapsto \mathcal{T})\}$ *and*
$N' = N \cup \{(z \mapsto o)\}$ *and*
$B' = B \cup \{\langle p, z, z' \rangle \mid z' \in B_s(p)\}$

$$\mathcal{C} \xrightarrow{del(z)} \langle Z \setminus \{z\}, T', N', B' \rangle$$
if $T' = \{(z' \mapsto \mathcal{T}) \in T \mid z \neq z'\}$ *and*
$N' = \{(z' \mapsto o) \in N \mid z \neq z'\}$ *and*
$B' = \{\langle p, z_1, z_2 \rangle \in B \mid z \notin \{z_1, z_2\}\}$

A *deployment plan* is simply a sequence of actions that transform a provisionally correct configuration (without violating provisional correctness along the way) and, finally, reach a correct configuration.

Definition 9 (Deployment plan). *A* deployment plan P *from a provisionally correct configuration* $\mathcal{C}_0$ *is a sequence of actions* $\alpha_1, \ldots, \alpha_m$ *such that:*

- *there exist* $\mathcal{C}_1, \ldots, \mathcal{C}_m$ *provisionally correct configurations, with* $\mathcal{C}_{i-1} \xrightarrow{\alpha_i} \mathcal{C}_i$ *for* $1 \leq i \leq m$*, and*
- $\mathcal{C}_m$ *is a correct configuration.*

Deployment plans are also denoted with $\mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$.

In our example, a deployment plan that reconfigures the initial provisionally correct configuration into the final correct one is as follows: a *new* action to create the new instance of Attachment Analyzer, followed by two *new* actions for the new Message Analyzers (as commented above, the connection with the Attachment Analyzer is part of these *new* actions), and finally two *bind* actions to connect the Message Receiver to the two new instances of Message Analyzer.

We now have all the ingredients to define the *optimal deployment problem*, that is our main concern: given a universe of microservice types, a set of available nodes and an initial configuration, we want to know whether and how it is possible to deploy at least one microservice of a given microservice type $\mathcal{T}$ by optimizing the overall cost of nodes hosting the deployed microservices.

Definition 10 (Optimal deployment problem). *The* optimal deployment problem *has, as input, a finite well-formed universe* U *of microservice types, a finite set of available nodes* O*, an initial provisionally correct configuration* $\mathcal{C}_0$ *and a microservice type* $\mathcal{T}_t \in U$*. The output is:*

- *A* **deployment plan** $\mathsf{P} = \mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$ *such that*
  - *for all* $\mathcal{C}_i = \langle Z_i, T_i, N_i, B_i \rangle$*, with* $1 \leq i \leq m$*, it holds* $\forall z \in Z_i.\ T_i(z) \in U \wedge N_i(z) \in O$*, and*
  - $\mathcal{C}_m = \langle Z_m, T_m, N_m, B_m \rangle$ *satisfies* $\exists z \in Z_m : T_i(z) = \mathcal{T}_t$*;*

  if there exists one. In particular, among all deployment plans satisfying the constraints above, one that minimizes $\sum_{o \in O.(\exists z. N_m(z) = o)} o.\mathtt{cost}$ *(i.e., the overall cost of nodes in the last configuration* $\mathcal{C}_m$*), is outputted.*
- **no** *(stating that no such plan exists); otherwise.*

We are finally ready to state our main result on the decidability of the optimal deployment problem. To prove the result we describe an approach that splits the problem in three incremental phases: (1) the first phase checks if there is a possible solution and assigns microservices to deployment nodes, (2) the intermediate phase computes how the microservices need to be connected to each other, and (3) the final phase synthesizes the corresponding deployment plan.

Theorem 1. *The optimal deployment problem is decidable.*

Proof. The proof is in the form of an algorithm that solves the optimal deployment problem. We assume that the input to the problem to be solved is given by U (the microservice types), O (the set of available nodes), $\mathcal{C}_0$ (the initial provisionally correct configuration), and $\mathcal{T}_t \in U$ (the target microservice type). We use $\mathcal{I}(U)$ to denote the set of interfaces used in the considered microservice types, namely $\mathcal{I}(U) = \bigcup_{\mathcal{T} \in U} \mathtt{dom}(\mathcal{T}.\mathtt{req_s}) \cup \mathtt{dom}(\mathcal{T}.\mathtt{req_w}) \cup \mathtt{dom}(\mathcal{T}.\mathtt{prov}) \cup \mathcal{T}.\mathtt{conf}$. The algorithm is based on three phases.

Phase 1 The first phase consists of the generation of a set of constraints that, once solved, indicates how many instances should be created for each microservice type $\mathcal{T}$ (denoted with $\mathtt{inst}(\mathcal{T})$), how many of them should be deployed on node o (denoted with $\mathtt{inst}(\mathcal{T}, o)$), and how many bindings should be established for each interface p from instances of type $\mathcal{T}$ — considering both weak and strong required interfaces — and instances of type $\mathcal{T}'$ (denoted with $\mathtt{bind}(p, \mathcal{T}, \mathcal{T}')$). We also generate an optimization function that guarantees that the generated configuration is minimal w.r.t. its total cost.

We now incrementally report the generated constraints. The first group of constraints deals with the number of bindings:

$$\bigwedge_{p \in \mathcal{I}(U)} \bigwedge_{\mathcal{T} \in U,\ p \in \mathtt{dom}(\mathcal{T}.\mathtt{req_s})} \mathcal{T}.\mathtt{req_s}(p) \cdot \mathtt{inst}(\mathcal{T}) \leq \sum_{\mathcal{T}' \in U} \mathtt{bind}(p, \mathcal{T}, \mathcal{T}') \tag{1a}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \bigwedge_{\mathcal{T} \in U,\ p \in \mathtt{dom}(\mathcal{T}.\mathtt{req_w})} \mathcal{T}.\mathtt{req_w}(p) \cdot \mathtt{inst}(\mathcal{T}) \leq \sum_{\mathcal{T}' \in U} \mathtt{bind}(p, \mathcal{T}, \mathcal{T}') \tag{1b}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \bigwedge_{\mathcal{T} \in U,\ \mathcal{T}.\mathtt{prov}(p) < \infty} \mathcal{T}.\mathtt{prov}(p) \cdot \mathtt{inst}(\mathcal{T}) \geq \sum_{\mathcal{T}' \in U} \mathtt{bind}(p, \mathcal{T}', \mathcal{T}) \tag{1c}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \bigwedge_{\mathcal{T} \in U,\ \mathcal{T}.\mathtt{prov}(p) = \infty} \mathtt{inst}(\mathcal{T}) = 0 \ \Rightarrow \sum_{\mathcal{T}' \in U} \mathtt{bind}(p, \mathcal{T}', \mathcal{T}) = 0 \tag{1d}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \bigwedge_{\mathcal{T} \in U,\ p \notin \mathtt{dom}(\mathcal{T}.\mathtt{prov})} \sum_{\mathcal{T}' \in U} \mathtt{bind}(p, \mathcal{T}', \mathcal{T}) = 0 \tag{1e}$$

Constraint 1a and 1b guarantee that there are enough bindings to satisfy all the required interfaces, considering both strong and weak requirements. Symmetrically, constraint 1c guarantees that the number of bindings is not greater than the total available capacity, computed as the sum of the single capacities of each provided interface. In case the capacity is unbounded (i.e., ∞), it is sufficient to have at least one instance that activates such port to support any possible requirement (see constraint 1d). Finally, constraint 1e guarantees that no binding is established connected to provided interfaces of microservice types that are not deployed.

The second group of constraints deals with the number of instances of microservices to be deployed.

$$\texttt{inst}(\mathcal{T}_t) \geq 1 \tag{2a}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \bigwedge_{\substack{\mathcal{T} \in U, \\ p \in \mathcal{T}.\texttt{conf}}} \bigwedge_{\substack{\mathcal{T}' \in U - \{\mathcal{T}\}, \\ p \in \texttt{dom}(\mathcal{T}'.\texttt{prov})}} \texttt{inst}(\mathcal{T}) > 0 \;\Rightarrow\; \texttt{inst}(\mathcal{T}') = 0 \tag{2b}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \bigwedge_{\substack{\mathcal{T} \in U,\, p \in \mathcal{T}.\texttt{conf} \wedge \\ p \in \texttt{dom}(\mathcal{T}.\texttt{prov})}} \texttt{inst}(\mathcal{T}) \leq 1 \tag{2c}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \bigwedge_{\mathcal{T} \in U} \bigwedge_{\mathcal{T}' \in U - \{\mathcal{T}\}} \texttt{bind}(p, \mathcal{T}, \mathcal{T}') \leq \texttt{inst}(\mathcal{T}) \cdot \texttt{inst}(\mathcal{T}') \tag{2d}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \bigwedge_{\mathcal{T} \in U} \texttt{bind}(p, \mathcal{T}, \mathcal{T}) \leq \texttt{inst}(\mathcal{T}) \cdot (\texttt{inst}(\mathcal{T}) - 1) \tag{2e}$$

The first constraint 2a guarantees the presence of at least one instance of the target microservice. Constraint 2b guarantees that no two instances of different types will be created if one activates a conflict on an interface provided by the other one. Constraint 2c, consider the other case in which a type activates the same interface both in conflicting and provided modality: in this case, at most one instance of such type can be created. Finally, the constraints 2d and 2e guarantee that there are enough pairs of distinct instances to establish all the necessary bindings. Two distinct constraints are used: the first one deals with bindings between microservices of two different types, the second one with bindings between microservices of the same type.

The last group of constraints deals with the distribution of microservice instances over the available nodes O.

$$\texttt{inst}(\mathcal{T}) = \sum_{o \in O} \texttt{inst}(\mathcal{T}, o) \tag{3a}$$

$$\bigwedge_{r \in \mathcal{R}} \bigwedge_{o \in O} \sum_{\mathcal{T} \in U} \texttt{inst}(\mathcal{T}, o) \cdot \mathcal{T}.\texttt{res}(r) \leq o.\texttt{res}(r) \tag{3b}$$

$$\bigwedge_{o \in O} \Big(\sum_{\mathcal{T} \in U} \texttt{inst}(\mathcal{T}, o) > 0 \Big) \Leftrightarrow \texttt{used}(o) \tag{3c}$$

$$\min \sum_{o \in O,\, \texttt{used}(o)} o.\texttt{cost} \tag{3d}$$

Constraint 3a simply formalizes the relationship among the variables $\texttt{inst}(\mathcal{T})$ and $\texttt{inst}(\mathcal{T}, o)$ (the total amount of all instances of a microservice type, should correspond to the sum of the instances locally deployed on each node). Constraint 3b checks that each node has enough resources to satisfy the requirements of all the hosted microservices. The last two constraints define the optimization function used to minimize the total cost: constraint 3c introduces the boolean variable $\texttt{used}(o)$ which is true if and only if node o contains at least one microservice instance; constraint 3d is the function to be minimized, i.e., the sum of the costs of the used nodes.

These constraints, and the optimization function, are expected to be given in input to a constraint/optimization solver. If a solution is not found it is not possible to deploy the required microservice system; otherwise, the next phases of the algorithm are executed to synthesize the optimal deployment plan.

Phase 2 The second phase consists of the generation of another set of constraints that, once solved, indicates the bindings to be established between any pair of microservices to be deployed. More precisely, for each type $\mathcal{T}$ such that $\texttt{inst}(\mathcal{T}) > 0$, we use $s_i^{\mathcal{T}}$, with $1 \leq i \leq \texttt{inst}(\mathcal{T})$, to identify the microservices of type $\mathcal{T}$ to be deployed. We also assume a function N that associates microservices to available nodes O, which is compliant with the values $\texttt{inst}(\mathcal{T}, o)$ already computed in Phase 1, i.e., given a type $\mathcal{T}$ and a node o, the number of $s_i^{\mathcal{T}}$, with $1 \leq i \leq \texttt{inst}(\mathcal{T})$, such that $N(s_i^{\mathcal{T}}) = o$ coincides with $\texttt{inst}(\mathcal{T}, o)$.

In the constraints below we use the variables $\texttt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'})$ (with $i \neq j$, if $\mathcal{T} = \mathcal{T}'$): its value is 1 if there is a connection between the required interface p of $s_i^{\mathcal{T}}$ and the provided interface p of $s_j^{\mathcal{T}'}$, 0 otherwise. We use n and m to denote $\texttt{inst}(\mathcal{T})$ and $\texttt{inst}(\mathcal{T}')$, respectively, and an auxiliary total function $limProv(\mathcal{T}', p)$ that extends $\mathcal{T}'.\texttt{prov}$ associating 0 to interfaces outside its domain.

$$\bigwedge_{\mathcal{T} \in U} \bigwedge_{p \in \mathcal{I}(U)} \bigwedge_{i \in 1 \ldots n} \sum_{j \in (1 \ldots m) \setminus \{i \mid \mathcal{T} = \mathcal{T}'\}} \texttt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) \leq limProv(\mathcal{T}', p) \tag{4a}$$

$$\bigwedge_{\mathcal{T} \in U} \bigwedge_{p \in \texttt{dom}(\mathcal{T}.\texttt{reqs})} \bigwedge_{i \in 1 \ldots n} \sum_{j \in (1 \ldots m) \setminus \{i \mid \mathcal{T} = \mathcal{T}'\}} \texttt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) \geq \mathcal{T}.\texttt{reqs}(p) \tag{4b}$$

$$\bigwedge_{\mathcal{T} \in U} \bigwedge_{p \in \texttt{dom}(\mathcal{T}.\texttt{reqw})} \bigwedge_{i \in 1 \ldots n} \sum_{j \in (1 \ldots m) \setminus \{i \mid \mathcal{T} = \mathcal{T}'\}} \texttt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) \geq \mathcal{T}.\texttt{reqw}(p) \tag{4c}$$

$$\bigwedge_{\mathcal{T} \in U} \bigwedge_{p \notin \texttt{dom}(\mathcal{T}.\texttt{reqs}) \cup \texttt{dom}(\mathcal{T}.\texttt{reqw})} \bigwedge_{i \in 1 \ldots n} \sum_{j \in (1 \ldots m) \setminus \{i \mid \mathcal{T} = \mathcal{T}'\}} \texttt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) = 0 \tag{4d}$$

Constraint 4a considers the provided interface capacities to fix upper bounds to the bindings to be established, while constraints 4b and 4c fix lower bounds based on the required interface capacities, considering both the weak (see 4b) and the strong (see 4c) ones. Finally, constraint 4d indicates that it is not possible to establish connections on interfaces that are not required.

A solution for these constraints exists because, as also shown in [13], the constraints 1a ... 2e (already solved during Phase 1) guarantee that the config-

uration to be synthesized contains enough capacity on the provided interfaces to satisfy all the required interfaces.

Phase 3 In this last phase we synthesize the deployment plan that, when applied to the initial configuration $\mathcal{C}_0$, reaches a new configuration $\mathcal{C}_t$ with nodes, microservices and bindings as computed in the first two phases of the algorithm. Without loss of generality, in this decidability proof we show the existence of a simple plan that first removes the elements in the initial configuration and then deploys the target configuration from scratch. However, as also discussed in Sect. 3, in practice it is possible to define more complex planning mechanisms that re-use microservices already deployed.

Reaching an empty configuration is a trivial task since it is always possible to perform in the initial configuration unbind actions for all the bindings connected to weak required interfaces. Then, the microservices can be safely deleted. Thanks to the well-formedness assumption (Definition 2) and using a topological sort, it is possible to order the microservices to be removed without violating any strong required interface (e.g., first remove the microservice not requiring anything and repeat until all the microservices have been deleted).

The deployment of the target configuration follows a similar pattern. Given the distribution of microservices over nodes (computed in the first phase) and the corresponding bindings (computed in the second phase), the microservices can be created by following a topological sort considering the microservices dependencies following from the strong required interfaces. When all the microservices are deployed on the corresponding nodes, the remaining bindings (on weak required ports) may be added in any possible order. □

Remark 1. The constraints generated during Phase 2 of the algorithm, in order to establish the microservice bindings, are expected to be given in input to a constraint/optimization solver. One can enrich such constraints with metrics to optimize, e.g., the number of local bindings (i.e., give a preference to the connections among microservices hosted in the same node):

$$\min \sum_{\mathcal{T},\mathcal{T}' \in U, i \in 1 \ldots \mathtt{inst}(\mathcal{T}), j \in 1 \ldots \mathtt{inst}(\mathcal{T}'), p \in \mathcal{I}(U), N(s_i^{\mathcal{T}}) \neq N(s_j^{\mathcal{T}'})} \mathtt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'})$$

Another example, used in the case study discussed in Sect. 3, is the following metric that maximizes the number of bindings[4]:

$$\max \sum_{s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}, p \in \mathcal{I}(U)} \mathtt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'})$$

From the complexity point of view, it is possible to show that the decision versions of the optimization problem solved in Phase 1 is NP-complete, in Phase

[4] We model a load balancer as a microservice having a weak required interface, with arity 0, that can be provided by its back-end service. By adopting the above maximization metric, the synthesized configuration connects all possible services to such required interface, thus allowing the load balancer to forward requests to all of them.

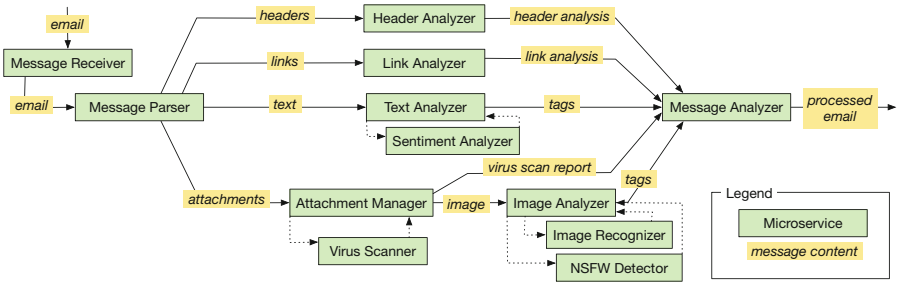


Fig. 2. Microservice architecture for email processing.

2 is in NP, while the planning in Phase 3 is synthesized in polynomial time. Unfortunately, due to the fact that numeric constraints can be represented in log space, the output of Phase 2 requiring the enumeration of all the microservices to deploy can be exponential in the size of the output of Phase 1 (indicating only the total number of instances for each type). For this reason, the optimal deployment problem is in NEXPTIME. However, we consider unfeasible in practice the deployment of an exponential number of microservices on one node having limited resources. If at most a polynomial number of microservices can be deployed on each node, we have that the optimal deployment problem becomes an NP-optimization problem and its decision version is NP-complete. See the companion technical report [8] for the formal proofs of complexity.

3 Application of the Technique to the Case-Study

Given the asymptotic complexity of our solution (NP under the assumption of polynomial size of the target configuration) we have decided to evaluate its applicability in practice by considering a real-world microservice architecture, namely the email processing pipeline described in [22]. The considered architecture separates and routes the components found in an email (headers, links, text, attachments) into distinct, parallel sub-pipelines with specific tasks (e.g., remove malicious attachments, tag the content of the mail). We report in Fig. 2 a depiction of the architecture. When an email reaches the Message Receiver it is forwarded to the Message Parser, which sends each component into a specific sub-pipeline. In the sub-pipelines, some microservices — e.g., Text Analyzer and Attachment Analyzer — coordinate with other microservices — e.g., Sentiment Analyzer and Virus Scanner — to process their inputs. Each microservice in the architecture has a given resource consumption (expressed in terms of CPU and memory). As expected, the processing of each email component entails a specific load. Some microservices can handle large inputs, e.g., in the range of 40K simultaneous requests (e.g., Header Analyzer that processes short and uniform inputs). Other microservices sustain heavier computations (e.g., Image Recognizer) and can handle smaller simultaneous inputs, e.g., in the range of 10K requests.

To model the system above, we use the Abstract Behavioral Specification (ABS) language, a high-level object-oriented language that supports deployment modeling [31]. ABS is agnostic w.r.t. deployment platforms (Amazon AWS, Microsoft Azure) and technologies (e.g., Docker or Kubernetes) and it offers high-level deployment primitives for the creation of new *deployment components* and the instantiation of objects inside them. Here, we use ABS deployment components as computation nodes, ABS objects as microservice instances, and ABS object references as bindings. Finally, to describe the requirements in our model, we use ABS with SmartDepl [25], an extension that supports deployment annotations. Strong required interfaces are modeled as class annotations indicating mandatory parameters for the class constructor: such parameters contain the references to the objects corresponding to the microservices providing the strongly required interfaces. Weak required interfaces are expressed as annotations concerning specific methods used to pass, to an already instantiated object, the references to the objects providing the weakly required interfaces. We define a class for each microservice type, plus one *load balancer* class for each microservice type. A load balancer distributes requests over a set of instances that can scale horizontally. Finally, we model nodes corresponding to Amazon EC2 instances: `c4_large`, `c4_xlarge`, and `c4_2xlarge` (with the corresponding provided resources and costs).

Microservice (max computational load)	**Initial (10K)**	**+20K**	**+50K**	**+80K**
MessageReceiver(∞)	1	-	-	-
MessageParser(40K)	1	-	+1	-
HeaderAnalyzer(40K)	1	-	+1	-
LinkAnalyzer(40K)	1	-	+1	-
TextAnalyzer(15K)	1	+1	+2	+2
SentimentAnalyzer(15K)	1	+3	+4	+6
AttachmentsManager(30K)	1	+1	+2	+2
VirusScanner(13K)	1	+3	+4	+6
ImageAnalyzer(30K)	1	+1	+2	+2
NSFWDetector(13K)	1	+3	+4	+6
ImageRecognizer(13K)	1	+3	+4	+6
MessageAnalyzer(70K)	1	+1	+2	+2

In the table above, we report the result of our algorithm w.r.t. four incremental deployments: the initial in column 2 and under incremental loads in 3–5. We also consider an availability of 40 nodes for each of the three node types. In the first column of the Table, next to a microservice type, we report its corresponding maximum computational load, i.e., the maximal number of simultaneous requests that it can manage. As visible in columns 2–5, different maximal computational loads imply different scaling factors w.r.t. a given

number of simultaneous requests. In the initial configuration we consider 10K simultaneous requests and we have one instance of each microservice type (and of the corresponding load balancer). The other deployment configurations deal with three scenarios of horizontal scaling, assuming three increasing increments of inbound messages (20K, 50K, and 80K). In the three scaling scenarios, we do not implement the planning algorithm described in Phase 3 of the proof of Theorem 1. Contrarily, we take advantage of the presence of the load balancers and, as described in Remark 1, we achieve a similar result with an optimization function that maximizes the number of bindings of the load balancers. For every scenario, we use SmartDepl [33] to generate the ABS code for the plan that deploys an optimal configuration, setting a timeout of 30 min for the computation of every deployment scenario.[5] The ABS code modeling the system and the generated code are publicly available at [7]. A graphical representation of the initial configuration is available in the companion technical report [8].

4 Related Work and Conclusion

In this work, we consider a fundamental building block of modern Cloud systems, microservices, and prove that the generation of a deployment plan for an architecture of microservices is decidable and fully automatable; spanning from the synthesis of the optimal configuration to the generation of the deployment actions. To illustrate our technique, we model a real-world microservice architecture in the ABS [31] language and we compute a set of deployment plans.

The context of our work regards automating Cloud application deployment, for which there exist many specification languages [5,11], reconfiguration protocols [6,19], and system management tools [26,32,37,38]. Those tools support the specification of deployment plans but they do not support the automatic distribution of software instances over the available machines. The proposals closest to ours are those by Feinerer [20] and by Fischer et al. [21]. Both proposals rely on a solver to plan deployments. The first is based on the UML component model, which includes conflicts and dependencies, but lacks the modeling of nodes. The second does not support conflicts in the specification language. Neither proposals support the computation of optimal deployments.

Three projects inspire our proposal: Aeolus [13,14], Zephyrus [1], and ConfSolve [28]. The Aeolus model paved the way to reason on deployment and reconfiguration, proving some decidability results. Zephyrus is a configuration tool based on Aeolus and it constitutes the first phase of our approach. ConfSolve is a tool for the optimal allocation of virtual machines to servers and of applications to virtual machines. Both tools do not synthesize deployment plans.

[5] Here, 30 min are a reasonable timeout since we predict different system loads and we compute in advance a different deployment plan for each of them. An interesting future work would aim at shortening the computation to a few minutes (e.g., around the average start-up time of a virtual machine in a public Cloud) to obtain on-the-fly deployment plans tailored to unpredictable system loads.

Regarding autoscaling, existing solutions [2,4,17,29] support the automatic increase or decrease of the number of instances of a service/container, when some conditions (e.g., CPU average load greater than 80%) are met. Our work is an example of how we can go beyond single-component horizontal scaling policies (as analyzed, e.g., in [9]).

As future work, we want to investigate local search approaches to speed-up the solution of the optimization problems behind the computation of a deployment plan. Shorter computation times would open our approach to contexts where it is unfeasible to compute plans ahead of time, e.g., due to unpredictable loads.

References

1. Ábrahám, E., Corzilius, F., Johnsen, E.B., Kremer, G., Mauro, J.: Zephyrus2: on the fly deployment optimization using SMT and CP technologies. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) SETTA 2016. LNCS, vol. 9984, pp. 229–245. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47677-3_15
2. Amazon: Amazon CloudWatch. https://aws.amazon.com/cloudwatch/. Accessed Jan 2019
3. Amazon: AWS auto scaling. https://aws.amazon.com/autoscaling/. Accessed Jan 2019
4. Apache: Apache Mesos. http://mesos.apache.org/. Accessed Jan 2019
5. Bergmayr, A., et al.: A systematic review of cloud modeling languages. ACM Comput. Surv. **51**(1), 22:1–22:38 (2018)
6. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies. In: ICSE, pp. 13–22. IEEE Computer Society (2013)
7. Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: Code repository for the email processing example. https://github.com/IacopoTalevi/SmartDeploy-ABS-ExampleCode. Accessed Jan 2019
8. Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: Optimal and automated deployment for microservices. Technical Report (2019). https://arxiv.org/abs/1901.09782
9. Bravetti, M., Gilmore, S., Guidi, C., Tribastone, M.: Replicating web services for scalability. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 204–221. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78663-4_15
10. Brogi, A., Canciani, A., Soldani, J.: Modelling and analysing cloud application management. In: Dustdar, S., Leymann, F., Villari, M. (eds.) ESOCC 2015. LNCS, vol. 9306, pp. 19–33. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24072-5_2
11. Chardet, M., Coullon, H., Pertin, D., Pérez, C.: Madeus: a formal deployment model. In: HPCS, pp. 724–731. IEEE (2018)
12. Chuffed Team: The CP solver. https://github.com/geoffchu/chuffed. Accessed Jan 2019
13. Di Cosmo, R., Lienhardt, M., Mauro, J., Zacchiroli, S., Zavattaro, G., Zwolakowski, J.: Automatic application deployment in the cloud: from practice to theory and back (invited paper). In: CONCUR. LIPIcs, vol. 42, pp. 1–16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)

14. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a component model for the cloud. Inf. Comput. **239**, 100–121 (2014)
15. Di Cosmo, R., Zacchiroli, S., Zavattaro, G.: Towards a formal component model for the cloud. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 156–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_11
16. Docker: Docker compose documentation. https://docs.docker.com/compose/. Accessed Jan 2019
17. Docker: Docker swarm. https://docs.docker.com/engine/swarm/. Accessed Jan 2019
18. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
19. Durán, F., Salaün, G.: Robust and reliable reconfiguration of cloud applications. J. Syst. Softw. **122**, 524–537 (2016)
20. Feinerer, I.: Efficient large-scale configuration via integer linear programming. AI EDAM **27**(1), 37–49 (2013)
21. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: a deployment management system. In: PLDI (2012)
22. Fromm, K.: Thinking Serverless! How New Approaches Address Modern Data Processing Needs. https://read.acloud.guru/thinking-serverless-how-new-approaches-address-modern-data-processing-needs-part-1-af6a158a3af1. Accessed Jan 2019
23. GECODE: an open, free, efficient constraint solving toolkit. http://www.gecode.org. Accessed Jan 2019
24. Google: Optimization tools. https://developers.google.com/optimization/. Accessed Jan 2019
25. de Gouw, S., Mauro, J., Nobakht, B., Zavattaro, G.: Declarative elasticity in ABS. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) ESOCC 2016. LNCS, vol. 9846, pp. 118–134. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44482-6_8
26. Red Hat Ansible. https://www.ansible.com/. Accessed Jan 2019
27. Hellerstein, J.M., et al.: Serverless computing: one step forward, two steps back. arXiv preprint arXiv:1812.03651 (2018)
28. Hewson, J.A., Anderson, P., Gordon, A.D.: A declarative approach to automated configuration. In: LISA (2012)
29. Hightower, K., Burns, B., Beda, J.: Kubernetes: Up and Running Dive into the Future of Infrastructure, 1st edn. O'Reilly Media, Inc., Sebastopol (2017)
30. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional, Boston (2010)
31. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
32. Kanies, L.: Puppet: next-generation configuration management. `;login`: USENIX Mag. **31**(1), 19–25 (2006)
33. Mauro, J.: Smartdepl. https://github.com/jacopoMauro/abs_deployer. Accessed Jan 2019

34. Mauro, J., Zavattaro, G.: On the complexity of reconfiguration in systems with legacy components. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9234, pp. 382–393. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48057-1_30
35. Mccombs, S.: Outages? Downtime? https://sethmccombs.github.io/work/2018/12/03/Outages.html. Accessed Jan 2019
36. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
37. Opscode: Chef. https://www.chef.io/chef/. Accessed Jan 2019
38. Puppet Labs: Marionette collective. http://docs.puppetlabs.com/mcollective/. Accessed Jan 2019
39. Woods, D.: On infrastructure at scale: a cascading failure of distributed systems. https://medium.com/@daniel.p.woods/on-infrastructure-at-scale-a-cascading-failure-of-distributed-systems-7cff2a3cd2df. Accessed Jan 2019

A Data Flow Model with Frequency Arithmetic

Paul Dubrulle(✉), Christophe Gaston, Nikolai Kosmatov, Arnault Lapitre, and Stéphane Louise

CEA, List, 91191 Gif-sur-Yvette, France
{paul.dubrulle,christophe.gaston,nikolai.kosmatov,
arnault.lapitre,stephane.louise}@cea.fr

Abstract. Data flow formalisms are commonly used to model systems in order to solve problems of buffer sizing and task scheduling. A prerequisite for static analysis of a modeled system is the existence of a periodic schedule in which the sizes of communication channels can be bounded for an unbounded execution (consistency), and that communication dependencies do not introduce a deadlock in such an execution (liveness). In the context of Cyber-Physical Systems, components are often interfaced with the physical world and have frequency constraints. The existing data flow formalisms lack expressiveness to fully cover the expected behavior of these components. We propose an extension to Synchronous Data Flow (SDF) formalism, called Polygraph, that includes frequency constraints and adjustable communication rates. We show that with these extensions, the conditions for a model to be consistent and live are no longer sufficient, and we extend the corresponding theorems with necessary and sufficient conditions to preserve these properties. We also introduce a framework to check the liveness of a Polygraph model, implemented in the tool DIVERSITY, along with preliminary experiments to validate this approach.

1 Introduction

Context. Cyber-Physical Systems (CPS) are increasingly present in everyday life. In these systems, the components require a certain amount of input data to produce a known amount of output data, and some of them must do so in synchrony with a reference time scale. For example, the next generation of autonomous vehicles will heavily rely on sensor fusion systems to operate the car. Sensors and actuators have specified frequencies. To produce its output, the fusion kernel requires a certain number of samples from several sources, with a temporal correlation between them.

Often, when implementing this kind of system, the prediction of its performance is important to the system designer. The performance prediction covers different characteristics of the system, including its throughput, memory footprint, and latency. In distributed implementations of such systems, an analysis of

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 369–385, 2019.
https://doi.org/10.1007/978-3-030-16722-6_22

the communications between the components is necessary to configure a network capable to respect the application's real-time requirements.

Data flow formalisms [3,14] can be used to perform this kind of performance analysis [4,5,10–12]. A prerequisite to analyze a model is the existence of a periodic schedule with two properties. The first property, *consistency*, requires that the sizes of the communication buffers remain bounded for an unbounded execution of the periodic schedule. In practice, if a model is not consistent, it is not possible to implement the communications without losing data samples. The second property, *liveness*, requires the absence of deadlocks in the schedule.

Motivation and Goals. The limitation of the existing data flow formalisms to model the considered systems is the lack of expressiveness regarding the synchronization on a common time scale for different components. Overcoming this limitation is the subject of recent research work [6]. Our goal is to extend an existing data flow formalism for which the consistency and liveness properties of a given model are decidable. In doing so, we want to ensure that the expressiveness extension does not impact the decidability of these properties. With this extension, all applicative constraints are taken into account when checking the prerequisites for a performance analysis. The verification can be performed in abstraction of a particular implementation's characteristics (like execution times or mapping), and the results are the same for different implementations. Moreover, the performance analysis can benefit from the additional information on the system provided by the extension.

Approach and Main Results. This paper introduces Polygraph, an extension to Synchronous Data Flow (SDF) [14] for specification of frequency constraints on the components. We use an arithmetic based on rational numbers to reason on data exchanges between components. We show that the theorems that provide a theoretical foundation for practical verification of consistency and liveness for an SDF model can be generalized to this new formalism. Finally, we propose a symbolic execution framework to decide the liveness of models expressed in Polygraph, in a way similar to [11,14].

The contributions of this work include:

- a data flow formalism, called Polygraph, extending the well-known SDF [14] formalism, to support the synchronization of data production and consumption on a reference time scale;
- a demonstration that the decidability of two classical properties of dataflow models, namely consistency and liveness, is preserved for this new formalism;
- an adaptation to the new formalism of an existing symbolic execution technique for evaluation of liveness in the DIVERSITY tool and initial experiments to validate this approach.

Outline. The remainder of this paper is organized as follows. Section 2 gives an informal introduction to the proposed modeling approach, with a step-by-step explanation relying on an illustrative system. In Sect. 3, we formalize Polygraph

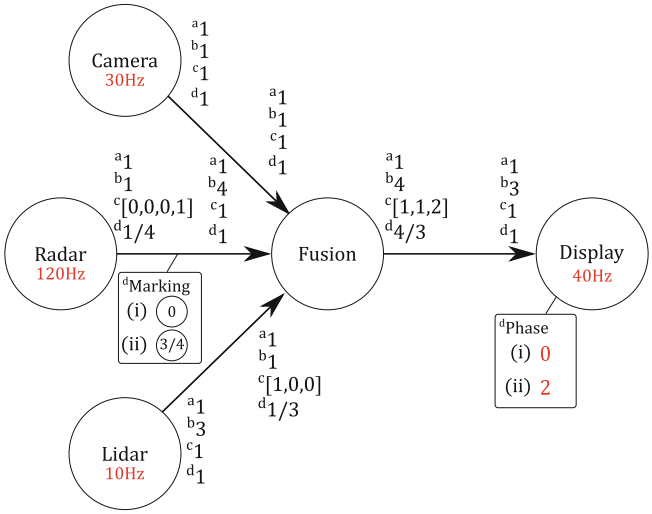


Fig. 1. Motivating example: a data fusion system modeled as a data flow graph. The upper indexes "a" to "d" denote an amount of data exchanged by the components in different variants of the model. The rates denoted by upper index "d" are those of Polygraph, and initial conditions for this configuration are denoted by (i) and (ii).

and provide extended statements and a sketch of proof for the consistency and liveness theorems. Section 4 presents a framework to check the liveness property for Polygraph and a preliminary evaluation. In Sect. 5, we discuss related work, while Sect. 6 presents conclusion and perspectives.

2 Motivation and Running Example

Running Example. To introduce the modeling approach behind Polygraph, we use a toy example of a data fusion system that could be integrated into the cockpit display of a car, depicted in Fig. 1. The system is composed of three sensors producing data samples to be used by a data fusion component, and a display component. The function of the sensor components is to read the data from their sensors, while the function of the data fusion component is to compute a result based on this data. The function of the display component is to render the fusion result on a screen. To do so, the sensor components send the data to the fusion component, and the fusion component sends the result to the display component. The first sensor component is a video camera producing frames. The other two sensor components analyze radar and lidar based samples to produce a descriptor of the closest detected obstacles. The fusion component uses this information to draw the obstacle descriptors on the corresponding frame.

The first step to model this system is to build a graph capturing data dependencies between the components. Each vertex of this graph models an *actor*, an abstract entity representing the function of a component. Each directed edge of

the graph models a communication *channel*, the source actor being the producer of data consumed by the destination actor. The structure of the graph in Fig. 1 illustrates the dependencies in our example. The communication policy on the channels is First-In First-Out (FIFO), the write operation is non-blocking, and the read operation is blocking. On each channel, the atomic amount of data exchanged by the connected actors is called a *token*, and all write and read operations are measured in tokens. An actor *produces* (resp. *consumes*) a certain number of tokens on a channel when it writes (resp. reads) the corresponding amount of data. With this policy, the graph can be assimilated to a Kahn Process Network (KPN) [13]. In a KPN, the communications are determinate, but in general it is not possible to decide if the sizes of the channels can be bounded for an unbounded execution of the system.

Synchronous and Asynchronous Constraints. In practice, sensors and actuators have a fixed sampling rate, and the production of each data sample occurs at that specified frequency. To model these constraints, we propose to label some actors with *frequencies*, corresponding to the real-life constraint. An actor with a frequency label must *fire* at that frequency. We further detail this notion of firing below, but for now it is sufficient to say that the firing of an actor is an atomic process, during which it performs the actions and communications expected from the modeled component. A global clock provides ticks to synchronize the firing of frequency labeled actors. For our example, we consider the frequency labeling illustrated by Fig. 1.

Generally, in real-life systems, computation kernels compute when input data is available and do not have frequency constraints. In our frequency labeling, the actors modeling such components can be left without a frequency label. In our example, this is the case for the fusion actor.

The possibility to have unlabeled actors is an important part of our approach, as further discussed in Sect. 5. It allows to mix a synchronous firing policy for labeled actors, and an asynchronous firing policy for unlabeled actors. This means that the scheduling of firings has periodic constraints only where needed, which offers more options for optimization algorithms.

Static Rates. Another characteristic of real-life software components in our context is that they require a fixed number of input samples from each different source. Also, there must be a correlation between the production time of the samples consumed from different sources. In our example, the fusion component requires one token from each sensor, and these samples must have a close-enough production time. This constraint can be captured by KPN restrictions, such as Synchronous Data Flow (SDF) [14]. In SDF, both ends of each channel are assigned a communication rate, denoting the fixed number of tokens produced or consumed by the connected actors' firings. This characteristic allows to decide whether the sizes of the channels are bounded for an unbounded execution. Graphs respecting this property are said to be *consistent*.

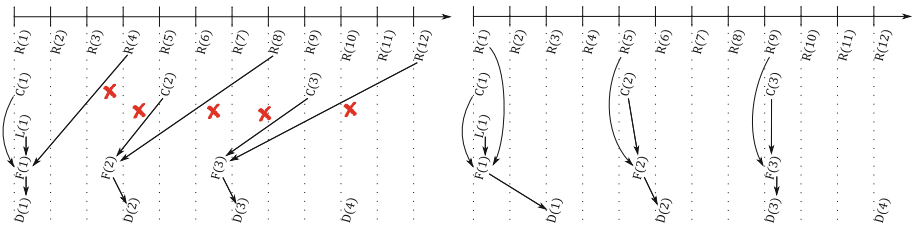
Without taking frequencies into account, the communication rates denoted by an upper index "a" in Fig. 1 match the description of the system. Indeed, the

sensor actors produce one token each, the fusion actor consumes these tokens, and in turn produces one token to be consumed by the display actor. With these rates, considering a marking of the graph with any number of tokens stored in the channels, if firing all the actors once, the same number of tokens remains in the channels. Hence, the SDF graph is consistent. But when taking frequencies into account, the graph is no longer consistent. In this example, the camera produces 30 tokens per second, the radar produces 120 tokens per second, and the lidar produces 10 tokens per second. This means that per second, because of the production rate and frequency of the lidar, the fusion actor will be able to fire only 10 times. It will consume only 10 tokens from the camera and radar actors, leaving 20 and 110 unconsumed tokens per second on their respective channels. Hence, it is no longer possible to bound the size of these channels for an unbounded execution of the graph. This shows that to achieve consistency, for any frequency labeled actor, the number of asynchronous firings of its unlabeled predecessors and successors should be limited.

A possible adaptation of communication rates, denoted by upper index "b" in Fig. 1, takes frequency inheritance into account and restores the consistency property. With the production and consumption rates both set to 1 on the channel connecting the camera and the fusion actors, the fusion actor basically inherits a frequency constraint of 30 Hz. It inherits the same frequency constraint from the radar and lidar actors since it now consumes $4 \times 30 = 1 \times 120$ tokens per second from the radar, and $1 \times 30 = 3 \times 10$ tokens per second from the lidar. The rates on the channel connecting the fusion and display actors are also balanced. But with these rates, the number of tokens does not reflect accurately the expected behavior of the modeled components. For example, the fusion actor would consume 4 tokens per activation from the radar actor, while in reality the component only requires 1.

Cyclo-Static Rates. It is possible to use Cyclo-Static Data Flow (CSDF) [3] to get closer to the real communication requirements. In CSDF, the rates of the actors are fixed as in SDF, but the successive firings of an actor cyclically consume and produce a different number of tokens on every connected channel. The successive rates on each channel are expressed as a sequence of natural numbers. For example, an actor with a cyclo-static sequence of output rates $[1, 2]$ produces 1 token for its first firing, 2 tokens for the second, 1 for the third and so on. A zero rate may occur in the sequence, meaning that the actor does not push or pull tokens on the channel for the corresponding firing.

A cyclo-static sequence is necessary on a channel if the connected actors have frequency constraints conflicting with the expected communication behavior. In this case, we propose that one of the actors must be chosen as having the reference frequency for the communication, and the other actor must adapt its communication rate to a cyclo-static sequence accordingly. Back to our example (see variant "c" in Fig. 1), the fusion actor requires one token from each sensor every firing. Since the component is synchronized on camera frames, we decide that the actor's reference frequency should be 30 Hz. In this case, the frequency constraints do not conflict with the expected communication behavior, and we



(a) Initial conditions (i). (b) Initial conditions (ii).

Fig. 2. Firings of actors of the motivating example: the firings are identified by the initial letter of the corresponding actor and the rank of the firing, arrows show data dependencies between firings, and a reference time scale constrains the firing of timed actors. The data dependencies marked by a cross in (a) introduce a causality issue.

assign production and consumption rates of 1 on the channel connecting the fusion and camera actors. Now, considering the radar actor, the fusion actor only requires 30 tokens per second out of 120. Considering this ratio, we assign the sequence $[0, 0, 0, 1]$ as production rates for the radar actor, and the rate 1 for the fusion actor. The same logic applies for the lidar actor, the fusion actor requires 30 tokens per second, but only 10 tokens per second are produced. We then assign the cyclo-static sequence $[1, 0, 0]$ as consumption rates for the fusion actor, and the rate 1 for the lidar actor. A similar logic is applied for the display actor. The consequence on the stream of actual data values highly depends on the implemented function, and is therefore out of the scope of the data flow modeling. In the particular case of the radar actor in our example, the software implementation could perform a downsampling of the sensed data, or just send the latest sample.

The corresponding communication rates, denoted by upper index "c" in Fig. 1, give a graph where only the required tokens are exchanged on the channels, and the consistency property is preserved. But in all generality, choosing the appropriate cyclic rate sequences for all the channels in a graph is time consuming and error prone.

Rational Rates. We propose instead to extend the SDF model with rational communication rates. A rational communication rate $r = p/q$ specifies that the actor produces or consumes p tokens every q firings, and the natural number of tokens produced or consumed by any firing is r rounded either up or down, denoted $\lceil r \rceil$ and $\lfloor r \rfloor$ respectively. With the semantic formalized in the next section, there is a unique default cyclo-static sequence that corresponds to a given rational rate. The default sequences for the rates denoted by an upper index "d" in Fig. 1 are those denoted by upper index "c". As explained earlier when assigning cyclo-static sequences, in this extension, only one rate on a given channel can be a rational number with denominator greater than one. The methodology remains the same, for any channel, one actor's frequency is considered as a reference, and the other one adapts its rates according to that reference.

Initial Conditions. With the frequency labeling and rational communication rates, we obtain a model that describes as closely as possible the communication and timing requirements of our illustrative example. But there are causality issues in this model. Figure 2(a) illustrates the timing of actor firings in our example, and the data dependencies between them, according to the semantic defined in the next section. It is obvious that the data dependencies marked by a cross are not satisfied in time.

This kind of causality issue can also appear in SDF: in the case of cyclic graphs, the firings of the actors in a cycle all depend on each other. To prevent this, it is possible to *mark* the channels with an initial number of tokens, allowing sufficient initial firings to complete the firing of all actors in the cycle. The liveness property of an SDF graph is verified when all the cycles in the graph are marked with enough tokens to prevent a deadlock [14]. With the SDF extensions we propose, this condition is no longer sufficient. We need to be able to shift the production or consumption of tokens in order to make sure that when a firing requires input tokens, they are produced at an earlier tick of the global clock.

One way to achieve this is to rotate the default sequences defined by the rational rates. For this, we propose a rational initial marking of the graph. Each channel with natural rates at both ends can be marked with an initial number of tokens as in SDF. Each other channel with rational rate $r = p/q$ on either end can be initially marked with a rational number $n + k/q$ with $k < q$, which denotes that the channel initially holds n tokens (as in SDF), and the default sequence is rotated by k. If the rational rate is on the producer, the default sequence is rotated left, otherwise it is rotated right. In Fig. 1, considering the default sequences denoted by "c", the corresponding rational rates denoted by upper index "d", and the initial marking (ii), the marking of 3/4 on the channel connecting the radar and fusion actors rotates the default sequence $[0, 0, 0, 1]$ by 3 elements to the right, yielding the sequence $[1, 0, 0, 0]$.

Another way to prevent unsatisfied data dependencies is to shift the first tick on which a frequency labeled actor must fire. We propose to add a *phase* to each of these actors, giving the offset from the first tick at which it must fire. With the semantic formalized in the next section, that phase is constrained in order to have a periodic global clock. Figure 2(b) takes into account the marking and phase denoted (ii) in Fig. 1. With the rational marking, the dependencies between the radar and fusion firings are now satisfied, and with the phase on the display actor, the dependencies between the camera and display firings are also satisfied.

3 Formalization of the Polygraph Model

We denote by $\mathbb{B}$ the set $\{0, 1\}$, by $\mathbb{Z}$ the set of integers, by $\mathbb{N} = \{n \in \mathbb{Z} \mid n \geqslant 0\}$ the set of natural integers, and by $\mathbb{Q}$ the set of rational numbers. For any set S, the free semigroup on S is denoted S^+.

System graph. A *system graph* is a structure used to represent the topology of the communications. Formally, it is a connected finite directed graph $G = (V, E)$

with set of vertices V and set of edges $E \subseteq V \times V$ such that V is the set of *actors* and E is the set of *channels*. We use an index notation to identify elements with respect to a given actor or channel, considering that E and V are sets indexed respectively in $\{1, \cdots, |E|\}$ and $\{1, \cdots, |V|\}$. We denote v_i (resp. e_j) the actor (resp. channel) of index i (resp. j). For an actor $v \in V$, let $\text{in}(v) = \{\langle v', v\rangle \in E \,|\, v' \in V\}$ denote the set of *input channels* of v and $\text{out}(v) = \{\langle v, v'\rangle \in E \,|\, v' \in V\}$ the set of *output channels* of v.

Topology matrix and channel states. As for SDF and its derivations [3,14], the communication rates are defined by a topology matrix with one row per channel and one column per actor. The only difference in this definition is that we rely on rational numbers. The absolute value of a rate in the matrix defines how many tokens are produced or consumed per firing of the corresponding actor on the corresponding channel, and the sign of that rate indicates if the tokens are produced (positive rate) or consumed (negative rate). For a given actor and channel, the rate must be 0 if the actor is not connected to the channel, or if the actor is connected to both ends of the channel.

Definition 1 (Topology matrix). *A matrix $\mathbf{\Gamma} = (\gamma_{ij}) \in \mathbb{Q}^{|E|\times|V|}$ is a* topology matrix *of a system graph G if for every channel $e_i = \langle v_j, v_k\rangle \in E$ we have:*

- *$\gamma_{il} = 0$ for all $l \neq j, k$;*
- *if $j \neq k$, then $\gamma_{ij} > 0$ and $\gamma_{ik} < 0$ are irreducible fractions, and at most one of them has a denominator greater than 1;*
- *if $j = k$, then $\gamma_{ij} = 0$.*

We also use a rational number per channel to track the communication state of the system during an execution. A channel state is a vector with one row per channel. Each coordinate in the vector tracks the respective number of firings of the connected actors, by addition of their rates when they fire, and that coordinate rounded down is the number of tokens in the channel.

Definition 2 (Channel state). *A vector $\mathbf{c} \in \mathbb{Q}^{|E|\times 1}$ is a* channel state *of a system graph G with topology matrix $\mathbf{\Gamma}$ if for every channel $e_i = \langle v_j, v_k\rangle \in E$, the denominator of c_i is the maximum between the denominators of γ_{ij} and γ_{ik}, and $\lfloor c_i \rfloor$ is the number of tokens in the channel. We denote $C \subseteq \mathbb{Q}^{|E|\times 1}$ the set of all these possible states.*

Timed actors and global clock. A subset $V_F \subseteq V$ of *timed actors* are constrained by a *frequency*, expressed as a strictly positive natural number. We use a frequency mapping $\omega : V_F \longrightarrow \mathbb{N}^{>0}$ in order to map the timed actors to their frequency. There is an implicit system time unit, and each timed actor $v_i \in V_F$ is supposed to be fired exactly $\omega_i := \omega(v_i)$ times per system time unit. In order to have a minimal system time unit, we consider that the greatest common divisor of all the frequencies is $\gcd(\omega[V_F]) = 1$. This is not limiting, since any set of frequencies and system time unit can be adjusted to fit this constraint.

In addition, the timed actors must fire synchronously with respect to a global clock. The *resolution* of that global clock is a sufficient number of *ticks* per system

time unit to associate to each tick the set of timed actors that must fire at the corresponding date. For this, we consider the ticks $0, 1, \ldots, \pi - 1$ per system time unit, where π is the least common multiple of all the actor frequencies $\pi = \text{lcm}(\{\omega_i | v_i \in V_F\})$. Note that if V_F is empty, $\pi = 1$, and the global clock does not constrain the firing of any actor.

Given a timed actor $v_i \in V_F$, there should be ω_i out of π ticks associated with that actor's firings. To reflect the periodic nature of the firing of timed actors, for a timed actor v_i of period $p_i = \pi/\omega_i$, it fires every p_i-th tick.

As mentioned in Sect. 2, all the timed actors have a *phase*. We use a phase mapping $\varphi : V_F \longrightarrow \mathbb{N}$ to map the timed actors to their phase. The first firing of each timed actor $v_i \in V_F$ occurs at the tick $\varphi_i := \varphi(v_i)$. The only constraint to respect the expected frequency of the firings is that $\forall v_i \in V_F$ we have $0 \leqslant \varphi_i < \pi/\omega_i$.

Definition 3 (Global clock, firing ticks). *For a system graph G with frequency mapping ω, resolution π, and phase mapping φ, the* global clock *is a set* $\mathrm{T} = \{0, 1, \ldots, \pi - 1\}$ *and for each timed actor $v_i \in V_F$ there is a subset of* firing ticks $\mathrm{T}_i = \{\tau \in \mathrm{T} \,|\, \tau \equiv \varphi_i \pmod{\pi/\omega_i}\}$.

Polygraphs. We now define the notion of *polygraph* which introduces a basic communication topology, a topology matrix, a frequency and phase mapping for all timed actors, and an initial marking of the graph.

Definition 4 (Polygraph, initial marking). *A* polygraph *is a tuple* $\mathcal{P} = \langle G, \mathbf{\Gamma}, \omega, \varphi, \mathbf{m} \rangle$ *where G is a system graph, $\mathbf{\Gamma}$ is a topology matrix, ω is a frequency mapping, φ is a phase mapping and $\mathbf{m} \in C$ is an* initial marking *such that $\forall e_i \in E$ we have $m_i \geqslant 0$.*

In the following, we consider that a polygraph $\mathcal{P} = \langle G, \mathbf{\Gamma}, \omega, \varphi, \mathbf{m} \rangle$ is given, with its global clock T and sets of firing ticks T_i for all the timed actors $v_i \in V_F$.

States and transitions. The state of a polygraph is composed of a channel state, the current tick of the global clock, and a vector with one row per actor used to track the number of firings of the timed actors since the last change in the current tick. This *tracking vector* is used to check that the timed actors respect their synchronous firing constraints.

Definition 5 (State). *A* state *of a polygraph $\mathcal{P}$ is a tuple $s = \langle \mathbf{c}, \tau, \mathbf{a} \rangle$ where $\mathbf{c} \in C$ is a channel state, $\tau \in \mathrm{T}$ is a tick, and $\mathbf{a} \in \mathbb{N}^{|V| \times 1}$ is a tracking vector. We denote $S \subseteq C \times \mathrm{T} \times \mathbb{N}^{|V| \times 1}$ the set of all possible states for $\mathcal{P}$.*

The effect of the firing of an actor on the channel state is to add its rates to the respective coordinate of all the channels. For an actor v_i, the i-th column of $\mathbf{\Gamma}$ gives all the rates per channel. Therefore, to extract that column from the matrix for each actor $v_i \in V$, we use a *unitary firing vector* $\mathbf{u} \in \mathbb{B}^{|V| \times 1}$, such that $u_i = 1$, and for all $j \neq i$ we have $u_j = 0$. We denote $U \subset \mathbb{B}^{|V| \times 1}$ the set of these vectors, and for convenience we denote the unitary activation vector of actor v_i by $\mathbf{u}_i$. With the unitary firing vector of any actor v_i, the product $\mathbf{\Gamma u}_i$

gives a vector holding for each channel e_j the rate of v_i on e_j. For any channel state $\mathbf{c}$, the channel state after the atomic firing of v_i is then $\mathbf{c} + \mathbf{\Gamma u^i}$. Also, the firing of a timed actor is tracked by adding its unitary firing vector to the tracking vector. The firing of an actor has no effect on the current tick.

Definition 6 (Fire). *For a polygraph $\mathcal{P}$, the mapping* $\text{fire} : U \times S \longrightarrow S$ *maps a unitary activation vector* $\mathbf{u}_i$ *and a state* $s = \langle \mathbf{c}, \tau, \mathbf{a} \rangle$ *to the state* $s' = \langle \mathbf{c}', \tau', \mathbf{a}' \rangle$ *such that we have* $\mathbf{c}' = \mathbf{c} + \mathbf{\Gamma u}_i$, $\tau' = \tau$, *and if* $v_i \in V_F$ *then* $\mathbf{a}' = \mathbf{a} + \mathbf{u}_i$, *otherwise* $\mathbf{a}' = \mathbf{a}$.

Remark 1. For two consecutive firings of any actors v_i and v_j from a state $s = \langle \mathbf{c}, \tau, \mathbf{a} \rangle$, the resulting state $s'' = \langle \mathbf{c}'', \tau'', \mathbf{a}'' \rangle$ does not depend on the order of the firings, and $\mathbf{c}'' = \mathbf{c} + \mathbf{\Gamma}(\mathbf{u}_i + \mathbf{u}_j)$. This property can be generalized to any finite number of consecutive firings.

The other possible transition between two states occurs when the global clock ticks. When the global clock ticks, the channel state is not changed, the current tick is adjusted, and the tracking vector is reset.

Definition 7 (Tick). *For a polygraph $\mathcal{P}$, the mapping* $\text{tick} : S \longrightarrow S$ *maps a state* $s = \langle \mathbf{c}, \tau, \mathbf{a} \rangle$ *to the state* $s' = \langle \mathbf{c}', \tau', \mathbf{a}' \rangle$ *such that we have* $\mathbf{c}' = \mathbf{c}$, $\tau' = (\tau + 1) \bmod \pi$, *and* $\mathbf{a}' = \mathbf{0}$.

Executions. The state of $\mathcal{P}$ can evolve by successive application of either *fire* or *tick*. An *execution* of $\mathcal{P}$ is a sequence of such applications starting from a state $s_1 \in S$ and leading to states $e = s_1 \cdots s_n \in S^+$. However, with the frequency constraints, there are some conditions for the applications.

Consider the firing $\text{fire}(\mathbf{u}_i, s)$ of a timed actor v_i in a state $s = \langle \mathbf{c}, \tau, \mathbf{a} \rangle$. In this case, v_i may fire only if the current tick τ is one of its firing ticks, *i.e.* $\tau \in \mathrm{T}_i$. Since it must fire exactly once on such a tick, an additional constraint to fire a timed actor v_i is that it has not fired yet, *i.e.* its coordinate in the tracking vector $\mathbf{a}$ is $a_i = 0$. To capture this constraint, we define a *tick firing vector* $\mathbf{t}^\tau \in \mathbb{B}^{|V| \times 1}$ for each tick $\tau \in \mathrm{T}$, in which a coordinate is set to one if the corresponding actor is expected to fire at tick τ. More formally, for any $v_i \in V \setminus V_F$ we have $t_i^\tau = 0$, and for any $v_j \in V_F$ we have $t_j^\tau = 1$ if $\tau \in \mathrm{T}_j$, and $t_j^\tau = 0$ otherwise. The constraint to fire $v_i \in V_F$ in a state with current tick τ and tracking vector $\mathbf{a}$ is then $a_i < t_i^\tau$.

The clock update $\text{tick}(s)$ in a state $s = \langle \mathbf{c}, \tau, \mathbf{a} \rangle$ is also subject to a constraint: the timed actors that were supposed to fire synchronously with the current tick have done so exactly once, *i.e.* $\mathbf{a} = \mathbf{t}^\tau$.

Definition 8 (Synchronous execution). *An execution* $e = s_1 \cdots s_n \in S^+$ *of a polygraph $\mathcal{P}$ is* synchronous *if* $\forall 1 \leqslant k < n$, *we have* $s_k = \langle \mathbf{c}, \tau, \mathbf{a} \rangle$ *such that:*

- *either* $s_{k+1} = \text{fire}(\mathbf{u}_i, s_k)$ *for some* $v_i \in V$, *and in addition, if* $v_i \in V_F$, *then* $a_i < t_i^\tau$,
- *or* $s_{k+1} = \text{tick}(s_k)$, *and in addition,* $\mathbf{a} = \mathbf{t}^\tau$.

Until now, we considered executions of a polygraph where the order of the firings is constrained only by the frequencies. However, for an actor to fire, there must be enough tokens on its input channels, or its rational communication rate must allow firings consuming 0 tokens. In order to fire an actor v_i in a state $s = \langle \mathbf{c}, \tau, \mathbf{a} \rangle$, we require that for each input channel e_j of v_i, since the rate γ_{ji} is negative, the channel state c_j must be large enough to avoid reaching a negative state, *i.e.* $c_j + \gamma_{ji} \geqslant 0$, or equivalently $c_j \geqslant |\gamma_{ji}|$. This constraint requires an ordering of the actor firings such that a producer is fired a sufficient number of times for a consumer to be able to fire in turn.

Definition 9 (Non-blocking execution). *An execution $e = s_1 \cdots s_n \in S^+$ of a polygraph $\mathcal{P}$ is* non-blocking *if $\forall 1 \leqslant k < n$, we have $s_k = \langle \mathbf{c}, \tau, \mathbf{a} \rangle$ such that:*

- *either $s_{k+1} = \text{fire}(\mathbf{u}_i, s_k)$ for some $v_i \in V$, and in addition, $\forall e_j \in \text{in}(v_i)$, $c_j \geqslant |\gamma_{ji}|$,*
- *or $s_{k+1} = \text{tick}(s_k)$.*

Consistency property. If verified, the *consistency* property of $\mathcal{P}$ guarantees that it is possible to build a synchronous execution $e = s_1 \cdots s_n \in S^+$ such that $s_1 = \langle \mathbf{m}, 0, \mathbf{0} \rangle$ and $s_1 = s_n$. Such an execution is called a *consistent* execution of $\mathcal{P}$, and can obviously be repeated an indefinite number of times to build a consistent execution of arbitrary length. [14, Theorem 1] states that a necessary and sufficient condition for a given SDF graph to be consistent is that there is a non-trivial solution $\mathbf{x}$ to $\mathbf{\Gamma x} = \mathbf{0}$.

To extend this result to polygraphs, as explained in the previous section, we need to take into account the frequencies of the timed actors. In other words, we need to make sure that it is possible to have a synchronous execution with x_i firings per actor v_i. The additional constraint due to the frequencies is that the number of firings x_i of all the timed actors v_i corresponds to a number $r \in \mathbb{N}$ of repetitions of the global clock period.

To state the conditions for a polygraph to be consistent, we thus want to separate the number of firings of the timed actors from the others. We define the vector $\mathbf{t} = \sum_{\forall \tau \in \mathrm{T}} \mathbf{t}^\tau$ giving for each timed actor v_i the number t_i of expected firings per period of the global clock. We then define the set $Y \subset \mathbb{N}^{|V| \times 1}$ of vectors $\mathbf{y}$ such that we have a number of firings $y_i \neq 0$ only for $v_i \in V \setminus V_F$.

Theorem 1. *A polygraph $\mathcal{P}$ has a consistent execution if and only if there exists a non-trivial solution $\mathbf{x} \in \mathbb{N}^{|V| \times 1}$ to $\mathbf{\Gamma x} = \mathbf{0}$ such that $\mathbf{x} = \mathbf{y} + r\mathbf{t}$ for some $\mathbf{y} \in Y$ and $r \in \mathbb{N}$. Any such solution is called a* repetition vector *of $\mathcal{P}$. Moreover, there exists a* minimal repetition vector $\mathbf{x}$ *such that for any other repetition vector $\mathbf{x}'$ we have $\mathbf{x}' = k\mathbf{x}$ for some $k \in \mathbb{N}$.*

Sketch of proof. First, we prove that the condition is sufficient, and suppose that there exists such a solution $\mathbf{x}$. Then we can decompose:

$$\mathbf{x} = \mathbf{y} + \underbrace{\underbrace{(\mathbf{t^0} + \ldots + \mathbf{t^{\pi-1}})}_{=\mathbf{t}} + \ldots + \underbrace{(\mathbf{t^0} + \ldots + \mathbf{t^{\pi-1}})}_{=\mathbf{t}}}_{=r\mathbf{t}}$$

The required consistent execution can be obtained by constructing sub-executions corresponding to this decomposition, relying on Definition 8 and Remark 1.

Claim (1). There exists a synchronous execution $e_1 \in S^+$ with starting state $s = \langle \mathbf{m}, 0, \mathbf{0} \rangle$ and ending state $s' = \langle \mathbf{m} + \mathbf{\Gamma y}, 0, \mathbf{0} \rangle$.

The execution e_1 is constructed by applying y_i firings of each actor $v_i \in V \setminus V_F$ (in any order). Since the fired actors are not timed actors, any such sequence is synchronous. The resulting channel state is $\mathbf{m} + \mathbf{\Gamma y}$ as per Remark 1.

Claim (2). For any starting state $s = \langle \mathbf{c}, \tau, \mathbf{0} \rangle$, there exists a synchronous execution $e_2 \in S^+$ starting from s with ending state $s' = \langle \mathbf{c} + \mathbf{\Gamma t}^\tau, (\tau + 1) \bmod \pi, \mathbf{0} \rangle$.

The execution e_2 for τ is constructed by firing exactly once each timed actor supposed to do so at tick τ, and then applying the tick mapping.

Claim (3). For any starting state $s = \langle \mathbf{c}, 0, \mathbf{0} \rangle$, there exists a synchronous execution $e_3 \in S^+$ starting from s with ending state $s' = \langle \mathbf{c} + \mathbf{\Gamma t}, 0, \mathbf{0} \rangle$.

The execution e_3 is obtained by successively executing e_2 for $\tau = 0, \ldots, \pi - 1$.

Claim (4). There exists a synchronous execution $e_4 \in S^+$ with starting state $s = \langle \mathbf{m}, 0, \mathbf{0} \rangle$ and ending state $s' = \langle \mathbf{m} + \mathbf{\Gamma}(\mathbf{y} + r\mathbf{t}), 0, \mathbf{0} \rangle$.

The sequence e_4 is constructed by executing e_1, followed by e_3 repeated r times. Hence, given that $\mathbf{\Gamma x} = \mathbf{0}$ and $\mathbf{x} = \mathbf{y} + r\mathbf{t}$, it can be easily checked that the ending state of e_4 is the same as its starting state, and e_4 is consistent. The fact that the condition is also necessary follows from the definitions. Since the current tick must return to 0 after a consistent execution, such an execution must perform a number r of periods of the global clock for some $r \in \mathbb{N}$, in other words it must contain $r\pi$ applications of the tick mapping and rt_i firings of each timed actor v_i. The existence of a minimal solution immediately follows from the fact that in this case $\operatorname{rank}(\mathbf{\Gamma}) = |V| - 1$ according to [14, Corollary of Lemma 2].

Due to lack of space, a detailed proof is left to the reader. □

Liveness property. If verified, the *liveness* property of $\mathcal{P}$ guarantees that it is possible to build a consistent execution $e = s_1 \cdots s_n \in S^+$ such that e is also a non-blocking execution. Such an execution e is called a *live execution*.

In a way similar to [14, Theorem 3], we define the notion of a scheduler building only synchronous and non-blocking executions. Our goal is to show that $\mathcal{P}$ has a live execution if and only if any such scheduler can build a consistent execution.

From now on, we consider that $\mathcal{P}$ is consistent with minimal repetition vector $\mathbf{x}$. We define the mapping $count : V \times S^+ \longrightarrow \mathbb{N}$ that given an actor v_i and an execution $e = s_1 \cdots s_n \in S^+$ returns the number of firings of v_i in e, *i.e.* the number of k such that $1 \leqslant k < n$ and $s_{k+1} = \text{fire}(\mathbf{u}_i, s_k)$. Notice that since a live execution e of $\mathcal{P}$ is also consistent, by definition we have $\forall v_i \in V, \text{count}(v_i, e) = x_i$. Also, we say that an actor $v_i \in V$ is *runnable* after an execution $e \in S^+$ with ending state s if $\text{count}(v_i, e) < x_i$ and the one-step execution $ss' \in S^+$ with $s' = \text{fire}(\mathbf{u}_i, s)$ is synchronous and non-blocking.

Definition 10 (Scheduler). *A* scheduler *of $\mathcal{P}$ is a mapping $\sigma : S^+ \longrightarrow S^+$ that maps an execution $e = s_1 \cdots s_n \in S^+$ to an execution $e' \in S^+$ such that if we denote $s_n = \langle \mathbf{c}, \tau, \mathbf{a} \rangle$ we have:*

- *either $e' = s_1 \cdots s_n s' \in S^+$ with $s' = \text{fire}(\mathbf{u}_i, s_n)$ for some actor v_i runnable after e;*
- *or $e' = s_1 \cdots s_n s' \in S^+$ with $s' = \text{tick}(s_n)$ and $\mathbf{a} = \mathbf{t}^\tau$;*
- *or $e' = e$ if there is no runnable actor after e and $\mathbf{a} \neq \mathbf{t}^\tau$.*

An execution defined by a scheduler σ is the fixed point constructed by recursive application[1] of σ starting from an initial execution $e = (\langle \mathbf{m}, 0, \mathbf{0} \rangle)$.

Theorem 2. *Let $\mathcal{P}$ be a consistent polygraph with minimal repetition vector $\mathbf{x}$, σ a scheduler of $\mathcal{P}$, and e the execution defined by σ. Then $\mathcal{P}$ has a live execution if and only if $\forall v_i \in V, \text{count}(v_i, e) = x_i$.*

Sketch of proof. The condition is obviously sufficient. The proof that it is also necessary can be easily made by induction. If e is a live execution and e' is a synchronous and non-blocking execution constructed by σ so far, with $|e'| < |e|$, we can show that e' can be extended by one more step (*e.g.* by taking the first step present in e but not in e', since its preconditions are necessarily satisfied). □

4 Tool Support for Liveness Checking

DIVERSITY is a customizable model analysis tool based on symbolic execution, available in the *Eclipse Formal Modeling Project* [17]. DIVERSITY provides a pivot language called *xLIA* (eXecutable Language for Interaction and Architecture) introducing a set of communication and execution primitives allowing one to encode a wide class of dynamic model semantics [2,9], Communicating STS [1], and abstractions of hybrid systems [15]. In this work, we use it to analyze Polygraph models, to check their liveness in a similar way to that defined by a scheduler as per Definition 10.

The root entity in an xLIA model is a so-called *system*. A system is an executable entity that can be atomic (state-machine) or compositional or hierarchical. A Polygraph model translated to xLIA is a system where the actors are state-machines with input/output ports associated with the ends of the channels. They communicate asynchronously over FIFO queues, bounded or not, using xLIA connectors. Variables are used to store received tokens on input instructions in transitions, with guards conditioning their firing, and output statements to model their token productions.

Figure 3 represents such a state machine for any actor of the polygraph in Fig. 1. Each transition is labeled with xLIA macros representing the actions performed. The *init* macro moves the initial marking from the input queues to the

[1] Hence, a scheduler can be also defined as a *partial* mapping on $\sigma^*(\langle \mathbf{m}, 0, \mathbf{0} \rangle)$.

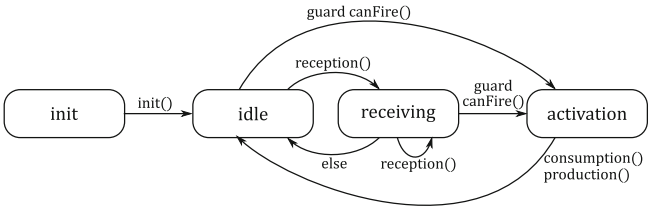


Fig. 3. xLIA state machine pattern for an actor of a polygraph

counter of available input tokens, *canFire()* tests if enough tokens are present for a non-blocking firing, *consumption* decrements the counter of available input tokens, *production* sends the production rate on the successor's queue, and *reception* reads that rate and adds it to the number of available tokens. Regarding state machine semantics, all the states are pseudo-states, except *idle* which is stable. This means that any fired transition must be completed until returning to the idle state. The *else* transition will be evaluated if there is no possible *reception*.

The xLIA language allows a fine-grained definition of an execution model for the actors of a polygraph. Some instructions associate a sequence of actors to fire with each tick of a clock. When attempting to fire a timed actor, only one firing is triggered if possible, and when attempting the same for other actors, as many firings as possible are triggered. Hence, the timed actors are only fired at the expected tick, and cause a deadlock result if it's not possible. For the other actors, a counter limits their number of firings to their coordinate in the minimal repetition vector, as required by Theorem 2. With this setup, for a polygraph $\mathcal{P}$ with minimal repetition vector $\mathbf{x} = \mathbf{y} + r\mathbf{t}$, the length of a live execution path is $r\pi$, plus one for the initialization step handling the initial marking. Any path with less steps leads to a deadlock.

We tested this technique using DIVERSITY on an *Intel core i7*. For the polygraph of Fig. 1 with initial marking (ii), the tool finds that the liveness property is verified. We also tested the initial marking (i), and the tool correctly identified a deadlock in less than 200 ms. This example is extracted from a more complex polygraph modeling an Advanced Driver-Assistance System (ADAS), that we also used to evaluate the liveness checking tool. The considered polygraph has 18 actors (5 of which are timed actors), 32 channels (6 of which have an initial marking), where 10 actors have rational communication rates. For a correctly marked model, we find a live execution sequence in 4s.

5 Discussion and Related Work

In [16], an extension to SDF is proposed to add a single throughput constraint on a channel of a consistent graph. From this constraint, a firing frequency is derived for the actors by transitivity. This approach, while preserving the consistency property by construction, does not allow the expression of a frequency constraint

per actor, based on a real-life constraint on the modeled component, nor the explicit synchronization of the firings on a reference time scale.

The programming model PTIDES [18] combines a real-time semantic for sensors and actuators, and a discrete event semantic for other components like computation kernels. These other components have an awareness of the real time through a logical time abstraction. The resulting execution semantic has similarities with Polygraph, since some components are constrained by real-time and others only react to their stimuli. The semantic of PTIDES is much more flexible than Polygraph, since it does not require fixed production or consumption rates. On the other hand, and as opposed to Polygraph, there is no way to derive a consistent and live periodic schedule in PTIDES, which makes static performance prediction more difficult. Nevertheless, since the semantics are similar, we believe that the notion of logical time as defined in PTIDES is applicable to practical distributed implementations of polygraphs.

Synchronous programming languages [7,8] can be used to express a data flow between synchronous periodic nodes, in order to generate correct-by-construction programs. In these approaches, all the nodes are synchronous, while in Polygraph, some actors fire asynchronously when enabled. Also, the goal of our approach is to be able to reason formally on the modeled systems, and automate as many tasks as possible in its design, implementation and validation. Such a task could be the association of the asynchronous firings to ticks of the global clock, and the generation of a synchronous program for automatic code generation.

Recently published research [6] follows a similar approach to ours. By mixing elements from two existing formalisms, one allowing the specification of time-triggered tasks and the other the specification of data flow actors, the expressiveness of the resulting modeling framework is comparable to that of Polygraph. The main difference is that Polygraph is a single formalism with decidable properties and algorithms to check them in practice. In [6], the impact of the combination of constraints from two different formalisms on their respective properties is not discussed, as the proposed approach is more focused on the performance evaluation. The experimental results the authors obtained are in favor of the modeling approach we have in common.

6 Conclusion

We have introduced Polygraph, a data flow formalism extending SDF with synchronous firing semantics for the actors. We have shown that with this extension, the existing conditions to decide of a given SDF graph's consistency and liveness were no longer sufficient. We have extended the corresponding theorems and shown that the expressiveness extensions we proposed do not impact the decidability of these properties. Finally, as a first step towards tool assisted modeling of polygraphs, we have introduced a framework relying on DIVERSITY to verify their liveness.

Our next step is to further extend Polygraph to add flexibility in the execution semantic, with the same objective to preserve the capability to perform

accurate static analysis of a system's performance. Still, with this first extension, there are already interesting research perspectives regarding the applicability of existing static performance analysis techniques, and their potential extensions to take into account the specifics of a polygraph's scheduling.

Acknowledgement. Part of this work has been realized in the FACE project, involving CEA List and Renault. The Polygraph formalism has been used as a theoretical foundation for the software methodology in the project.

References

1. Arnaud, M., Bannour, B., Lapitre, A.: An illustrative use case of the DIVERSITY platform based on UML interaction scenarios. Electr. Notes Theor. Comput. Sci. **320**, 21–34 (2016)
2. Bannour, B., Escobedo, J.P., Gaston, C., Le Gall, P.: Off-line test case generation for timed symbolic model-based conformance testing. In: Nielsen, B., Weise, C. (eds.) ICTSS 2012. LNCS, vol. 7641, pp. 119–135. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34691-0_10
3. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.A.: Cyclo-static data flow. In: Proceedings of the 1995 International Conference on Acoustics, Speech, and Signal Processing, vol. 5, pp. 3255–3258 (1995)
4. Bodin, B., Munier-Kordon, A., de Dinechin, B.D.: K-periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph. In: Proceedings of the 2012 International Conference on Embedded Computer Systems (SAMOS), pp. 152–159 (2012)
5. Bouakaz, A., Fradet, P., Girault, A.: Symbolic buffer sizing for throughput-optimal scheduling of dataflow graphs. In: Proceedings of the 22nd IEEE Real-Time Embedded Technology and Applications Symposium (RTAS 2016) (2016)
6. Breaban, G., Stuijk, S., Goossens, K.: Efficient synchronization methods for LET-based applications on a multi-processor system on chip. In: Design, Automation Test in Europe Conference Exhibition (DATE) 2017, pp. 1721–1726 (2017)
7. Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M.: N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. SIGPLAN Not. **41**(1), 180–193 (2006)
8. Forget, J., Boniol, F., Lesens, D., Pagetti, C.: A multi-periodic synchronous dataflow language. In: 2008 11th IEEE High Assurance Systems Engineering Symposium, pp. 251–260 (2008)
9. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 1–18. Springer, Heidelberg (2006). https://doi.org/10.1007/11754008_1
10. Geilen, M., Basten, T., Stuijk, S.: Minimising buffer requirements of synchronous dataflow graphs with model checking. In: Proceedings of the 42nd Design Automation Conference, pp. 819–824. IEEE (2005)
11. Ghamarian, A.H., et al.: Throughput analysis of synchronous data flow graphs. In: Proceedings of the Sixth International Conference on Application of Concurrency to System Design (ACSD 2006), pp. 25–36 (2006)

12. Ghamarian, A.H., Stuijk, S., Basten, T., Geilen, M.C.W., Theelen, B.D.: Latency minimization for synchronous data flow graphs. In: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007), pp. 189–196 (2007)
13. Kahn, G., MacQueen, D., Laboria, I.: Coroutines and Networks of Parallel Processes. IRIA Research Report, IRIA laboria (1976)
14. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **C–36**(1), 24–35 (1987)
15. Medimegh, S., Pierron, J.Y., Gallois, J., Boulanger, F.: A new approach of qualitative simulation for the validation of hybrid systems. In: Proceedings of the workshop on Model Driven Engineering Languages and Systems (MODELS). ACM (2016)
16. Selva, M.: Performance monitoring of throughput constrained dataflow programs executed on shared-memory multi-core architectures. Theses, INSA de Lyon (2015)
17. The List Institute: CEA Tech: The DIVERSITY Tool. http://projects.eclipse.org/proposals/eclipse-formal-modeling-project/
18. Zhao, Y., Liu, J., Lee, E.A.: A programming model for time-synchronized distributed real-time systems. In: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS 2007), pp. 259–268. IEEE (2007)

Software Testing

CoVeriTest: Cooperative Verifier-Based Testing

Dirk Beyer and Marie-Christine Jakobs

LMU Munich, Munich, Germany

Abstract. Testing is a widely used method to assess software quality. Coverage criteria and coverage measurements are used to ensure that the constructed test suites adequately test the given software. Since manually developing such test suites is too expensive in practice, various automatic test-generation approaches were proposed. Since all approaches come with different strengths, combinations are necessary in order to achieve stronger tools. We study cooperative combinations of verification approaches for test generation, with high-level information exchange.
We present CoVeriTest, a hybrid approach for test-case generation, which iteratively applies different conditional model checkers. Thereby, it allows to adjust the level of cooperation and to assign individual time budgets per verifier. In our experiments, we combine explicit-state model checking and predicate abstraction (from CPAchecker) to systematically study different CoVeriTest configurations. Moreover, CoVeriTest achieves higher coverage than state-of-the-art test-generation tools for some programs.

Keywords: Test-case generation · Software testing · Test coverage · Conditional model checking · Cooperative verification · Model checking

1 Introduction

Testing is a commonly used technique to measure the quality of software. Since manually creating such test suites is laborious, automatic techniques are used: e.g., model-based techniques for black-box testing and techniques based on control-flow coverage for white-box testing. Many automatic techniques have been proposed, ranging from random testing [36,57] and fuzzing [26,52,53], over search-based testing [55] to symbolic execution [23,24,58] and reachability analyses [5,12,45,46]. The latter are well-suited to find bugs and derive test suites that achieve high coverage, and several verification tools support test generation (e.g., Blast [5], PathFinder [61], CPAchecker [12]). The reachability checks for all test goals seem too expensive, but in practice, those approaches can be made pretty efficient.

Encouraged by tremendous advances in software verification [3] and a recent case study that compared model checkers with test tools w.r.t. bug finding [17], we study a new kind of combination of reachability analyses for test generation. Combinations are necessary because different analysis techniques have different strength and weaknesses. For example, consider function `foo` in Listing 1. Explicit state model checking [18,33] tracks the values of variables i and s and easily

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 389–408, 2019.
https://doi.org/10.1007/978-3-030-16722-6_23

detects the reachability of the statements in the outermost `if` branch (lines 3–6), while it has difficulties with the complex condition in the else-branch (line 8). In contrast, predicate abstraction [33,39] can easily derive test values for the complex condition in line 8, but to handle the `if` branch (lines 3–6) it must spent effort on the detection of the predicates $s = 0$, $s = 1$, and $i = 0$. Independently of each other, test approaches [1,34,47,54] and verification approaches [9,10,29,37] employ combinations to tackle such problems. However, there are no approaches yet that combine different reachability analyses for test generation.

```
0 void foo(int i, int n) {
1   int s=0;
2   if(i==0)
3     while (i==0)    {
4       if(s==0) init();
5       if(s==1) i = exec();
6       s=(s+1)%2;
7     }
8   else if(2*i<n && i>0) exec();
9 }
```

Fig. 1. Example program foo

Inspired by abstraction-driven concolic testing [32], which interleaves concolic execution and predicate abstraction, we propose CoVeriTest, which stands for cooperative verifier-based testing. CoVeriTest iteratively executes a given sequence of reachability analyses. In each iteration, the analyses are run in sequence and each analysis is limited by its individual, but configurable time limit. Furthermore, CoVeriTest allows the analysis to share various types of analysis information, e.g., which paths are infeasible, have already been explored, or which abstraction level to use. To get access to a large set of reachability analyses, we implemented CoVeriTest in the configurable software-analysis framework CPAchecker [15]. We used our implementation to evaluate different CoVeriTest configurations on a large set of well-established benchmark programs and to compare CoVeriTest with existing state-of-the-art test-generation techniques. Our experiments confirm that reachability analyses are valuable for test generation.

Contributions. In summary, we make the following contributions:

- We introduce CoVeriTest, a flexible approach for high-level interleaving of reachability analyses with information exchange for test generation.
- We perform an extensive evaluation of CoVeriTest studying 54 different configurations and two state-of-the-art test-generation tools[1].
- CoVeriTest and all our experimental data are publically available[2] [13].

2 Testing with Verifiers

The basic idea behind testing with verifiers is to derive test cases from counter-examples [5,61]. Thus, meeting a test goal during verification has to trigger a specification violation. First, we remind the reader of some basic notations.

[1] We choose the best two tools VeriFuzz and Klee from the international competition on software testing (Test-Comp 2019) [4]. https://test-comp.sosy-lab.org/2019/

[2] https://www.sosy-lab.org/research/coop-testgen/

Programs. Following literature [9], we represent programs by control-flow automata (CFAs). A CFA $P = (L, \ell_0, G)$ consists of a set L of program locations (the program-counter values), an initial program location $\ell_0 \in L$, and a set of control-flow edges $G \subseteq L \times Ops \times L$. The set Ops describes all possible operations, e.g., assume statements (resulting from conditions in `if` or `while` statements) and assignments. For the program semantics, we rely on an operational semantics, which we do not further specify.

Abstract Reachability Graph (ARG). ARGs record the work done by reachability analyses. An ARG is constructed for a program $P = (L, \ell_0, G)$ and stores (a) the abstract state space that has been explored so far, (b) which abstract states must still be explored, and (c) what abstraction level (tracked variables, considered predicates, etc.) is used. Technically, an ARG is a five-tuple $(N, succ, root, F, \pi)$ that consists of a set N of abstract states, a special node $root \in N$ that represents the initial states of program P, a relation $succ \subseteq N \times G \times N$ that records already explored successor relations, a set $F \subseteq N$ of frontier nodes, which remembers all nodes that have not been fully explored, and a precision π describing the abstraction level. Every ARG must ensure that a node n is either contained in F or completely explored, i.e., all abstract successors have been explored. We use ARGs for information exchange between reachability analyses.

Test Goals. In this paper, we are interested in structural coverage, e.g., branch coverage. Transferred to our notion of programs, this means that our test goals are a subset of the program's control-flow edges. For using a verifier to generate tests, we have to encode the test goals as a specification violation. Figure 2 shows a possible encoding, which uses a protocol automaton. Whenever a test goal is executed, the automaton transits from the initial, safe state q_0 to the accepting state q_e, which marks a property violation. Note that reachability analyses, which we consider for test generation, can easily monitor such specifications during exploration.

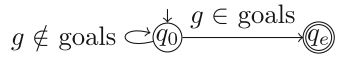


Fig. 2. Encoding test goals as specification violation

Now, we have everything at hand to describe how reachability analyses generate tests. Algorithm 1 shows the test-generation process. The algorithm gets as input a program, a set of test goals, and a time limit for test generation. For cooperative test generation, we need to guide state-space explorations. To this end, we also provide an initial ARG and a condition. A condition is a concept known from conditional model checking [10] and describes which parts of the state space have already been explored by other verifiers. A verifier, e.g., a reachability analysis, can use a condition to ignore the already explored parts of the state space. Verifiers that do not understand conditions can safely ignore them.

At the beginning, Alg. 1 sets up the data structures for the test suite and the set of covered goals. To set up the specification, it follows the idea of Fig. 2. As long as not all test goals are covered, there exist abstract states that must be explored, and the time limit has not elapsed, the algorithm tries to generate new tests. Therefore, it resumes the exploration of the current ARG [5] taking into

Algorithm 1. Generating tests with a (conditional) reachability analysis

Input: `prog` $= (L, \ell_0, G)$, `goals` $\subseteq G$, `limit` $\in \mathbb{N}$, `arg` =(N,succ, root, F, π), condition ψ
Output: generated `test_suite`, `covered` goals, updated `arg`

```
1: test_suite=∅;  covered=∅;
2: φ=generate_specification(goals);

3: while (goals ≠ ∅ and arg.F ≠ ∅ and elapsed_time<limit) do
4:     arg = explore(prog, φ, arg, ψ, limit − elapsed_time);

5:     if (arg.F ≠ ∅ and elapsed_time<limit) then
6:         τ = extract_counterexample_trace(arg);
7:         test_suite = test_suite ∪ generate_test_from_trace(τ);

8:         goals = goals\{last_edge(τ)};  covered = covered ∪ {last_edge(τ)}

9:     φ=generate_specification(goals);
10: return (test_suite, covered, arg);
```

account program `prog`, specification φ, and (if understood) the condition ψ. If the exploration stops, then it returns an updated ARG. Exploration stops due to one of three reasons: (1) the state space is explored completely ($F = \emptyset$), (2) the time limit is reached, or (3) a counterexample has been found.[3] In the latter case, a new test is generated. First, a counterexample trace is extracted from the ARG. The trace describes a path through the ARG that starts at the root and its last edge is a test goal (the reason for the specification violation). Next, a test is constructed from the path and added to the test suite. Basically, the path is converted into a formula and a satisfying assignment[4] is used as the test case. For the details, we refer the reader to the work that defined the method [5]. Additionally, the covered goal (last edge on the counterexample path) is removed from the set of open test goals and added to the set of covered goals. Finally, the specification is updated to no longer consider the covered goal. When the algorithm finishes, it returns the generated test suite, the set of covered goals and the last ARG considered. The ARG is returned to enable cooperation.

3 CoVeriTest

The previous section described how to use a single reachability analysis to produce tests for covering a set of test goals. Due to different strengths and weaknesses, some test goals are harder to cover for one analysis than for another. To

[3] We assume that an exploration is only complete if no counterexample exists.
[4] We assume that only feasible counterexamples are contained and infeasible counterexamples were eliminated by the reachability analysis during exploration.

Algorithm 2. CoVeriTest: alternating reachability analyses to generate tests

Input: `prog` $= (L, \ell_0, G)$, `goals` $\subseteq G$, `total_limit` $\in \mathbb{N}$, `configs` $\in (\text{analysis} \times \mathbb{N})^+$
Output: `test_suite`

```
 1: test_suite=∅;  args=⟨⟩;  current=0;
 2: while (goals ≠ ∅ and elapsed_time<total_limit) do
 3:     analysis = configs[current].first;  limit = configs[current].second;

 4:     (arg,ψ) = cooperateAndInit(prog, args, configs.length);
 5:     (tests, covered, arg) = analysis(prog, goals, limit, arg, ψ);

 6:     test_suite=test_suite ∪ tests; goals=goals\covered; args=args ∘⟨arg⟩;
 7:     if (arg.F=∅) then
 8:         return test_suite;
 9:     current = (current+1) % configs.length;
10: return test_suite;
```

maximize the number of covered goals, different analyses should be combined. In CoVeriTest, we rotate analyses for test generation. Thus, we avoid that analyses try to cover the same goal in parallel and we do not need to know in advance which analysis can cover which goals. Moreover, analyses that get stuck trying to cover goals that other analyses handle later, get a chance to recover. Additionally, CoVeriTest supports cooperation among analyses. More concrete: analyses may extract and use information from ARGs constructed by previous analysis runs.

Algorithm 2 describes the CoVeriTest workflow. It gets four inputs. Program, test goals, and time limit are already known from Alg. 1 (test generation with a single analysis). Additionally, CoVeriTest gets a sequence of configurations, namely pairs of reachability analysis and time limit. The time limit accompanied with the analysis restricts the runtime of the respective analysis per call (see line 5). In contrast to Alg. 1, CoVeriTest does not get an ARG or condition. To enable cooperation between analyses, CoVeriTest constructs these two elements individually for each analysis run. During construction, it may extract and use information from results of previous analysis runs.

After initializing the test suite and the data structure to store analysis results (`args`), CoVeriTest repeatedly iterates over the configurations. It starts with the first pair in the sequence and finishes iterating when its time limit exceeded or all goals are covered. In each iteration, CoVeriTest first extracts the analysis to execute and its accompanied time limit (line 3). Then, it constructs the remaining inputs of the analysis: ARG and condition. Details regarding the construction are explained later in Alg. 3. Next, CoVeriTest executes the current analysis with the given program, the remaining test goals, the accompanied time limit, and the constructed ARG and condition. When the analysis has finished, CoVeriTest adds the returned tests to its test suite, removes all test goals covered by the analysis run from the set of goals, and stores the analysis result for cooperation (concatenates `arg` to the sequence of ARGs). If the analysis finished its exploration (`arg`.F$=\emptyset$), any remaining test goal should be unreachable and

Algorithm 3. cooperateAndInit: set up start point for analysis exploration, possibly transferring knowledge from previous analysis runs

Input: `prog` $= (L, \ell_0, G)$, `args` $\in (arg)^+$, `numAnalyses` $\in \mathbb{N}$
Output: ARG for program `prog`, condition describing explored state space

```
1: ψ=false;   π = ∅;   root = (ℓ0, ⊤);
2: if (length(args)≥numAnalyses) then
3:     if (reuse-arg) then
4:         return (last_arg_of_analysis(numAnalyses, args), ψ);
5:     if (reuse-precision) then
6:         π = last_arg_of_analysis(numAnalyses, args).π;
7: if (use-condition ∧ length(args)>0) then
8:     ψ = extract_condition(args[length(args)-1]);
9: return (({root}, ∅, root, {root}, π), ψ);
```

CoVeriTest returns its test suite. Otherwise, CoVeriTest determines how to continue in the next iteration (i.e., which configuration to consider). At the end of all iterations, CoVeriTest returns its generated test suite.

Next, we explain how to construct the ARG and the condition input for an analysis. The ARG describes the level of abstraction and where to continue exploration while the condition describes which parts of the state space have already been explored. Both guide the exploration of an analysis, which makes them well-suited for cooperation. While there are plenty of possibilities for cooperation, we currently only support three basic options: continue exploration of the previous ARG of the analysis (`reuse-arg`), reuse the analysis' abstraction level (`reuse-precision`), and restrict the exploration to the state space left out by the previous analysis (`use-condition`). The first two options only ensure that an analysis does not loose too much information due to switching. The last option, which is inspired by abstraction-driven concolic execution [32], indeed realizes cooperation between different analyses. Note that the last two options can also be combined.[5] If all options are turned off, no information will be exchanged.

Algorithm 3 shows the cooperative initialization of ARG and condition discussed above. It gets three inputs: the program, a sequence of args needed to realize cooperation, and the number of analyses used. At the beginning, it initializes the ARG components and the condition assuming no cooperation should be done. The condition states that nothing has been explored, the abstraction level becomes the coarsest available, and the ARG root considers the start of all program executions (initial program location and arbitrary variable values). If no cooperation is configured or the ARG required for cooperation is not available (e.g., in the first round), the returned ARG and condition tell the analysis to explore the complete state space from scratch. In all other cases, the analysis will be guided by information obtained in previous iterations. Option `reuse-arg`

[5] In contrast, the options `reuse-arg` and `use-conditions` cannot be combined because they are incompatible. The existing ARG does not fit to the constructed condition. Since `reuse-arg` subsumes `reuse-precision`, a combination makes no sense.

looks up the last ARG of the analysis stored in `args`. `Reuse-precision` considers the same ARG as `reuse-arg`, but only provides the ARG's precision π. For `use-condition`, a condition is constructed from the last ARG in args. For the details of the condition construction, we refer to conditional model checking [10].

Next, we study the effectiveness of different CoVeriTest configurations and compare CoVeriTest with existing test-generation tools.

4 Evaluation

We systematically evaluate CoVeriTest along the following claims:

Claim 1. For analyses that discard their own results from previous iterations (i.e., `reuse-arg` and `reuse-precision` turned off), CoVeriTest achieves higher coverage if switches between analyses happen rarely. *Evaluation Plan:* We look at CoVeriTest configurations in which analyses discard their own, previous results and compare the number of covered test goals reported by configurations that only differ in the analyses' time limits.

Claim 2. For analyses that reuse knowledge from their own, previous execution (i.e., `reuse-arg` or `reuse-precision` turned on), CoVeriTest achieves higher coverage if favoring more powerful analyses. *Evaluation Plan:* We look at CoVeriTest configurations in which analyses reuse their own, previous knowledge and compare the number of covered test goals reported by configurations that only differ in the analyses' time limits.

Claim 3. CoVeriTest performs better if analyses reuse knowledge from their own, previous execution (i.e., `reuse-arg` or `reuse-precision` turned on). *Evaluation Plan:* From all sets of CoVeriTest configurations that only differ in the analyses' time limits, we select the best and compare these.

Claim 4. Interleaving multiple analyses with CoVeriTest often achieves better results than using only one of the analyses for test generation. *Evaluation Plan:* We compare the number of covered goals reported by the best CoVeriTest configuration with those numbers achieved when running only one analysis of the CoVeriTest configuration for the total time limit.

Claim 5. Interleaving verifiers for test generation is often better than running them in parallel. *Evaluation Plan:* We compare the number of covered goals reported by the best CoVeriTest configuration with the number achieved when running all analyses of the CoVeriTest configuration in parallel.

Claim 6. CoVeriTest complements existing test-generation tools. *Evaluation Plan:* We use the same infrastructure and resources as used by the International Competition on Software Testing (Test-Comp'19)[6] and let the best CoVeriTest configuration construct test suites. These test suites are executed by the Test-Comp'19 validator to measure the achieved branch coverage. Then, we compare the coverage achieved by CoVeriTest with the coverage of the best two test-generation tools from Test-Comp'19.

[6] https://test-comp.sosy-lab.org/2019/

4.1 Setup

CoVeriTest Configurations. We implemented CoVeriTest in the software analysis framework CPAchecker [15]. Basically, we implemented Algs. 1, 2 and integrated Alg. 3 into Alg. 2. For condition construction, we reuse the code from conditional model checking [10]. For our experiments, we combine value [18] and predicate analysis [16]. Both have been used in cooperative verification [10,11,21].

Value analysis. CPAchecker's value analysis [18] tracks the values of variables stored in its current precision explicitly while assuming that the remaining variables may have any possible value. It iteratively increases its precision, i.e., the variables to track, combining counterexample-guided abstraction [28] with path-prefix slicing [22], and refinement selection [21]. Value analysis is efficient if few variable values need to be tracked, but it may get stuck in loops or suffers from a large state space in case variables are assigned many different values.

Predicate analysis. CPAchecker's predicate analysis uses predicate abstraction with adjustable-block encoding (ABE) [16]. ABE is configured to abstract at loop heads and uses the strongest postcondition at all remaining locations. To compute the set of predicates—its precision—, it uses counterexample-guided abstraction refinement [28] combined with lazy refinement [43] and interpolation [41]. While the predicate analysis is powerful and often summarizes loops easily, successor computation may require expensive SMT solver calls.

For both analyses, a CoVeriTest configuration specifies how Alg. 3 reuses the ARGs returned by previous analysis runs to set up the initial ARG and condition. In our experiments, we consider the following types of reuses.

plain Ignores all ARGs returned by previous analysis runs, i.e., `reuse-arg`, `reuse-prec`, and `use-condition` are turned off.

cond$_{\text{v}}$ The value analysis does not obtain information from previous ARGs and the predicate analysis is only steered by the condition extracted from the ARG returned by the previous value analysis.

cond$_{\text{p}}$ The value analysis is steered by the condition extracted from the ARG returned by the previous run of the predicate analysis and the predicate analysis ignores all previous ARGs.

cond$_{\text{v,p}}$ Value and predicate analysis are steered by the condition extracted from the last ARG returned, i.e., only `use-condition` turned on.

reuse-prec In each round, each analysis resumes its precision from the previous round, but restarts exploration, i.e., only `reuse-prec` is turned on.

reuse-arg In each round, each analysis continues to explore the ARG it returned in the previous round, i.e., only `reuse-arg` is turned on.

cond$_{\text{v}}$**+r** Similar to cond$_{\text{v}}$, but additionally the value analysis continues to explore the ARG it returned in the previous round and the predicate analysis restarts exploration with its precision from the previous round.

cond$_{\text{p}}$**+r** Similar to cond$_{\text{p}}$, but additionally the value analysis restarts exploration with its precision from the previous round and the predicate analysis continues to explore the ARG it returned in the previous round.

cond$_{\text{v,p}}$**+r** Like cond$_{\text{v,p}}$, but additionally the value and predicate analysis reuse their previous precision, i.e., `reuse-prec` and `use-condition` are turned on.

Finally, we need to fix the time limit for each analysis. We want to find out whether switches between analyses are important to the CoVeriTest approach. Therefore, we chose four limits (10 s, 50 s, 100 s, 250 s) that are applied to both analyses and trigger switches often, sometimes, or rarely. Additionally, we want to study whether it is advantageous if the time CoVeriTest spends in a round is not equally spread among the analyses. Thus, we come up with two additional time limit pairs: (20 s, 80 s) and (80 s, 20 s).

We combine all nine reuse types with the six time limit pairs, which results in 54 CoVeriTest configurations. All 54 configurations aim at generating tests to cover the assume edges of a program.

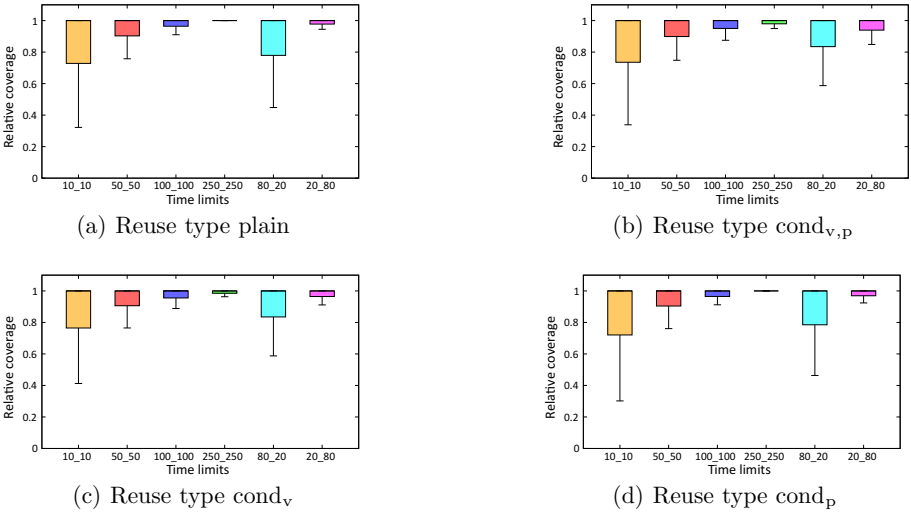
Tools. For CoVeriTest, we used the implementation in CPAchecker version 29 347. Moreover, we compare CoVeriTest against the two best tools VeriFuzz [26] and Klee [23] from Test-Comp'19 (in the versions submitted to Test-Comp'19[7]). The tool VeriFuzz is based on the evolutionary fuzzer AFL and uses verification techniques to compute initial input values and parameters for AFL. Klee applies symbolic execution. To compare CoVeriTest against Klee and VeriFuzz, we use the validator TBF Test-Suite Validator v1.2[8] to measure branch coverage. TBF Test-Suite Validator is based on gcov[9].

Programs. CoVeriTest, Klee, and VeriFuzz produce tests for C programs. All three tools participated in TestComp'19. Thus, for comparison of the three tools, we consider all 1 720 tasks of the TestComp'19 benchmark set[10] that support the branch-coverage property. Since we do not need to execute tests for the comparison of the different CoVeriTest configurations, we evaluated them on a larger benchmark set, which contains all 6 703 C programs from the well-established SV-benchmark set[11] in the version tagged svcomp18.

Computing Resources. We run our experiments on machines with 33 GB of memory and an Intel Xeon E3-1230 v5 CPU with 8 processing units and a frequency of 3.4 GHz. The underlying operating system is Ubuntu 18.04 with Linux kernel 4.15. As in TestComp'19, for test generation we grant each run a maximum of 8 processing units, 15 min of CPU time, and 15 GB of memory, and for test-suite execution (required to compare against Klee and VeriFuzz), the TBF Test-Suite Validator is granted 2 processing units, 3 h of CPU time, and 7 GB of memory per run. We use BenchExec [20] to enforce the limits of a run.

Availability. Our experimental data are available online[12] [13].

[7] https://gitlab.com/sosy-lab/test-comp/archives-2019/tree/testcomp19/2019
[8] https://gitlab.com/sosy-lab/test-comp/archives-2019/blob/testcomp19/2019/tbf-testsuite-validator.zip
[9] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[10] https://github.com/sosy-lab/sv-benchmarks/tree/testcomp19
[11] https://github.com/sosy-lab/sv-benchmarks
[12] https://www.sosy-lab.org/research/coop-testgen/



(a) Reuse type plain

(b) Reuse type $\text{cond}_{\text{v,p}}$

(c) Reuse type cond_{v}

(d) Reuse type cond_{p}

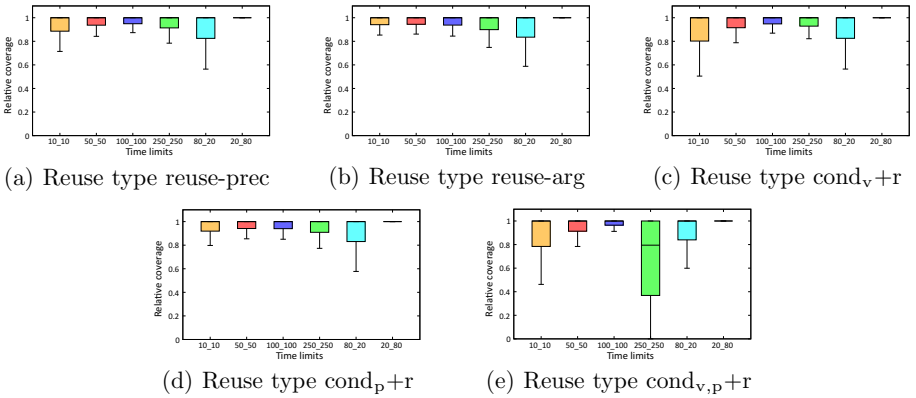
Fig. 3. Comparing relative coverage (number of covered goals divided by maximal number of covered goals) achieved by CoVeriTest configurations with different time limits. All configurations let analyses discard their own knowledge gained in previous executions.

4.2 Experiments

Claim 1 (Reduce switching when discarding own results). Four types of reuse (namely, plain, cond_{v}, cond_{p}, and $\text{cond}_{\text{v,p}}$) let the analyses discard their own knowledge from their previous executions. For each of these types, we compare the coverage achieved by all six CoVeriTest configurations that use this type[13]. More concrete, for all six CoVeriTest configurations applying the same reuse type, we first compute for each program the maximum over the number of covered goals achieved by each of these six configurations for that program. Then, for each of the six CoVeriTest configurations that use that reuse type, we divide the number of covered goals achieved for a program by the respective maximum computed. We call this measure *relative coverage* because the value is relative to the maximum and not the total number of goals. Figure 3 shows box plots per reuse type. The box plots show the distribution of the relative coverage. The closer the bottom border of a box is to value one, the higher coverage is achieved. For all four reuse types, the fourth box plot has the bottom border closest to value one. Since the fourth box plot is a configuration that grants each analysis 250 s per round (highest limit considered, only three switches), the claim holds.

Claim 2 (Favor powerful analysis when reusing own results). Five types of reuse (namely, reuse-prec, reuse-arg, cond_{v}+r, cond_{p}+r, and $\text{cond}_{\text{v,p}}$+r) let analyses reuse knowledge from their own, previous execution. Similar to the previous claim, we compute for each of these types the relative coverage of all six configurations using this particular type of reuse. For each reuse type,

[13] Note that those six configurations only differ in the analyses' time limits.



(a) Reuse type reuse-prec (b) Reuse type reuse-arg (c) Reuse type cond_v+r

(d) Reuse type cond_p+r (e) Reuse type $\text{cond}_{\text{v,p}}$+r

Fig. 4. Comparing relative coverage (number of covered goals divided by maximal number of covered goals) achieved by CoVeriTest configurations when using different time limits and a fixed reuse type. All considered configurations let analyses reuse knowledge from their own, previous execution.

Fig. 4 shows box plots of the distributions of the relative coverage. As before, a bottom border closer to value one reflects higher coverage. In all five cases, the last box plot has the bottom border closest to value one. The last box plots represent CoVeriTest configurations that grant the value analysis 20 s and the predicate analysis 80 s in each round. Since the predicate analysis, which gets more time per round, is more powerful than the value analysis, our claim is valid.[14]

Claim 3 (Better reuse own results). So far, we know how to configure time limits. Now, we want to find out how to reuse information from previous analysis runs. For each reuse type, we select from the six available configurations the configuration that performed best. Again, we use the relative coverage to compare the resulting nine configurations. Figure 5 shows box plots of the distributions of the relative coverage. The first four box plots show configurations in which analyses discard their own results, while the last five box plots refer to configurations in which analyses reuse knowledge from their own, previous executions. Since the last five boxes are smaller than the first four and their bottom borders are closer to one, the last five configurations achieve higher coverage. Hence, our claim holds. Moreover, from Fig. 5 we conclude that it is best to reuse the ARG (although cond_v+r and cond_p+r are close by).

Claim 4 (Interleave multiple analyses rather than use one of them). To evaluate whether CoVeriTest benefits from interleaving, we compare CoVeriTest against the analyses used by it. CoVeriTest interleaves value and predicate analysis. Figure 6(a) and 6(b) show scatter plots that compare for each program the coverage, i.e., number of covered goals divided by number of total goals, achieved by the best CoVeriTest configuration (x-axis) with the coverage achieved when only using either value or predicate analysis for test generation. Note that we excluded those programs from the scatter plots, for which we miss

[14] This insight is independently partially backed by a sequential combination of explicit-value analysis and predicate analysis that performed well in SV-COMP 2013 [62].

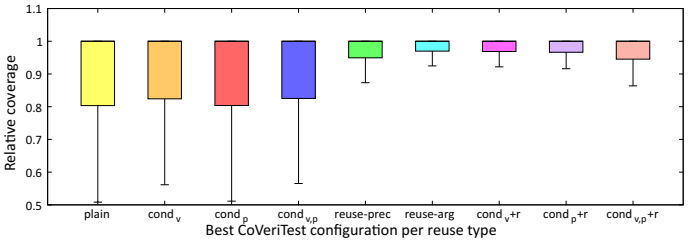
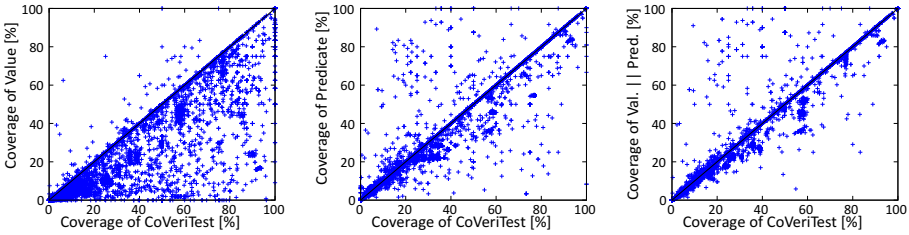


Fig. 5. Comparing relative coverage achieved by CoVeriTest configurations applying different strategies to reuse information gained by previous verifier runs.



(a) CoVeriTest (x-axis) vs. value analysis

(b) CoVeriTest (x-axis) vs. predicate analysis

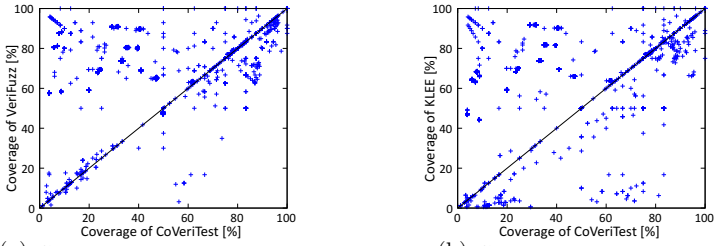
(c) CoVeriTest (x-axis) vs. value and predicate analysis in parallel

Fig. 6. Compares the coverage achieved by CoVeriTest (best configuration) with the coverage achieved when running CoVeriTest's analyses alone or in parallel

the number of covered goals for at least one test generator, e.g., due to timeout of the analysis. Figure 6(a) compares CoVeriTest and value analysis; we see that almost all points are in the lower right half. Thus, CoVeriTest typically achieves higher coverage than value analysis alone. Figure 6(b), comparing CoVeriTest with predicate analysis, is more diverse. About 54% of the points are on the diagonal, i.e., CoVeriTest and predicate analysis cover the same number of goals. The upper left half contains 19% of the points, i.e., predicate analysis alone achieves higher coverage. These points for example reflect float programs and ECA programs without arithmetic computations. In contrast, CoVeriTest achieves higher coverage in 27% of the programs. CoVeriTest is beneficial for programs that only need few variable values to trigger the branches, like ssh programs or programs from the product-lines subcategory. CoVeriTest also profits from the value analysis when considering ECA programs with arithmetic computations, since the variables have a fixed value in each loop iteration. All in all, CoVeriTest performs slightly better than predicate analysis alone.

Claim 5 (Interleave rather than parallelize). Figure 6(c) shows a scatter plot that compares for each program the coverage achieved by CoVeriTest (x-axis) and a test generator that runs the value analysis and the predicate analysis in parallel[15]. As before, we exclude programs for which

[15] The test generator uses CPAchecker's parallel algorithm and lets the two analyses share information about covered test goals.



(a) CoVeriTest vs. VeriFuzz (b) CoVeriTest vs. Klee

Fig. 7. Compares the branch coverage achieved by CoVeriTest (best configuration) with the branch coverage achieved by existing state-of-the-art test-generation tools

we could not get the number of covered goals for at least one of the analyses. Looking at Fig. 6(c), we observe that many points (60%) are on the diagonal, i.e., the achieved coverage is identical. Moreover, CoVeriTest performs better for 30% (lower right half), while approximately 10% of the points are in the upper left half. Since CoVeriTest achieves the same or better coverage results in about 90% of the cases, it should be preferred over parallelization. This is no surprise since we showed that a test generator should favor the more powerful analysis (which CoVeriTest does, but parallelization evenly distributes CPU time).

Claim 6 (CoVeriTest complementary). Our goal is to compare CoVeriTest and the two best tools of Test-Comp'19 [4]: VeriFuzz and Klee. All three tools aim at constructing test suites with high branch coverage. Thus, we use branch coverage as comparison criterion. We measure branch coverage with TBF Test-Suite Validator. Figure 7 shows two scatter plots. Each plot compares branch coverage achieved by CoVeriTest and by one of the other techniques.[16] Points in the lower right half indicate that CoVeriTest achieved higher coverage. Looking at the two scatter plots, we observe that there exist programs for which CoVeriTest performs better and vice versa. Generally, we observed that CoVeriTest has problems with array tasks and ECA tasks. We already know from verification that CPAchecker sometimes lacks refinement support for array tasks. Moreover, the problem with the ECA tasks is that CPAchecker splits conditions with conjunctions or disjunctions—which ECA tasks contain a lot—into multiple assume edges. Thus, the number of test goals is much larger than the actual branches to be covered. However, CoVeriTest seems to benefit from splitting for some of the float tasks. Additionally, CoVeriTest is often better on tasks of the sequentialized subcategory. We think that CoVeriTest benefits from the value analysis since the tasks of the sequentialized subcategory contain lots of branch conditions checking for a specific value or interpreting variable values as booleans. All in all, CoVeriTest is not always best, but is also not dominated. Thus, CoVeriTest complements the existing approaches.

[16] Note that the scatter plots only contain points that have a positive x and y value because there exist different reasons (timeout, out of memory, tool failure, etc.) why we might get no or a zero coverage value from the test validator. The plots contain points for about 98% of the 1 720 programs.

4.3 Threats to Validity

All our CoVeriTest configurations consider the same two analyses. Our results might not apply if using CoVeriTest with a different set of analyses. In our experiments, we used benchmark programs instead of real-world applications. Although the benchmark set is diverse and well-established, our results may not carry over into practice.

The validator TBF Test-Suite Validator might contain bugs that result in wrong coverage numbers. However, the validator was used in Test-Comp'19 already, and is based on the well-established coverage-measurement tool `gcov`.

For the comparison of the CoVeriTest configurations as well as the comparison of CoVeriTest with the single analyses and the parallel approach, we relied on the number of covered goals reported by CoVeriTest. Invalid counterexamples could be used to cover test goals. The analyses used by CoVeriTest apply CEGAR approaches and should detect spurious counterexamples. Moreover, these analyses run in the SV-COMP configuration of CPAchecker and are tuned to not report false results. Another problem is that whenever CPAchecker does not output statistics (due to timeout, out of memory, etc.), we use the last number of covered goals reported in the log. However, this might be an underapproximation of the number of covered goals. All these problems do not occur in the comparison of CoVeriTest with Klee and VeriFuzz, in which the coverage is measured by the validator. Thus, this comparison still supports the value of CoVeriTest.

5 Related Work

CoVeriTest interleaves reachability analyses to construct tests for C programs. To enable cooperation, CoVeriTest extracts information from ARGs constructed by previous analysis runs.

A few tools use reachability analyses for test generation. Blast [5] considers a target predicate p and generates a test for each program location that can be reached with a state fulfilling the predicate p. For test generation, Blast uses predicate abstraction. FShell [44–46] and CPA/Tiger [12] generate tests for a coverage criterion specified in the FShell query language (FQL) [46]. Both transform the FQL specification into a set of test-goal automata and check for each automaton whether its final state can be reached. FShell uses CBMC to answer those reachability queries and CPA/Tiger uses predicate abstraction.

Various combinations have been proposed for verification [2,10,11,14,25,27, 29–31,35,37,40,50,64] and test-suite generation [1,32,34,36,38,47,51,54,56,59, 60,63]. We focus on combinations that interleave approaches. SYNERGY [40] and DASH [2] alternate test generation and proof construction to (dis)prove a property. Similarly, SMASH [37] combines underapproximation with overapproximation. Interleaving is also used in test generation. Hybrid concolic testing [54] interleaves random testing with symbolic execution. When random testing gets stuck, symbolic execution is started from the current state. As soon as a new goal is covered, symbolic execution hands over to random testing providing the values used to cover the goal. Similarly, Driller [60] and Badger [56] combine fuzzing

with concolic execution. However, they only exchange inputs. Xu et al. [51,63] interleave different approaches to augment test suites. The approach closest to CoVeriTest is abstraction-driven concolic testing [32]. Abstraction-driven concolic testing interleaves concolic execution and predicate analysis. Furthermore, it uses conditions extracted from the ARGs generated by the predicate analysis to direct the concolic execution towards feasible paths. Abstraction-driven concolic testing can be seen as one particular configuration of CoVeriTest.

Also, ARG information has been reused in different contexts. Precision reuse [19] uses the precision determined in a previous analysis run to reverify a modified program. Similarly, extreme model checking [42] adapts an ARG constructed in a previous analysis to fit to the modified program. CPA/Tiger [12] transforms an ARG that was constructed for one test goal such that it fits to a new test goal. Lazy abstraction refinement [43] adapts an ARG to continue exploration after abstraction refinement. Configurable program certification [48,49] constructs a certificate from an ARG, which can be used to reverify a program. Similarly, reachability tools like CPAchecker construct witnesses [6,7] from ARGs. Conditional model checking [10,14] constructs a condition from an ARG when a verifier gives up. The condition describes the remaining verification task and is used by a subsequent verifier to restrict its exploration.

6 Conclusion

Testing is a standard technique for software quality assurance. But state-of-the-art techniques still miss many bugs that involve sophisticated branching conditions [17]. It turns out that techniques performing abstract reachability analyses are well-suited for this task. They simply need to check the reachability of every branch and generate a test for each positive check. However, in practice, for every such technique there exist reachability queries on which the technique is inefficient or fails [8]. We propose CoVeriTest to overcome these practical limitations. CoVeriTest interleaves different reachability analyses for test generation. We experimented with various configurations of CoVeriTest, which vary in the time limits of the analyses and the type of information exchanged between different analysis runs. CoVeriTest works best when each analysis resumes its exploration, different analyses only share test goals, and more powerful analyses get larger time budgets. Moreover, a comparison of CoVeriTest with (a) the reachability analyses used by CoVeriTest and (b) state-of-the-art test-generation tools witness the benefits of the new CoVeriTest approach.

CoVeriTest participated in Test-Comp 2019 [4] and achieved rank 3 (out of 9) in both categories, bug finding and branch coverage.[17]

In future, we plan to integrate further analyses, e.g., bounded model checking or symbolic execution, into CoVeriTest and to evaluate CoVeriTest on real-world applications.

[17] https://test-comp.sosy-lab.org/2019/results/

References

1. Baars, A.I., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., Vos, T.E.J.: Symbolic search-based testing. In: Proc. ASE, pp. 53–62. IEEE (2011). https://doi.org/10.1109/ASE.2011.6100119
2. Beckman, N., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Proc. ISSTA, pp. 3–14. ACM (2008). https://doi.org/10.1145/1390630.1390634
3. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS, LNCS, vol. 10206, pp. 331–349. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_20
4. Beyer, D.: International competition on software testing (Test-Comp). In: Proc. TACAS, Part 3, LNCS, vol. 11429, pp. 167–175. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_11
5. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE, pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
6. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE, pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
7. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE, pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
8. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
9. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Clarke, E.M., Henzinger, T.A., Veith, H. (eds.) Handbook on Model Checking, pp. 493–540. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_16
10. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE, pp. 57:1–57:11. ACM (2012). https://doi.org/10.1145/2393596.2393664
11. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008). http://dx.doi.org/10.1109/ASE.2008.13
12. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Information reuse for multi-goal reachability analyses. In: Proc. ESOP, LNCS, vol. 7792, pp. 472–491. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_26
13. Beyer, D., Jakobs, M.C.: Replication package for article "CoVeriTest: Cooperative verifier-based testing" in Proc. FASE 2019. Zenodo (2019). https://doi.org/10.5281/zenodo.2566735
14. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE, pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
15. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV, LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16

16. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010). http://ieeexplore.ieee.org/document/5770949/
17. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Proc. HVC, LNCS, vol. 10629, pp. 99–114. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70389-3_7
18. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE, LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37057-1_11
19. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE, pp. 389–399. ACM (2013). https://doi.org/10.1145/2491411.2491429
20. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Proc. SPIN, LNCS, vol. 9232, pp. 160–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_12
21. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Proc. SPIN, LNCS, vol. 9232, pp. 20–38. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_3
22. Beyer, D., Löwe, S., Wendler, P.: Sliced path prefixes: An effective method to enable refinement selection. In: Proc. FORTE, LNCS, vol. 9039, pp. 228–243. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19195-9_15
23. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI, pp. 209–224. USENIX Association (2008). http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
24. Chalupa, M., Vitovská, M., Strejcek, J.: SYMBIOTIC 5: Boosted instrumentation (competition contribution). In: Proc. TACAS, LNCS, vol. 10806, pp. 442–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_29
25. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: Proc. SAC, pp. 1284–1291. ACM (2012). http://doi.acm.org/10.1145/2245276.2231980
26. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: Program aware fuzzing. In: Proc. TACAS, Part 3, LNCS, vol. 11429, pp. 244–249. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_22
27. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proc. ICSE, pp. 144–155. ACM (2016). http://doi.acm.org/10.1145/2884781.2884843
28. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). http://doi.acm.org/10.1145/876638.876643
29. Cousot, P., Cousot, R.: Systematic design of program-analysis frameworks. In: Proc. POPL, pp. 269–282. ACM (1979). http://doi.acm.org/10.1145/567752.567778
30. Csallner, C., Smaragdakis, Y.: Check 'n' crash: Combining static checking and testing. In: Proc. ICSE, pp. 422–431. ACM (2005). http://doi.acm.org/10.1145/1062455.1062533
31. Czech, M., Jakobs, M.C., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE, LNCS, vol. 9033, pp. 100–114. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_7

32. Daca, P., Gupta, A., Henzinger, T.A.: Abstraction-driven concolic testing. In: Proc. VMCAI, LNCS, vol. 9583, pp. 328–347. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_16
33. D'Silva, V., Kröning, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. CAD Integr. Circ. Syst. **27**(7), 1165–1178 (2008). https://doi.org/10.1109/TCAD.2008.923410
34. Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: Proc. ISSRE, pp. 360–369. IEEE (2013). https://doi.org/10.1109/ISSRE.2013.6698889
35. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: Dynamic symbolic execution guided with static verification results. In: Proc. ICSE, pp. 992–994. ACM (2011). http://doi.acm.org/10.1145/1985793.1985971
36. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. PLDI, pp. 213–223. ACM (2005). http://doi.acm.org/10.1145/1065010.1065036
37. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: Unleashing the power of alternation. In: Proc. POPL, pp. 43–56. ACM (2010). http://doi.acm.org/10.1145/1706299.1706307
38. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proc. NDSS. The Internet Society (2008)
39. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Proc. CAV, LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
40. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: A new algorithm for property checking. In: Proc. FSE, pp. 117–127. ACM (2006). https://doi.org/10.1145/1181775.1181790
41. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL, pp. 232–244. ACM (2004). http://doi.acm.org/10.1145/964001.964021
42. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Verification: Theory and Practice, pp. 332–358. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39910-0_16
43. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279
44. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic test case generation for dynamic analysis and measurement. In: Gupta, A., Malik, S. (eds.) Proc. CAV, LNCS, vol. 5123, pp. 209–213. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_20
45. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: Proc. VMCAI, LNCS, vol. 5403, pp. 151–166. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-93900-9_15
46. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: Proc. ASE, pp. 407–416. ACM (2010). https://doi.org/10.1145/1858996.1859084
47. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: Proc. ASE, pp. 297–306. IEEE (2008). https://doi.org/10.1109/ASE.2008.40

48. Jakobs, M.C.: Speed up configurable certificate validation by certificate reduction and partitioning. In: Proc. SEFM, LNCS, vol. 9276, pp. 159–174. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_12
49. Jakobs, M.C., Wehrheim, H.: Certification for configurable program analysis. In: Proc. SPIN, pp. 30–39. ACM (2014). https://doi.org/10.1145/2632362.2632372
50. Jalote, P., Vangala, V., Singh, T., Jain, P.: Program partitioning: A framework for combining static and dynamic analysis. In: Proc. WODA, pp. 11–16. ACM (2006). http://doi.acm.org/10.1145/1138912.1138916
51. Kim, Y., Xu, Z., Kim, M., Cohen, M.B., Rothermel, G.: Hybrid directed test suite augmentation: An interleaving framework. In: Proc. ICST, pp. 263–272. IEEE (2014). https://doi.org/10.1109/ICST.2014.39
52. Lemieux, C., Sen, K.: FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proc. ASE, pp. 475–485. ACM (2018). https://doi.org/10.1145/3238147.3238176
53. Li, J., Zhao, B., Zhang, C.: Fuzzing: A survey. Cybersecurity **1**(1), 6 (2018). https://doi.org/10.1186/s42400-018-0002-y
54. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proc. ICSE, pp. 416–426. IEEE (2007). https://doi.org/10.1109/ICSE.2007.41
55. McMinn, P.: Search-based software test data generation: A survey. Softw. Test. Verif. Reliab. **14**(2), 105–156 (2004). https://doi.org/10.1002/stvr.294
56. Noller, Y., Kersten, R., Pasareanu, C.S.: Badger: Complexity analysis with fuzzing and symbolic execution. In: Proc. ISSTA, pp. 322–332. ACM (2018). http://doi.acm.org/10.1145/3213846.3213868
57. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proc. ICSE, pp. 75–84. IEEE (2007). https://doi.org/10.1109/ICSE.2007.37
58. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT **11**(4), 339–353 (2009). https://doi.org/10.1007/s10009-009-0118-1
59. Sakti, A., Guéhéneuc, Y., Pesant, G.: Boosting search based testing by using constraint based testing. In: Proc. SSBSE, LNCS, vol. 7515, pp. 213–227. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33119-0_16
60. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. NDSS. The Internet Society (2016). http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf
61. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Proc. ISSTA, pp. 97–107. ACM (2004). http://doi.acm.org/10.1145/1007512.1007526
62. Wendler, P.: CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proc. TACAS, LNCS, vol. 7795, pp. 613–615. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_45

63. Xu, Z., Kim, Y., Kim, M., Rothermel, G.: A hybrid directed test suite augmentation technique. In: Proc. ISSRE, pp. 150–159. IEEE (2011). https://doi.org/10.1109/ISSRE.2011.21
64. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: Better together! In: Proc. ISSTA, pp. 145–156. ACM (2006). http://doi.acm.org/10.1145/1146238.1146255

PARDIS: Priority Aware Test Case Reduction

Golnaz Gharachorlu(✉) and Nick Sumner

Simon Fraser University, Burnaby, BC, Canada
{ggharach,wsumner}@sfu.ca

Abstract. Test cases play an important role in testing and debugging software. Smaller tests are easier to understand and use for these tasks. Given a test that demonstrates a bug, *test case reduction* finds a smaller variant of the test case that exhibits the same bug. Classically, one of the challenges for test case reduction is that the process is slow, often taking hours. For hierarchically structured inputs like source code, the state of the art is Perses, a recent grammar aware and queue driven approach for test case reduction. Perses traverses nodes in the abstract syntax tree (AST) of a program (test case) based on a priority order and tries to reduce them while preserving syntactic validity.

In this paper, we show that Perses' reduction strategy suffers from *priority inversion*, where significant time may be spent trying to perform reduction operations on lower priority portions of the AST. We show that this adversely affects the reduction speed. We propose PARDIS, a technique for priority aware test case reduction that avoids priority inversion. We implemented PARDIS and evaluated it on the same set of benchmarks used in the Perses evaluation. Our results indicate that compared to Perses, PARDIS is able to reduce test cases 1.3x to 7.8x faster and with 46% to 80% fewer queries.

Keywords: Test case reduction · Automated debugging · Priority aware reduction

1 Introduction

Test case reduction is a technique that aids in testing and debugging software. When an input for a program causes the program to exhibit a property of interest, like a bug, finding a smaller input that also exhibits the property can help to explain the behavior [1–3]. Given an input $I \in \mathbb{I}$ and an oracle $\psi : \mathbb{I} \rightarrow \mathbb{B}$ that performs a test and returns true iff a property holds, test case reduction aims to find a smaller input I' such that $\psi(I') = \text{true}$. Often, this problem is approached through Delta Debugging (DD), a longstanding and effective algorithm for test case reduction that essentially generalizes binary search [2]. However, for inputs with significant structure, generic DD can perform poorly, requiring significant time and not performing much reduction [3,4]. For compilers in particular, where

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 409–426, 2019.
https://doi.org/10.1007/978-3-030-16722-6_24

the inputs must be valid programs, this has led to specialized techniques like Hierarchical Delta Debugging [3,4], language specific reducers like C-Reduce [5], and most recently to Syntax Guided Program Reduction as seen in Perses [6].

Syntax Guided Program Reduction (SGPR) is the present state of the art for compiler targeted test case reduction. The intuition behind SGPR is that the grammar defining the language of inputs eliminates many invalid sub-inputs from the search space. For example, when an input must adhere to the C programming language [7], removing the return type of a function declaration would not be valid because the C grammar specifies that the return type is required. Such syntactically invalid inputs are removed from the search space by SGPR.

Perses, a form of SGPR, takes as arguments not only a program p and oracle ψ, but also the context free grammar G of valid inputs [6]. It transforms the grammar so that removable parts of the input can be identified by the names of the grammar rules used to parse them. This also normalizes the grammar so that all removable components are expressed through quantifiers in an extended context free grammar [8], i.e. optionality (?) and lists (*, +). This transformation is illustrated in Fig. 1. Notice, for instance, that the recursive rule BAR denoting a list is transformed ($\Longrightarrow$) into a Kleene-+ quantified list. Individual elements of the list may be removed while preserving syntactic validity. Perses then parses the input of interest into an abstract syntax tree (AST) and traverses the AST while trying to (1) remove optional nodes and (2) perform DD to minimize the children of nodes representing lists. The grammar transformations have the benefit of making many syntactically correct removals easy and efficient to locate.

$$\mathsf{FOO} \rightarrow \mathsf{a} \mid \mathsf{a\ b} \Longrightarrow \begin{array}{l}\mathsf{FOO} \rightarrow \mathsf{a\ FOO\_opt}\\ \mathsf{FOO\_opt} \rightarrow \mathsf{b?}\end{array}$$

(a) Optional elements like b are refactored into rules with ? quantifiers.

$$\mathsf{BAR} \rightarrow \mathsf{c} \mid \mathsf{c\ BAR} \Longrightarrow \begin{array}{l}\mathsf{BAR} \rightarrow \mathsf{BAR\_plus}\\ \mathsf{BAR\_plus} \rightarrow \mathsf{c+}\end{array}$$

(b) Lists of elements are refactored into rules with * or + quantifiers.

Fig. 1. Overview of Perses grammar transformations for SGPR.

Perses has significantly improved the speed of program reduction. However, it still takes several hours to reduce some inputs. Consider the code in Listing 1.1 along with its AST in Fig. 3. This example is similar to a C program generated by the compiler testing tool CSmith [9]. In this example, Perses first considers the root node with ID (1) of the AST. Since the rule for this node ends in _star, it is a list node, and its children are the elements of the list. Thus, Perses applies DD to the list of children for node (1) to minimize the number of children. When such lists are long, significant time can be devoted to this task. We show in Sect. 4 that this can lead to substantial *stalls* in reduction, where no progress is made while a list is being processed. However, most of the children of this node have low *token weight*, the number of tokens beneath a given node that is denoted by `w:` in Fig. 3. Indeed, greater value would be found by focusing

on just *one* of its children, node ⑤, which contains the majority of the input beneath it. By spending greater effort up front on portions of the AST of lesser value, Perses suffers from a form of *priority inversion*. Priority inversion occurs when a low priority task is scheduled instead of a high priority task. In this case, Perses focuses on removing low token weight nodes instead of high token weight nodes. Indeed, Perses may even fail to remove elements that would enable better reduction success overall. In this case, the declarations of `foo`, `S`, and `d` are used within the code beneath node ⑤. Thus, those uses need to be eliminated *before* any of the declarations can be removed successfully. In practice, we find that priority inversion has a significant impact on reduction time in SGPR.

To address priority inversion, we have developed *priority aware reduction strategies* for program reduction. By focusing the reduction effort on the nodes of the AST that cover the greatest number of tokens, we prioritize reduction of the most complex parts of the input first. This has multiple important benefits: (1) Dependencies between program elements are more likely to be broken by eliminating the complex uses first. (2) Stalls in reduction from unsuccessful rounds of DD can be mitigated. (3) By removing large portions of an input earlier on, each oracle query to ψ can take less time because smaller inputs tend to be faster to check. We have designed and evaluated a tool, PARDIS, that makes use of these techniques and found that it leads to consistent and significant performance improvements over Perses on the Perses benchmarks [6].

In summary, this paper makes the following contributions:

1. **Priority awareness.** We identify *priority inversion* as a key problem facing SGPR techniques and develop priority aware reduction strategies as a potential solution. *Priority aware reduction strategies* focus the reduction effort on the complex portions of an input first, enabling earlier and thus faster test case reduction (Sects. 3, 4.1).
2. **Optimization.** We identify redundancies in the reduction process when using Perses' transformed grammars and develop a solution to prune them from the candidate search space (Sect. 3.2).
3. **Significant performance improvement.** We implemented our strategies in a tool, PARDIS, and evaluated it on the same benchmarks used by Perses. Experimental results show that PARDIS both removes more of the input earlier on and is faster overall. Compared to Perses, PARDIS reduces test cases 1.3x to 7.8x faster and with 46% to 80% fewer oracle queries (Sect. 4.1).

2 Background and Motivation

Consider again the example in Fig. 3 and suppose that the oracle (ψ) checks that this program p should print `"Hello World!"` on line 24 (marked with $*$). Thus, the smallest subprogram for which ψ returns true is the main function with the desired `print` statement.

To search for this smaller input inside the original input, Perses traverses the AST using a priority queue ordered by the token weight. In each trial, the node

Listing 1.1: A C program with property of interest on line 24.

```
1  double d = 0.10;
2  struct S {
3    int f1;
4    int f2;
5  };
6  void foo(struct S s, char str[]){
7    double v = s.f2 + s.f2 * d;
8    printf("%s %f\n",str,v);
9  }
10 int main() {
11   unsigned int a = 1;
12   char b[] = "first";
13   char c[] = "second";
14   if (a) {
15     struct S s1;
16     s1.f1 = 1;
17     s1.f2 = 4000;
18     struct S s2;
19     s2.f1 = 2;
20     s2.f2 = 2000;
21     foo(s1, b);
22     foo(s2, c);
23   }
24   printf("Hello World!\n"); (*)
25   return 0;
26 }
```

node(s) to remove	removed
{2,3,4,5}	F
{2,3}	F
{4,5}	F
{2}	F
{3}	F
{4}	F
{5}	F
{3,4,5}	F
{2,4,5}	F
{2,3,4}	F
{2,3,5}	F
{8,9,10,11,12,13}	F
{8,9,10}	F
{11,12,13}	F
{8,9}	F
{10,11}	T
{12,13}	F
{8}	T
{9}	T
{12}	F

(a) Perses

node to remove	removed
{1}	F
{5}	F
{7}	F
{11}	T
{4}	T
{3}	T
{10}	T
{9}	T
{8}	T
{12}	F
{2}	T

(b) Pardis

node(s) to remove	removed
{1}	F
{5}	F
{7}	F
{11}	T
{4}	T
{3}	T
{9,10}	T
{8}	T
{12}	F
{2}	T

(c) Pardis Hybrid

Fig. 2. One round of removal trials in Perses, Pardis and Pardis Hybrid for the AST in Fig. 3. Numbers are node IDs.

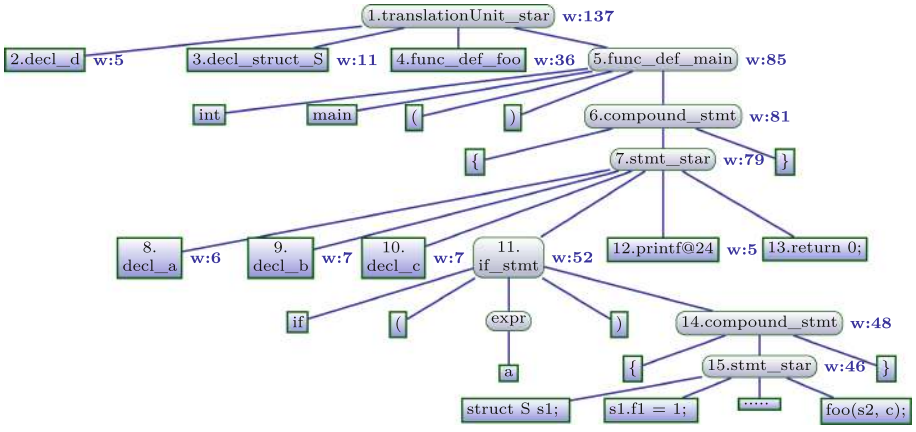


Fig. 3. AST of the program in Listing 1.1. `w` denotes the token weight of each node.

with the maximum weight is removed from the work queue and traversed. In our example, the queue starts out containing only the root of the AST, node ①. Perses performs specific reduction operations on different types of nodes during traversal. For instance, on optional nodes, Perses tries to remove the optional child node. For list nodes, Perses minimizes the list of children using DD. Any

remaining children of the traversed node are then added to the priority queue in order to be traversed in the future.

Observe that in this example, Perses will first examine node ① and remove it from the queue. Because ① is a list node, DD is applied to the children of ①. Different combinations of children are removed from ① and the result is checked by ψ to find a smaller input. First, all children are removed and ψ is checked. After this fails, the first half of the children (② and ③) are removed, but ψ returns false again because this removes required declarations. Since removing the second half of the children (④ and ⑤) also fails, the process continues recursively. First DD tries shrinking the list by *removing* each individual child, and next it tries *only keeping* each individual child. Ultimately none of the trials succeed, so all children are added to the queue, and reduction continues with node ⑤. The intervening node ⑥ is not tested by SGPR because it is not syntactically removable. The next node removed from the work queue is node ⑦. This continues until the queue is empty. The precise trials exercised in this process are illustrated in Fig. 2(a). Note that 16 steps elapse until a successful trial occurs.

While the priorities used by Perses are controlled by the token weight, they determine how the *children* of the traversed nodes are removed. Thus, any node whose *parent* in the AST is a list is given the same priority as all other elements in the list. This is because DD recursively tries to minimize the entire list until no single element can be removed, regardless of the priorities of individual list elements. As a result, Perses must employ DD on the entirety of the children of ① even though it would be more beneficial to focus on just one child, node ⑤.

Instead, PARDIS more directly models the priorities. We note that in an optional or list node, such as ①, each child may be removed in a syntactically valid fashion. We call such removable nodes *nullable*. When traversing a nullable node in the AST, we can simply try directly to remove it, adding its children if the removal fails. For instance, in the running example, we would visit ① first. Because ① cannot get removed, we would simply add its children to the priority queue. Note that all children of ① are nullable, but ⑤ has the highest *token weight*. Thus, we next select ⑤ to traverse but removing ⑤ also fails. From the given token weights, we next traverse ⑥, which is syntactically not removable, and then ⑦, which we attempt to remove but is unsuccessful. Next ⑪ is visited and successfully removed. Removing ⑪ *enables the removal of* ④, ③ *and* ②. Thus, they are removed in a single pass of the tree using PARDIS, whereas Perses would require multiple traversals of the AST to remove them. This process continues until the desired output is achieved. As seen in Fig. 2(b), just 4 steps elapse until the first successful trial removes node ⑪.

Note that in this example, PARDIS is able to reduce to the desired output in a *single pass*, while Perses requires multiple passes of the AST. In practice, all program reduction techniques continue until a fixed point is reached, including PARDIS, however PARDIS can achieve greater reduction in a single traversal of the AST, accelerating convergence on the fixed point.

This priority aware approach can still have drawbacks, however. After focusing on the highest priority nodes, there may be many lower priority nodes remaining. For example, there are multiple remaining nodes of weight 7 in the tree after

performing the reduction by Pardis as described above. We also show experimentally that these lower priority nodes occur in practice in Sect. 5. The above approach of Pardis considers each node *one at a time*, which can have poor performance when reducing such long lists. In addition, we thus propose a *hybrid* approach that still prioritizes nodes by maximum token weight but also uses a list based reduction technique for spans of nodes that have *the same* token weight. This hybrid approach is able to achieve the benefits of being priority aware while still avoiding the cost of considering each node of the AST individually.

Section 3 presents the algorithms behind these techniques in detail.

3 Approach

Recall that the core of Pardis, similar to Perses, maintains a priority queue of the nodes in an AST and traverses the nodes in order to process them. It also makes use of Perses Normal Form, the result of the grammar transformations that Perses introduced [6]. The key difference is that instead of using the token weight of a parent node to determine when its nullable children may be removed, Pardis identifies all nullable nodes (see Sect. 3.2) and uses their token weights directly to prioritize the search. The core algorithm for this process is quite straightforward and presented in Algorithm 1.

Algorithm 1: Priority queue driven program reduction.

```
   Input: P : ℙ – The program to reduce as an AST
   Input: ψ : ℙ → 𝔹 – Oracle for the property to preserve
   Input: ρ : 𝕍 → ℕ × ··· × ℕ – Prioritizer for AST nodes
   Result: A minimum program p ∈ ℙ s.t. ψ(p)
1  work ← MaxPriorityQueue({p.root}, ρ)
2  while !work.empty() do
3      node ← work.takeMax()
4      if node.isNullable && ψ(p − node) then
5          p ← p - node
6      else
7          work.insert(node.children)
8  return p
```

Line 1 of the algorithm constructs the priority queue (a max-heap), initializing it with the root of the AST and using a parameterizable priority ρ. ρ is simply a function that takes a node and returns its priority as a tuple. The priority queue selects the element with a lexicographically maximal priority, so ties on the *first* element of the priority tuple are broken by the *second* element and so on. As seen in Fig. 4, for Pardis, ρ_{Pardis} returns a pair of numbers, the token weight of the node and the position of the node in a decreasing, right-to-left, breadth first search. The specific breadth first order means that for an AST with n nodes, bfsOrder(p.root)=n, the last child c of p.root has bfsOrder(c)=n-1, and so on. Thus, if several nodes have the same token weight, the one highest in the AST and furthest to the right is selected next. This ordering decreases the chances of trying to remove a declaration before its uses [10].

Line 2 starts the core of the algorithm. While there are more nodes to explore in the queue, the node with the next highest priority is considered. If it is nullable

and can be successfully removed, we remove it from the AST, otherwise we add its children to the queue so that they will also be traversed.

While the algorithm is surprisingly simple, we have found it to perform significantly better than the state of the art in practice. As we explore in Sect. 4.2, this results from prioritizing the search toward those portions of the input where reduction can have the greatest impact. To more closely compare with Perses, consider a version of Perses that upon visiting a list or optional node only tries removing each child of that node once[1]. This "one node at a time" variant of Perses can also be implemented using Algorithm 1 by carefully choosing the priority formula ρ. Because Perses considers removing the *children* of the nodes it traverses, it actually prioritizes the work queue using the token weight of the *parent* rather than the token weight of nullable nodes being considered for removal. This leads to the alternative prioritizer ρ_{perses} presented in Fig. 4. Observe that all children of a list node receive the same token weight, that of the entire list. This can inflate the priority of some nodes in the work queue and leads to poor performance.

Definitions:
tokensBelow(n) – returns the number of tokens beneath an AST node n.
bfsOrder(n) – returns the position of an AST node n in a decreasing, right to left, breadth first search.

Prioritizers:

$$\rho_{\text{PARDIS}} = (\text{tokensBelow(n)}, \text{bfsOrder(n)})$$

$$\rho_{perses} = \begin{cases} \textbf{let } \text{parentWeight} \leftarrow \text{tokensBelow(n.parent)} \textbf{ if } \text{n.parent} \textbf{ else } \infty \textbf{ in} \\ \textbf{let } \text{parentOrder} \leftarrow \text{bfsOrder(n.parent)} \textbf{ if } \text{n.parent} \textbf{ else } \infty \textbf{ in} \\ (\text{parentWeight}, \text{parentOrder}) \end{cases}$$

$$\rho_{\text{PARDIS HYBRID}} = \begin{cases} \textbf{let } \text{parentOrder} \leftarrow \text{bfsOrder(n.parent)} \textbf{ if } \text{n.parent} \textbf{ else } \infty \textbf{ in} \\ (\text{tokensBelow(n)}, \text{parentOrder}, \text{bfsOrder(n)}) \end{cases}$$

Fig. 4. Prioritizers used for PARDIS, node at a time Perses, and PARDIS HYBRID.

Like other program reduction algorithms [3,5,6,11,12], Algorithm 1 is used to compute a fixed point. That is, in practice the algorithm is repeated until no further reductions can be made. As in prior work, we omit this from our presentation for clarity. In theory, this means that the worst case complexity of the technique is $O(n^2)$ where n is the number of nodes in the AST. This arises when only one leaf of the AST is removed in each pass through the algorithm. In practice, most nodes are not syntactically nullable, and we show in Sect. 4.1 that performance of PARDIS exceeds the state of the art.

In addition, while we focus on *removing* nodes of the AST, Perses also tries to *replace* non-list and -optional nodes with compatible nodes in their subtrees. We do not focus on this aspect of the algorithm. In practice, we found it to

[1] We compare against *both* versions of Perses in Sect. 4.1.

significantly hurt performance (see Sect. 4.1) and we consider efficient replacement strategies to be orthogonal to and outside the scope of this work.

3.1 Pardis Hybrid

The initial priority aware technique from Algorithm 1 can also encounter performance bottlenecks, however. The original motivation for using DD on lists of children in the AST was that its best case behavior is $O(log(n))$ where n is the number of children in the list. This is because it tries removing multiple children at the same time. Processing one node at a time, however, requires that every list element is considered individually, guaranteeing $O(n)$ time for one round of Algorithm 1. Priority aware reduction that proceeds one node at a time faces a different set of inefficiencies that can still cause stalls in the reduction process.

Thus, we desire a means of removing multiple elements from lists at the same time while *still* preserving priority awareness. In order to achieve this, we developed Pardis Hybrid, as presented in Algorithm 2. This approach uses a modified prioritizer as presented in Fig. 4 that first orders by token weight, then by parent traversal order, then by node traversal order. The effect this has is that all children of the same parent with the same weight are grouped together. As a result, we can remove them from the priority queue together and perform list based reduction (like DD) to more efficiently remove groups of elements in a list that have the same priority (for instance, nodes ⑨ and ⑩ get removed as a group in one trial using Pardis Hybrid as shown in Fig. 2(c)). Because the search is still primarily directed by the token weights of the removed nodes, the technique still fully respects the priorities of the removed nodes.

Algorithm 2: Pardis Hybrid algorithm with priority aware list reduction.

Input: $p : \mathbb{P}$ – The program to reduce as an AST
Input: $\psi : \mathbb{P} \to \mathbb{B}$ – Oracle for the property to preserve
Result: A minimum program $p \in \mathbb{P}$ s.t. $\psi(p)$

```
1 work ← MaxPriorityQueue({p.root}, ρ_PardisHybrid)
2 while !work.empty() do
3     nodes ← work.takeWithSameWeightAndParent()
4     nullable, nonnullable ← partitionNullable(nodes)
5     removed, retained ← minimize(p, nullable, ψ)
6     p ← p - removed
7     work.insert(⋃_{x∈retained∪nonnullable} x.children)
8 return p
```

Similar to the previous approach, line 1 of Algorithm 2 starts by creating the priority queue. Note that it specifically uses the prioritizer $\rho_{\text{Pardis Hybrid}}$, which groups children having the same token weight in the priority queue. As long as there are more nodes to consider, line 3 takes all nodes from the queue with the same weight and parent. If the weight of a node is unique, this simply returns a list of length 1. Line 4 filters out non-nullable nodes from the trial, and line 5 just applies list based reduction to any nullable nodes. Lines 6 and 7 then remove the eliminated nodes from the tree and add the children of remaining nodes to the work queue. Again, this algorithm actually runs to a fixed point.

While the worst case behavior of DD is $O(n^2)$ [2], this can be improved to $O(n)$ by giving up hard *guarantees* on minimality [13]. Since this reduction process is performed to a fixed point anyway, minimize on line 5 makes use of this $O(n)$ approach to list based reduction (OPDD) without losing 1-minimality. As a result, the theoretical complexity of PARDIS HYBRID is the same as PARDIS.

3.2 Nullability Pruning

Finally, we observed that many oracle queries were simply unnecessary. Specifically, recall that a node can be tagged nullable because it is an element of a list or a child of an optional node, as previously defined by Perses grammar transformations [6]. The complete algorithm for this tagging is in *TagNullable* of Algorithm 3. However, for example, a list of one element could contain another list of one element. In the AST, this appears as a chain of nodes, at least two of which are nullable. Removing *any one* of these nodes removes the same tokens from the AST. Thus, it is only necessary to select a single nullable node from any *chain* of nodes, and the others can be disregarded.

We exploit this through an optimization called *nullability pruning*. We traverse every chain of nodes in the AST, preserving the nullability of the highest node in the chain and removing nullability from those below it. The complete algorithm is presented in *PruneNullable* of Algorithm 3. In effect, it is just a depth first search that removes redundant nullability from nodes along the way instantaneously.

In practice, we find that this can statically (ahead of time) prune most of the AST from the search space. Specifically, in the benchmarks we examine in Sect. 4, we find that of 1,593,875 total nullable nodes, 17% are redundant optional nodes and 44% are redundant list element nodes. We observe the impact of this pruning on the actual reduction process in Sect. 4.1.

Algorithm 3: Nullability tagging and pruning.

```
1  Function TagNullable(p)
       Input: p : ℙ – The program to reduce as an AST
2      foreach Node n ∈ p do
3          if n ∈ KleeneStar ∪ KleenePlus ∪ Optional then
4              foreach c ∈ n.children do c.isNullable ← true
5  Function PruneNullable(p)
       Input: p : ℙ – The program to reduce as an AST
6      Function OptimizeBelow(n)
7          hasNullable ← false
8          Loop
9              if hasNullable then
10                 n.isNullable ← false
11             else if n.isNullable then
12                 hasNullable ← true
13             if 1 == |n.children| then
14                 break
15             n ← n.getOnlyChild()
16         foreach c ∈ n.children do OptimizeBelow(n)
17     OptimizeBelow(p.root)
```

4 Evaluation

We evaluate PARDIS's performance and examine the impact of priority inversion on reduction by answering the following research questions:

- **RQ1.** How does PARDIS perform compared to Perses in terms of reduction time and speed, number of oracle queries, and size of the reduced test case?
- **RQ2.** Does priority inversion adversely affect the reduction efficiency? In particular, does reduction require more work with a traversal order suffering from priority inversion?

4.1 RQ1. Performance: PARDIS vs. Perses

Experimental Set-Up. We evaluate PARDIS on the set of C test cases used in the evaluation of Perses, including the oracle scripts provided by authors of Perses. While using these, we observed that they still allowed for some undefined behavior [5,14], so we updated all oracles to reject test case variants with undefined behavior. As a result, we were able to reproduce bugs for 14 out of 20 original test cases. The remaining benchmarks that could not reproduce their original failures were elided for this study. Since the implementation of Perses' components is not publicly available, we implemented the Perses grammar transformations and reduction based on the algorithms available in the paper [6] using the C++ bindings of ANTLR [15]. All of our implementations have been made available[2]. Our experiments were conducted on an Intel Xeon E5-2630 CPU and 64 GB memory running Ubuntu.

Variants of Reduction Techniques. To better explain performance differences, we benchmark several algorithms that each add one difference. All approaches compute fixed points as previously described.

- *Perses DD*- The removal-based algorithm of Perses that applies DD on children of list nodes [6].
- *Perses OPDD*- The same as Perses DD but using the $O(n)$ reduction algorithm of OPDD [13]. It is faster than Perses DD in practice.
- *Perses N*- The one node at a time Perses that does not apply DD on list elements but removes them one by one using Perses' parent oriented priorities.
- PARDIS W/O PRUNING- This uses the PARDIS algorithm but does not apply nullability pruning optimization proposed in Sect. 3.2.
- PARDIS- Our proposed removal algorithm that also applies nullability pruning.
- PARDIS HYBRID- The hybrid version of PARDIS with nullability pruning and OPDD as its version of DD.

[2] https://github.com/golnazgh/PARDIS.

Table 1. Original and reduced test case size and number of oracle queries.

Bug	$O(\#)$	Perses DD		Perses OPDD		Perses N		PARDIS w/o PRUNING		PARDIS		PARDIS HYBRID	
		$R(\#)$	$Q(\#)$	$R(\#)$	$Q(\#)$	$R(\#)$	$Q(\#)$	$R(\#)$	$Q(\#)$	$R(\#)$	$Q(\#)$	$R(\#)$	$Q(\#)$
clang-22382	21,068	597	5,323	597	4,865	354	3,203	354	2,702	354	2,011	354	2,319
clang-22704	184,444	250	4,181	250	3,775	220	5,083	236	4,956	236	4,342	236	2253
clang-23309	38,647	1,624	8,688	1,624	8,095	1,522	6,106	1,726	4,618	1,726	3,004	1,726	3,684
clang-25900	78,960	618	4,455	618	4,020	600	2,816	618	2,343	618	1,652	618	1,997
clang-27137	174,538	725	9,035	725	8,299	681	6,858	807	5,889	807	4,293	807	4,891
clang-27747	173,840	379	3,171	379	2,845	311	1,773	313	1,418	313	1,074	308	1,218
clang-31259	48,799	821	4,457	821	4,073	821	3,282	538	2,464	538	1,662	538	1,853
gcc-64990	148,931	776	5,913	776	5,438	1,215	5,165	776	3,781	776	2,632	776	3,148
gcc-65383	43,942	462	5,503	462	5,002	486	3,502	598	2,559	598	1,839	598	2,204
gcc-66186	47,481	1,176	6,101	1,176	5,727	1,178	4,532	1,176	3,944	1,176	2,562	1,176	3,167
gcc-66375	65,488	1,232	7,989	1,232	6,780	1,198	4,202	1,232	4,512	1,232	3,036	1,232	3,851
gcc-70127	154,816	600	5,610	600	5,201	593	3,700	600	3,063	600	2,240	600	2,723
gcc-70586	212,259	1,583	7,671	1,583	7,276	1,489	5,582	1,497	5,233	1,497	3,491	1,497	4,318
gcc-71626	6,133	58	1,151	58	1,135	58	1,013	58	330	58	264	58	228
geomean	70300	609	5126	609	4705	583	3670	574	2881	574	2066	574	2270
median	72,224	672	5,556	672	5,102	640	3,951	609	3,422	609	2,401	609	2,521

O, R and Q denote number of tokens in the original test case, reduced one and total number of oracle queries performed by the reduction technique, respectively.

Reduction Performance. We compare these techniques in terms of *the number of oracle queries* (Q), *reduction quality* or size of the final reduced test case (R), *reduction time* (T), and *reduction speed* or the average number of tokens removed per second (E). Results are presented in Tables 1 and 2. The best values of queries, time, and speed are highlighted for each test case. As can be seen, in all cases, either PARDIS or PARDIS HYBRID outperform all variants of Perses. Compared to the full removal-based Perses algorithm (Perses DD), our proposed algorithms reduce **1.3x** to **7.8x** faster and with **46%** to **80%** fewer queries. The results across variants suggest that these benefits arise from priority awareness and nullability pruning. Due to fixed point computation, all approaches produce test

Table 2. Reduction time and speed for different variants of reduction techniques.

Bug	Perses DD		Perses OPDD		Perses N		PARDIS w/o PRUNING		PARDIS		PARDIS HYBRID	
	$T(s)$	$E(\#/s)$	$T(s)$	$E(\#/s)$	$T(s)$	$E(\#/s)$	$T(s)$	$E(\#/s)$	$T(s)$	$E(\#/s)$	$T(s)$	$E(\#/s)$
clang-22382	3,198	6	3,122	7	3,489	6	3,057	7	2,977	7	2,094	10
clang-22704	1,527	121	1,304	141	5,243	35	3,323	55	3,219	57	1,160	159
clang-23309	2,571	14	2,414	15	1,920	19	1,423	26	1,007	37	1,062	35
clang-25900	1,375	57	1,220	64	1,025	76	690	114	526	149	518	151
clang-27137	6,972	25	6,379	27	5,717	30	4,428	39	3,423	51	3,538	49
clang-27747	1,194	145	1,060	164	771	225	571	304	463	375	453	383
clang-31259	1,698	28	1,577	30	1,471	33	1,239	39	814	59	800	60
gcc-64990	1,980	75	1,768	84	1,981	75	1,237	120	932	159	916	162
gcc-65383	1,762	25	1,615	27	1,304	33	892	49	704	62	699	62
gcc-66186	1,583	29	1,493	31	1,299	36	1,016	46	691	67	741	62
gcc-66375	2,782	23	2,568	25	1,851	35	1,705	38	1,173	55	1,311	49
gcc-70127	3,083	50	2,812	55	2,265	68	1,520	101	1,124	137	1,173	131
gcc-70586	4,417	48	4,119	51	3,450	61	2,545	83	1,791	118	1,984	106
gcc-71626	156	39	156	39	206	29	57	107	54	112	20	304
geomean	1900	36	1750	40	1740	40	1202	58	933	75	807	86
median	1,871	34	1,692	35	1,886	35	1,331	52	970	64	989	84

T is reduction time in seconds. E is the efficiency of removal (number of tokens removed per second).

cases from which no one token can be removed while satisfying ψ (1-minimal) [2], but they can produce different final reduced test cases [2]. On average, Pardis yields reduced test cases with 574 tokens compared to Perses DD with 609 tokens.

In addition, we graphed the reduction progress of each test case for the different variants. Fig. 5 shows the percentage of remaining tokens over time during reduction. For sake of space, we only include graphs for six of the test cases. Note that the y-axis is log scaled. Pardis and Pardis Hybrid show much faster convergence to a reduced test case compared to Perses variants. Recall that the only factor differentiating Perses N from Pardis w/o Pruning is the order in which the queue of nodes is traversed. Unlike Perses N, Pardis w/o Pruning does not suffer from priority inversion and guides the reduction process based on token weights of the nodes to remove. As can be seen, this advantage leads to faster convergence to a reduced test case. We examine the impact of priority inversion on reduction speed more rigorously in Sect. 4.2.

Replacement. As mentioned in Sect. 3, Perses also considers a replacement strategy for non-list or -optional nodes in addition to removal for other nodes. For instance, in Fig. 3, Perses will attempt to replace node ⑥ with node ⑭ because they both match the same grammar rule (compound_stmt). This replacement fails since required declarations will get removed and ψ will return false.

Including replacement significantly increases the work done by reduction. For completeness, we implemented Perses DD with replacement as described in their paper [6] and defined a four-hour timeout for the reduction process. In 11 out of 14 cases, Perses DD with replacement could not finish the reduction process before reaching the timeout. In the remaining three, it generated reduced test cases with the same size or slightly smaller while performing a significantly larger number of oracle queries (more than 3× over Perses DD without replacement).

4.2 RQ2. The Impact of Priority Inversion

As shown in Fig. 5, avoiding priority inversion leads to faster convergence. One explanation for this is that priority awareness may decrease the amount of work required to remove a token (as seen in the motivating example). We explore this in a case study on `gcc-64990` with 148,931 tokens. The *number of removal attempts* for a token is number of times a single token is considered for removal. Removing any ancestor of a token in the AST will remove that token, so if a first attempt fails, a deeper ancestor may be attempted. We compute this for every token of the test case to get a sense of the work required for each token. A better traversal order of the AST should cause fewer overall token removal attempts. To measure only the impact of different traversal orders, we compare Pardis w/o Pruning with Perses N. As described in Sect. 4.1, they follow the exact same reduction rules and differ only in their traversal orders.

Figure 6 depicts histograms of the distributions of token removal attempts for Pardis w/o Pruning and Perses N. For clearer visualization, we show only the distributions for the number of attempts less than or equal to 20. We can see how Perses N distribution is inclined toward a larger number of removal attempts,

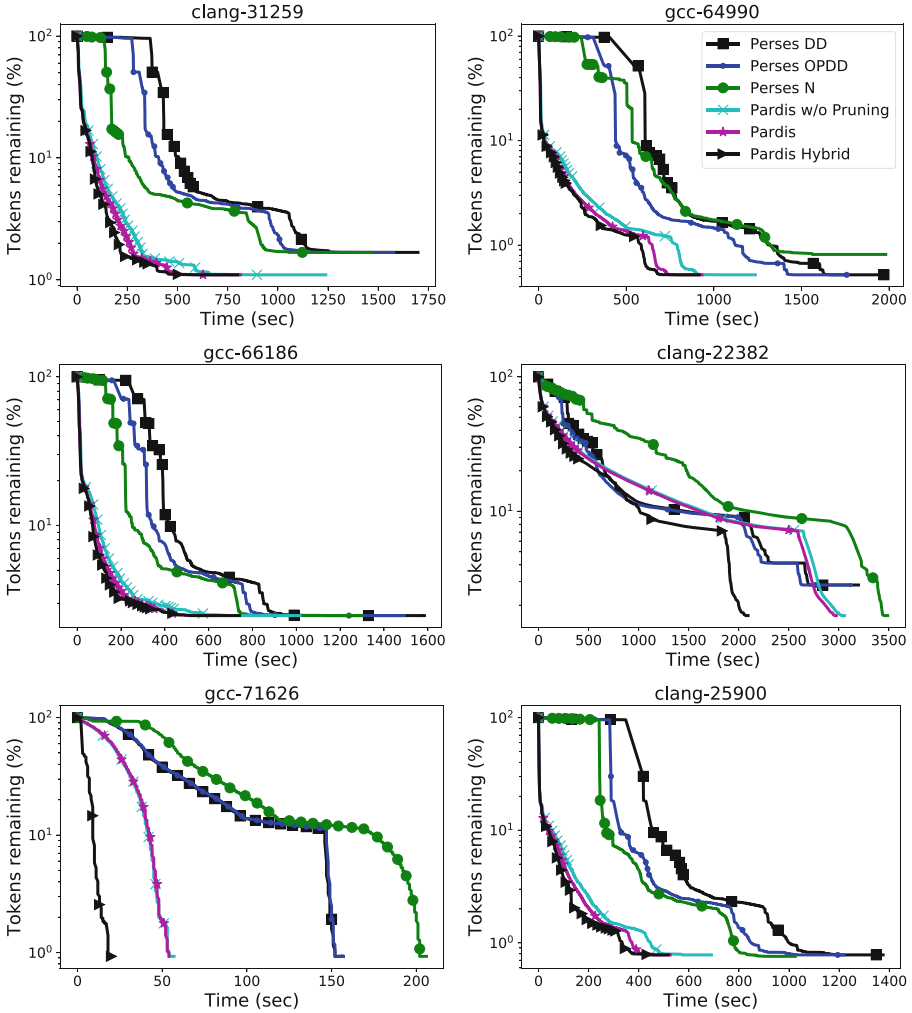


Fig. 5. Converging to a reduced test case in six variants of reduction techniques.

an indicator of more work required in order to remove individual tokens. In addition, we statically measure that the difference between the removal attempt distributions is significant. We use a one sided Wilcoxon rank-sum test [16] to determine whether the distribution of Perses N is indeed greater than that of PARDIS W/O PRUNING. The p-value computed for our data was less than $2.2e^{-16}$ which strongly supports this observation.

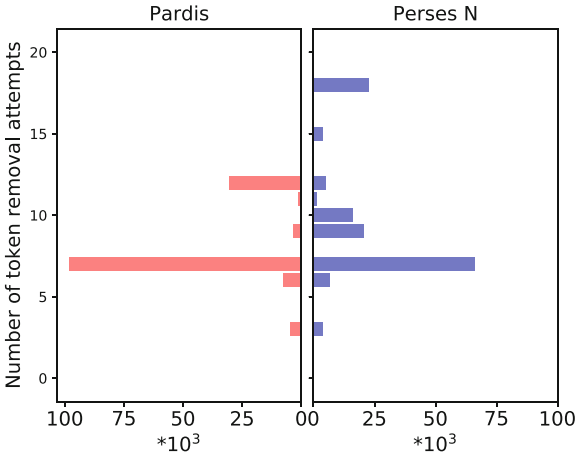


Fig. 6. Distributions of token removal attempts for Pardis w/o Pruning and Perses N.

5 Discussion

Pardis Hybrid **as a *sweet spot* in reducing test cases:** As discussed earlier, unlike Perses, Pardis Hybrid does not suffer from priority inversion because it prioritizes the search primarily on the token weight of nodes being considered for removal. Moreover, unlike Pardis, it does not strictly remove one node at a time and allows the removal of nodes with the same weight and the same parent as a group. Hence, it can be considered a sweet spot in reducing test cases. We conduct two studies that can further explore this idea.

(1) Oracle Verification Time. The number of oracle queries is a common metric used in similar studies to reason about reduction efficiency since it directly impacts the total reduction time [2,3,6,13,17]. For instance, both Pardis and Pardis Hybrid perform fewer oracle queries and take less time than Perses. However, the number of oracle queries is not the only factor involved. The time required to run each of these queries, or *oracle verification time*, also affects the total running time. For instance, as presented in Sect. 4.1, Pardis has the smallest number of oracle queries in 12 out of 14 test cases. However, in terms of total reduction time and speed, Pardis Hybrid is the fastest in 8 out of 14 cases, even while performing *more* queries compared to Pardis in 6 of them. Oracle verification time can depend on multiple elements such as the size and complexity of the test case. Since Pardis Hybrid takes advantage of the possibility to remove more than one node at a time, it may try variants of the test case that are smaller and may be faster to verify compared to Pardis. To check this hypothesis, we conducted a case study on `gcc-64990` and recorded the running time of each oracle query during reduction. As shown in Tables 1 and 2, Pardis reduces this test case in 932 s with 2,632 queries, and Pardis Hybrid

has a total reduction time of 916 s (16 s shorter) while performing 3,148 oracle queries (516 more queries). Both techniques yield the same final test case.

Figure 7 depicts the distribution of oracle verification times in Pardis and Pardis Hybrid, showing that Pardis has more queries that take longer compared to Pardis Hybrid. The shorter queries in Pardis Hybrid directly decrease its overall reduction time making it reduce test cases with fewer queries compared to Perses and shorter queries compared to Pardis.

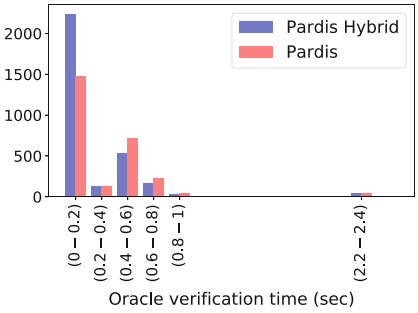


Fig. 7. Distribution of oracle verification time for Pardis and Pardis Hybrid.

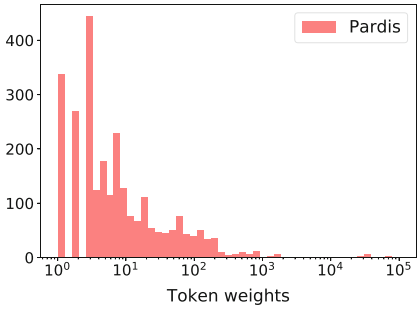


Fig. 8. Distribution of token weights of nodes visited during Pardis reduction.

(2) Distribution of Token Weights. The motivation behind proposing Pardis Hybrid as discussed in Sect. 3.1 was that if lists in a test case shrink after removing nodes with large unique token weights, applying DD on list elements with the same weight can be beneficial. In fact, the more of the remaining nodes that share token weights, the more beneficial using DD becomes since it provides the opportunity to remove those nodes in just one trial. This can avoid the possibly time-consuming process of visiting nodes one by one. To understand the distribution of token weights in practice, we perform Pardis (the one node at a time removal) on `gcc-64990` and record token weights of nodes visited during the removal process. Figure 8 shows the distribution with **5** as the median of token weights of nodes visited during the reduction. The small median motivates the use of Pardis Hybrid in practice since it indicates that half of the nodes have one of only five different token weights and can benefit from the grouped removals.

Syntactic vs Semantic Validity: Perses and Pardis discard *syntactically* invalid variants of the test case during reduction. However, there are also *semantically* invalid queries such as removing the declaration of a variable before removing its use. SGPR techniques cannot entirely avoid these queries since they guide the reduction process based on the syntax of the grammar. However, the priority order of Pardis can mitigate this problem. By prioritizing by token weight, it is more likely to visit and remove uses before spending effort on declarations. One reason for this is that a higher token weight tends to mean that there are more uses beneath that node. For instance, in Fig. 3, uses of variables `a`, `b` and

`c` are descendants of node ⑪ with nodes ⑧, ⑨ and ⑩ as their declarations. Pardis removes the uses by first removing ⑪ while Perses tries to remove the declarations first due to priority inversion. Hence, Pardis prunes nodes in one pass of the AST that Perses may require a fixed point mode to remove.

Threats to Validity: We evaluated Pardis on the same set of C test cases used in the evaluation of Perses. The implementation of Perses' grammar transformations and reduction is not publicly available, so we reimplemented Perses as described in its paper. Our implementation has been made available to provide a consistent platform for future work. However, the exact implementations, environmental settings and the scripts to check the property of interest can all impact the final results. For instance, the final sizes of the reduced test cases reported for the original Perses' implementation [6] are smaller than those using our reimplemented version of Perses. As discussed in Sect. 4.1, this may be because Perses' oracles allowed for undefined behavior, which can lead through smaller but invalid reduced test cases. To mitigate this problem, we made the oracles strictly prevent undefined behavior for both Pardis and Perses. Note that Pardis significantly outperforms both Perses' original implementation [6] and our reimplementation in terms of number of oracle queries.

While the techniques presented in Pardis are general in ability, our evaluation focuses on C in order to compare with Perses. Further investigation is required to claim that the performance benefits extend to other languages.

6 Related Work

The closest work to this paper is Perses [6]. Unlike Pardis, it suffers from priority inversion that adversely affects the reduction speed. Other generic test case reduction techniques are Delta Debugging (DD) [2], its $O(n)$ variant [13], and Berkeley Delta [18]. These face challenges when reducing hierarchical inputs. Several techniques focus on reducing hierarchically structured test cases [3,4,6,11,12,19,20]. Among these, only Perses is priority aware, in spite of its priority inversion. Indeed, most techniques process the input level by level. Like Pardis, Perses and Simp [20] are notable exceptions in that they can search across levels when deciding how to reduce. However, Simp is specific to SQL Queries. GTR [12] is notable in that it is trained when to apply different reduction operations. Finally, C-Reduce [5] is a tool for reducing C/C++ test cases that requires extensive domain-specific knowledge.

7 Conclusions

We have shown that the prior state of the art for test case reduction suffers from *priority inversion* and that this causes a significant increase in reduction time. We proposed priority aware reduction techniques, Pardis and Pardis Hybrid, that focus reduction effort where they can have the most impact. These techniques can speed reduction by 1.3× to 7.8× over the prior state of the art.

Acknowledgements. This research was partially supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. Clapp, L., Bastani, O., Anand, S., Aiken, A.: Minimizing GUI event traces. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, 13–18 November 2016, pp. 422–434 (2016)
2. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. **28**(2), 183–200 (2002)
3. Misherghi, G., Su, Z.: HDD: hierarchical delta debugging. In: 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, 20–28 May 2006, pp. 142–151 (2006)
4. Misherghi, G.S.: Hierarchical delta debugging. Master's thesis, University of California Davis (2007, Approved)
5. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China, 11–16 June 2012, pp. 335–346 (2012)
6. Sun, C., Li, Y., Zhang, Q., Gu, T., Su, Z.: Perses: syntax-guided program reduction. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 361–371 (2018)
7. Kernighan, B.W., Ritchie, D.: The C Programming Language, 2nd edn. Prentice-Hall, Upper Saddle River (1988)
8. Albert, J., Giammaressi, D., Wood, D.: Extended context-free grammars and normal form algorithms. In: Champarnaud, J.-M., Ziadi, D., Maurel, D. (eds.) WIA 1998. LNCS, vol. 1660, pp. 1–12. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48057-9_1
9. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, 4–8 June 2011, pp. 283–294 (2011)
10. IBM Support, Test Case Reduction Techniques. http://www-01.ibm.com/support/docview.wss?uid=swg21084174
11. Hodován, R., Kiss, Á.: Coarse hierarchical delta debugging. In: Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, 20–22 September 2017, pp. 194–203 (2017)
12. Herfert, S., Patra, J., Pradel, M.: Automatically reducing tree-structured test inputs. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, 30 October–03 November 2017, pp. 861–871 (2017)
13. Gharachorlu, G., Sumner, N.: Avoiding the familiar to speed up test case reduction. In: 2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, 16–20 July 2018, pp. 426–437 (2018)
14. Hathhorn, C., Ellison, C., Rosu, G.: Defining the undefinedness of C. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 336–345 (2015)
15. Parr, T.: The Definitive ANTLR 4 Reference, 2nd edn. Pragmatic Bookshelf, Raleigh (2013)

16. Wild, C., Seber, G.: Chance Encounters: A First Course in Data Analysis and Inference, 1st edn. Wiley, New York (1999)
17. Hodován, R., Kiss, Á.: Practical improvements to the minimizing delta debugging algorithm. In: Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, 24–26 July 2016, pp. 241–248 (2016)
18. McPeak, S., Wilkerson, D.S., Goldsmith, S.: Delta, July 2003. http://delta.stage.tigris.org/
19. Kiss, Á., Hodován, R., Gyimóthy, T.: HDDr: a recursive variant of the hierarchical delta debugging algorithm. In: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST 2018, pp. 16–22 (2018)
20. Bruno, N.: Minimizing database repros using language grammars. In: Proceedings of 13th International Conference on Extending Database Technology, EDBT 2010, Lausanne, Switzerland, 22–26 March 2010, pp. 382–393, 2010

Automatically Identifying Sufficient Object Builders from Module APIs

Pablo Ponzio[1,3(✉)], Valeria S. Bengolea[1], Mariano Politano[1,3], Nazareno Aguirre[1,3], and Marcelo F. Frias[2,3]

[1] Universidad Nacional de Río Cuarto, Río Cuarto, Argentina
{pponzio,vbengolea,mpolitano,naguirre}@dc.exa.unrc.edu.ar
[2] Instituto Tecnológico de Buenos Aires (ITBA), Buenos Aires, Argentina
mfrias@itba.edu.ar
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Buenos Aires, Argentina

Abstract. Various approaches to software analysis (e.g. test input generation, software model checking) require engineers to (manually) identify a subset of a module's methods in order to drive the analysis. Given a module to be analyzed, engineers typically select a subset of its methods to be considered as object builders to define a so-called *driver*, that will be used to automatically build objects for analysis, e.g., combining them non-deterministically, randomly, etc. This requires a careful inspection of the module and its API, since both the relative exhaustiveness of the analysis (leaving important methods out may systematically avoid generating different objects), as well as its efficiency (the different bounded combinations of methods grows exponentially as the number of methods increases), are affected by the selection.

We propose an approach for automatically selecting a set of builders from a module's API, based on an evolutionary algorithm that favors sets of methods whose combinations lead to producing larger sets of objects. The algorithm also takes into account other characteristics of these sets of methods, trying to prioritize the selection of methods with less and simpler parameters. As the implementation of this evolutionary mechanism requires in principle handling and comparing large sets of objects, and this grows very quickly both in terms of space and running times, we employ an *abstraction* of sets of objects, called field extensions, that involves using the field values of the objects in the set instead of the actual objects, and enables us to effectively implement our mechanism. An experimental assessment on a benchmark of stateful classes shows that our approach can automatically identify sets of builders that are *sufficient* (can be used to create any instance of the module) and *minimal* (do not contain superfluous methods), in a reasonable time.

1 Introduction

As software is becoming more ubiquitous thanks to the rapid advances in technology, guaranteeing the functional correctness of software is more crucial than

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 427–444, 2019.
https://doi.org/10.1007/978-3-030-16722-6_25

ever. Thus, a research area of growing importance is that of automated software analysis, whose goal is to assist engineers, through the provision of tools for automated analysis, in finding deficiencies both in software and software related models. Automated test generation [1,11,13,17,24,25,28,29,32], software model checking [9,34,35], and static analyses [6,16], among many others, are prominent approaches in this line of research.

While these techniques involve in many cases fully automated analyses, their application often requires some effort from the engineers. Software model checkers rely on the definition of *drivers*, programs that allow one to build inputs for the code under analysis. Similarly, in parameterized-unit testing approaches [33] a mechanism for building inputs is mandatory. Some symbolic execution based tools require the so-called *"object factories"* to build tests cases involving inputs with non-primitive types [32]. Automated test generation techniques based on a module's API can be used for building inputs for non-primitive types [11,24], thus automating the above-mentioned input-generation issues. But they usually present difficulties in generating a good set of diverse inputs for stateful, complex structures. This is even more difficult for structures with rich APIs [26]. Many authors have addressed this problem by defining different approaches for guiding test generation, to create more diverse sets of inputs [7,26].

In this paper, we take a different approach to address the problem of generating better inputs for stateful modules. We observe that the selection of routines from a module API, to feed an input generation tool so as to build input structures for program analysis (drivers for model checking, input structures for parameterized unit tests, etc.), has a crucial impact on the analysis. We call *builders* a set of routines B, drawn from a module's M API, that can be employed to create input structures in an automated program analysis for M (e.g. a driver for model checking). Clearly, the higher the number of different structures that can be created with B, the better the chances to find bugs in M. As the number of instances of a software module is potentially infinite, and the program analyses we target are also limited in the number of structures they can employ, we limit ourselves to a bounded-exhaustive set of structures for M [4] (e.g. all the instances of a linked list with up to k nodes). We denote this set by $BE(M,k)$. We say that a builders are *sufficient* if they can combined to build all the instances in $BE(M,k)$. Thus, sufficient builders are the best possible choice for bug finding (in a bounded setting). Notice that B can contain superfluous routines. A superfluous routine s is such that $BE(M,k)$ can be built using routines in $B - \{s\}$ (the simplest example being routines that never change the state of their parameters). These routines provide no benefits in terms of bug finding capabilities of the analysis. We call *minimal* a set of builders with no superfluous routines. Minimality is important because providing an analysis tool with superfluous routines often negatively impacts its efficiency (the number of ways k routines can be combined usually increases exponentially with k).

Manually selecting sufficient and minimal builders is not an easy task: it requires a thorough analysis of the available routines and a deep understanding of the program semantics. This is especially hard for programs with rich APIs,

where there are many routines and a lot of redundancy in the API (see Sect. 2). We propose an automated approach for identifying such a sufficient and minimal set of builders, based on an evolutionary algorithm that searches for a minimal set of routines that is capable of generating the maximum number of different (bounded) objects (i.e., $BE(M, k)$). Moreover, our evolutionary approach also takes into account other characteristics of the builders, such as the number and complexity of their parameters, so that "simpler" routines are favored in the search. The goal is to choose builders that can be more easily and more efficiently used by the subsequent program analyses.

The fitness value for a set of routines R is based on the number of bounded structures that can be generated using combinations of these routines. To compute the fitness we use a modified version of a random test case generation tool (Randoop [24]) to generate as many bounded structures as possible from R, allowing at most k of objects of each type in the structures (a parameter to our algorithm). As sets of objects are very expensive to maintain and manipulate, both in terms of space and running time, we employ an efficient abstraction of a set of objects, called *field extensions*, defined as the set of field values appearing in any of the objects in the set [25]. Thus, instead of counting the number of different objects achieved by a candidate, the fitness function will compute the field extensions as objects are generated, and return the number of field values in the extensions. Intuitively, a higher number of field values in the field extensions means that the builders can be used to construct a more diverse set of objects, and therefore they should be preferred over other sets of builders.

We assess our approach experimentally on a benchmark of stateful Java classes drawn from the literature. The results show that in our case studies our approach identifies sets of routines that are sufficient and minimal, in a reasonable time. We also assess the impact of our approach in an automated analysis, namely, in the generation of test cases for parameterized tests. We compare how the random test case generation tool Randoop behaves when fed with the full module API, against providing the tool with only the builders identified by our approach. The results indicate that in the latter case Randoop generated more (and larger) objects, within a fixed time budget.

2 Motivating Example

In this section, we motivate our approach by means of a running example. The Apache NodeCachingLinkedList (NCL for short) [36] consists of a main circular doubly linked list, that holds the elements of the collection, and a secondary singly linked list that acts as a cache for nodes that have been removed from the main list. Nodes stored in the cache can be reused, and added again to the main list when inserting elements in the main list. Thanks to its cache, in applications where insertions and removals from the list are very frequent, NCL can significantly reduce the overhead needed for memory allocation and garbage collection of nodes. As an illustration, Fig. 1 shows the three NCL instances that can be built with exactly two nodes.

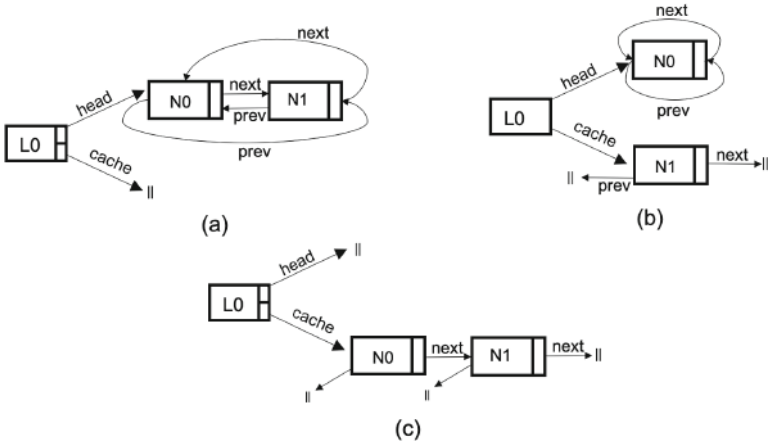


Fig. 1. Three NodeCachingLinkedList instances with exactly two nodes

Table 1. Apache's NodeCachingLinkedList API

No.	Return type	Method name	Obs?
0		NCL()	no
1		NCL(int)	no
2		NCL(Collection)	no
3	boolean	add(Object)	no
4	void	add(int,Object)	no
5	boolean	addAll(Collection)	no
6	boolean	addAll(int,Collection)	no
7	boolean	addFirst(Object)	no
8	boolean	addLast(Object)	no
9	void	clear()	no
10	boolean	contains(Object)	yes
11	boolean	containsAll(Collection)	yes
12	boolean	equals(Object)	yes
13	Object	get(int)	yes
14	Object	getFirst()	yes
15	Object	getLast()	yes
16	int	indexOf(Object)	yes

No.	Return type	Method name	Obs?
17	boolean	isEmpty()	yes
18	Iterator	iterator()	no
19	int	lastIndexOf(Object)	yes
20	ListIterator	listIterator()	no
21	ListIterator	listIterator(int)	no
22	Object	remove(int)	no
23	boolean	remove(Object)	no
24	boolean	removeAll(Collection)	no
25	Object	removeFirst()	no
26	Object	removeLast()	no
27	boolean	retainAll(Collection)	no
28	Object	set(int,Object)	no
29	int	size()	yes
30	List	subList(int,int)	no
31	Object[]	toArray()	yes
32	Object[]	toArray(Object[])	yes
33	String	toString()	yes

NCL has a very rich API, as shown in Table 1. However, for building any feasible NCL object only a few methods from the API suffice. For example, combinations of the methods in Fig. 1.1, when instantiated with appropriate parameters, can be used to build any desired (finite) NCL object. Thus, the methods therein are an example of a sufficient set of builders. Notice that, after using the constructor, the main list of NCL can be populated just by using the `addFirst` method. However, if we want to generate instances where the cache is not empty, we can do so through the `removeFirst` method, as the sufficient set of builders suggests. For most automated analyses, we would like to consider as varying scenarios (inputs) as possible, hence the motivation to build sufficient sets of builders. Furthermore, the builders in Fig. 1.1 are also minimal, since the lack of any one of them would imply that some NCL's objects cannot be constructed anymore with the routines.

```
(0)  NodeCachingLinkedList()
(7)  addFirst(Object)
(25) removeFirst()
```

Figure 1.1. A sufficient set of builders for NCL

```
(3) add(Object)
(4) add(int,Object)
(7) addFirst(Object)
(8) addLast(Object)
```

Figure 1.2. Add variants that can be used to populate NCL's main list

Notice that there can be many sets of sufficient and minimal builders. For example, we get sufficient and minimal builders by replacing `addFirst` in Fig. 1.1 with any of the other add variants shown in Fig. 1.2, as for any way of filling up NCL's main list with `addFirst` there exists a different way to build the same object using another add variant (perhaps invoked with different parameters and changing the execution order).

We also observe that the simpler the parameters of a routine, the easier to use the routine is for generating inputs in the context of a program analysis. For instance, among the alternative add routines for NCL (Fig. 1.2), `add(int,Object)` receives more parameters than the other three methods, therefore it is harder to generate parameters for it when generating inputs. This makes the other three alternatives preferred over it. Thus, our approach takes into account the number of parameters and their complexities for selecting the best possible builders.

Many methods in Table 1 are marked as observers (column Obs?), meaning that they do not modify the objects they operate on, nor they are useful for creating non-primitive objects. Hence, observers are always superfluous, and should never be included in a set of minimal builders. Our approach tries to recognize them beforehand, and discards them from the search to significantly reduce the search space.

To conclude this section we remark that, when fed with the whole NCL's API, our approach automatically identified the sufficient and minimal set of builders for NCL shown in Fig. 1.1.

3 Background

3.1 Field Extensions

The idea behind field extensions [25] is to define a representation for a set of objects that is smaller in size and easier to manipulate algorithmically. This representation implies some loss of information, but for certain applications (like the one in this paper) they are precise enough to be useful in practice [1,12,25, 26,29].

$head = (L_0, null), (L_0, N_0)$
$cache = (L_0, null), (L_0, N_1), (L_0, N_0)$
$next = (N_0, N_1), (N_1, N_0), (N_0, N_0), (N_1, null)$
$prev = (N_0, N_1), (N_0, N_0), (N_1, null), (N_0, null)$

Figure 1.3. Field extensions for the set of instances in Fig. 1

Given a set S of objects, its field extensions representation consist of a set of pairs for each field f, such that (obj,val) belongs to the field extensions of f if obj.f = val (i.e., the value of f for obj equals to val), for some object obj in S. As an example, consider the instances displayed in Fig. 1. Its corresponding field extensions are shown in Fig. 1.3. We omit the values stored in the nodes for the sake of clarity. Notice that structure (a) in Fig. 1 can be built using only add methods, whereas for (b) and (c) we have to also employ some kind of remove operation, to move nodes from the main list to the cache. Notice that values (L_0, N_0) and (L_0, N_1) for the cache field only appear in the field extensions when the structures have nodes in the cache, like (b) and (c). In addition, prev fields of nodes in the cache are always *null*, but prev fields can never be *null* in the main list (due to its circularity). This means that field extensions for structures that have non-empty caches have the potential of having a larger number of values than those for structures with no caches.

It is important to canonicalize structures before computing field extensions [12]. Canonicalization involves assigning unique identifiers $N_0, N_1, ...$ to each of its nodes during a traversal of the structure (we employ a breadth first traversal), starting at the root. Nodes visited first receive smaller identifiers than those visited afterwards during the traversal. Fields must be visited in a fixed order. Note that structures in Fig. 1 are all in canonical breadth-first form.

3.2 Random Test Case Generation

Random test generation consists of randomly producing inputs in order to test software [8,21,24]. Random input generation is straightforward when considering basic (numeric) data types, but producing inputs of other more complex types, in particular instances of *stateful* classes, is less obvious and calls for a more complex mechanism, other than just using random number generators. One such mechanism, that has been implemented by various tools for random test generation for object-oriented code, is based on randomly combining method sequences, that produce inputs of different types [8,21,24]. The process associated with the Randoop tool [24] that we use here, works essentially as follows. For every datatype, a set of sequences that produce inputs of such datatype, is maintained. To start with, for basic data types, a set of initial values is considered, and for class types, only null is considered at first (these can be considered test sequences of size one). The procedure to build a new test sequence starts by randomly selecting a method m, among all methods in the software under test. For example, one could randomly choose one of the methods for the NCL's API (Table 1), say add(Object). To actually build the test sequence, values for

each of the parameters of the method m, of the corresponding types, have to be provided. These are obtained by randomly selecting test sequences, from the sets of sequences of the corresponding types, and sequentially composing them, with method m as a last statement. As an example, say that a sequence containing only the constructor of NCL is randomly selected, from the available sequences for the NCL type, and for the parameter of add, an Integer with value 0 is randomly chosen. Combining all these sequences together results in:

```
NodeCachingLinkedList l = NodeCachingLinkedList();
l.add(new Integer(0));
```

This new sequence can now be stored for later use a as parameter for other methods that operate on NCL objects.

This process is repeated until either a time budget is exhausted, or the desired number of tests (set by the user) is generated. Randoop uses guidance from the execution of tests to avoid generating illegal tests. We refer the interested reader to the article introducing Randoop [24], for further details.

An important issue to remark here is that the execution of each test sequence generated by Randoop produces a number of objects for the given type (NCL in the example). We exploit this characteristic of Randoop to compute the fitness function for a set of methods, although instead of storing actual objects we will maintain field extensions, as we explain in more detail in Sect. 4.

4 An Evolutionary Algorithm for Identifying Sufficient Object Builders

As mentioned before, to find a sufficient set of builders from a program API we design a genetic algorithm, that we describe below. Genetic algorithms [14] are non-exhaustive guided search algorithms, based on a hill climbing strategy [30]. The search space is composed of a generally very large set of individuals (the candidates), and the search objective is to find an individual with sought-for features. As opposed to classic search algorithms, genetic algorithms maintain a set of individuals, called the population, and search progresses by iteratively selecting a number of individuals in the population, using these for evolution (building new individuals out of these), and leaving out some individuals of the whole set (the "old" ones and the "new" ones). Selection of individuals for population evolution, as well as individuals' removal, are guided by a fitness function, the heuristic function used to guide the search. This function applies to individuals, and its result is generalizable to the population too (e.g., the fitness of the population may be taken as the fitness of its "fittest" individual). This function captures the features sought for in the search, and thus can be used as a halting criterion (e.g., algorithm stops after finding an individual with fitness above a certain threshold). Finally, individuals are often called chromosomes, and represented as vectors of genes that capture their characteristics. This idea is strongly related to how new individuals are constructed: by representing candidates as

vectors of independent characteristics, one can build new candidates by combining part of the characteristics of an individual with part of the characteristics of another, or by arbitrarily changing a characteristic of a given individual. These two forms of evolution are called crossover and mutation, respectively, and are the traditional mechanism to build new candidates out of existing ones in genetic algorithms. For further details, we refer the reader to [22].

4.1 Chromosome Representation

In the context of our problem, candidate solutions represent sets of methods from the API of the module being analyzed. We then employ vectors of boolean values as chromosome representation. Let n be the number of methods in the API; the chromosomes in our algorithm will be vectors of size n. For any vector, the i-th position is true if and only if the chromosome contains the i-th method of the API. For example, there are 34 methods in the NCL's API (Table 1), and we enumerated them from 0 to 33. The sufficient set of builders in Fig. 1.1 is characterized by the vector with positions 0, 7 and 25 set to true, and the remaining positions set to false. In this case, the whole search space consists of the 2^{34} possible chromosomes.

4.2 Fitness Function

Given a chromosome representing a set of methods M, our fitness function computes an approximation of the number of bounded objects that can be built using combinations of methods in M. Chromosomes with higher fitness values are estimated to build more objects than those that have smaller fitness values.

Ideally, we would like to explore all the feasible objects within a small bound k, that can be built using the methods of the current chromosome, i.e., $BE(M, k)$. In other words, we need a bounded exhaustive generator for the set of methods. The bound k represents the maximum number of objects that can be created for each class (in Fig. 1, the number of nodes in the NCL objects are bounded by $k = 2$), and the maximum number of primitive values available (for example, integers from 0 to $k - 1$). For this purpose, we developed a prototype modifying the Randoop tool, discussed briefly in Sect. 3.2. First, we altered Randoop to work with a fixed set of primitive values (integers from 0 to $k - 1$). (Normally, Randoop would save primitive values that are returned by the execution of tests, and reuse these values in future tests.) Second, we make Randoop drop sequences of methods that create objects with more than k objects (of any type), to stop it from building objects larger than needed. To achieve this, we canonicalize the objects generated by the execution of each sequence, and we discard the sequence if some object has an index equal or larger than k. Third, we extend Randoop with "global" field extensions, and when the execution of a sequence terminates all the field values of the objects generated by the sequence are added to the field extensions. For example, if Randoop had generated the objects in Fig. 1, then the global field extensions would have the values shown in Fig. 1.3. Our goal is that, given a bound k, when our modified version of

```
(0)  NodeCachingLinkedList()
(7)  addFirst(Object)
(8)  addLast(Object)
(25) removeFirst()
```

Figure 1.4. A set of sufficient but not minimal builders for NCL

```
(0)  NodeCachingLinkedList()
(4)  add(int,Object)
(23) remove(Object)
```

Figure 1.5. Sufficient and minimal builders for NCL with more complex parameters than the ones in Fig. 1.1

Randoop terminates the global field extensions contain all the field values of the bounded exhaustive set of structures with up to k nodes, $BE(M, k)$. The result of the fitness function for the chromosome is the number of field values in the global extensions computed by the tool.

Our rationale for using bounded sets of objects is akin to the small scope hypothesis for bug finding [2]: if one set of methods cannot be used to build small objects that allow to differentiate it from another set of methods, then it is unlikely that these two sets can be distinguished with larger objects. This hypothesis held during our empirical evaluation across all our case studies.

We found that, besides being affected by chance, our tool rarely misses building objects that should add relevant values to the global extensions, when small values for k are employed.

Choosing Better Sets of Builders. In this section, we propose two ways to improve our evolutionary algorithm by tailoring the fitness function to obtain better sets of builders. This is strongly motivated by the way builders are used to build inputs in program analysis. On the one hand, if we have two sufficient set of builders, the set with the smaller number of methods should always be preferred. In this context, there is no reason to include superfluous methods in builders. For example, the builders in Fig. 1.4 can be used to create the same NCL objects as the builders in Fig. 1.1 of Sect. 2 (both sets are sufficient), but they are not minimal since `addLast` is superfluous.

On the other hand, builders with more parameters, or more complex ones, are more taxing on program analysis, as they require more effort to be adequately instantiated. Thus, we define a simple criterion of parameter complexity and adapt our fitness to favor builders with simpler parameters over the more complex ones. For example, both sets of builders in Figs. 1.1 and 1.5 are sufficient and minimal (with 3 routines each), but builders in Fig. 1.5 have more parameters that need to be instantiated. Comparing Figs. 1.1 and 1.5 we can observe that `addFirst` has been replaced by `add`, which has an additional integer parameter, and that `removeFirst` was interchanged with `remove`, which possesses a

non-primitive parameter of type Object. Following the criteria explained above, we would like our algorithm to choose the set in Fig. 1.1 over that of Fig. 1.5.

Incorporating these ideas, the fitness function of our approach is defined by:

$$f(M) = \#fieldExt(M) + \left(\frac{w_1 * \left(1 - \frac{\#M}{\#MT}\right) + w_2 * \left(1 - \frac{(\#PP(M) + w_3 * RP(M))}{(\#PP(MT) + w_3 * RP(MT))}\right)}{w_1 + w_2} \right)$$

For a chromosome representing a set M of methods, drawn from the whole set of available methods of the API, MT, the most important part of the fitness for M, is the number of values in the field extensions, $\#fieldExt(M)$, that can be generated using our custom Randoop tool as explained in the previous section. The summand on the right implements the ideas presented in this section. It returns a real value in the interval [0, 1] that is useful to break ties for sets of methods that generate field extensions with the same number of values. In the dividend, the first summand penalizes sets with larger numbers of methods, by computing the quotient of the number of methods in M to the number of methods in MT, and subtracting the result to 1. Constant w_1 ($w_1 \geq 1$) allows us to increase/decrease the weight of this summand with respect to the other summand. The second summand in the dividend penalizes sets of methods with more complex parameters. Similarly to w_1, constant w_2 ($w_2 \geq 1$) serves the purpose of increasing/decreasing the weight of this factor in the sum. Notice that we sum up the parameters differently depending on their types: each primitive parameter adds 1 ($PP(M)$ is the number of primitive parameters in the methods of M), and each reference parameter adds a constant w_3 ($w_3 \geq 1$, $RP(M)$ is the number of reference-typed parameters in the methods of M), which allows us to increase the weight of reference parameters with respect to primitive ones. Intuitively, the whole right-hand summand computes the ratio between the number of parameters of M (with added weight for reference parameters) to the number of (weighted) parameters for MT. The result is then subtracted from 1. Finally, we divide by $w_1 + w_2$ to obtain the desired number in the interval [0, 1].

In our experimental assessment we set $w_1 = 2, w_2 = 1, w_3 = 2$. These values were good enough for our approach to produce sufficient and minimal sets of builders in all our case studies.

It is important to remark that the presented criteria for choosing better builders is based on the kind of program analyses we target (generation of tests cases for parameterized tests, software model checking). New criteria can be defined with other goals in mind, and our approach can be adapted to support them by modifying the fitness function as we did in this section.

4.3 Overall Structure of the Genetic Algorithm

The previously described elements are the constituting parts of the genetic algorithm implementing our approach. A pseudocode of the genetic algorithm is shown in Algorithm 1. Notice that Algorithm 1 follows the general structure of

Algorithm 1. Genetic Algorithm implementing our approach

```
1: pop ← chromosomes with exactly one true gene
2: for i = 1...numEvo do
3:    pop ← keep the popSize fittest chromosomes from pop
4:    for j = 1...cRate * popSize do
5:       c1, c2 ← select two random chromosomes from pop
6:       new ← single point crossover c1, c2
7:       add new to pop
8:    end for
9:    for c ∈ pop do
10:      new ← mutate each gene of c with probability mRate
11:      if new ≠ c then
12:         add new to pop
13:      end if
14:   end for
15: end for
16: result ← fittest chromosome of pop
```

a genetic algorithm. The initial population is generated by producing all the feasible chromosomes with only one available method (vectors with false in all positions except one, set to true) (line 3). Then, it starts to iteratively evolve the population (lines 4–15). At the beginning of each evolution iteration, the algorithm discards some individuals to control population size, by keeping the *popSize* fittest individuals of the current population and discarding the rest (line 5). Then, the algorithm performs single-point crossover on randomly selected individuals (lines 6–10). Crossover is applied a number of times that is proportional to the population size *popSize*, determined by the product of *popSize* and the crossover rate parameter *cRate* ($0 \leq cRate \leq 1$). Then, the algorithm mutates individuals (lines 11–15) by changing the value of each of its genes with probability *mRate* ($0 \leq mRate \leq 1$). Any newly created individual by the crossover and mutation operations are added to the population.

The algorithm stops after *numEvo* evolutions, with *numEvo* a parameter of the algorithm. Notice that, we don't have a target value for our fitness, since an untried set of methods might produce a larger number of field extensions than the algorithm has currently seen. Again, there is a compromise to be made for choosing a good value for *numEvo*: a larger number increases the precision of the algorithm but increases its running time, whereas a smaller number makes it run faster but it might not result in the best set of builders.

As usual, we found a number for the parameters of our algorithm that seems to work well in practice. In our experimental evaluation, we set $numEvo = 20, popSize = 30, cRate = 0.35, mRate = 0.08$ (the last two are the default for the JGap library).

Most of Algorithm 1 is a default evolutionary implementation of the JGap Java library [37]. Notice that, if we take away the complexity of the fitness function, our evolutionary algorithm is rather standard, so it is not surprising that

an existing implementation works well for our purposes. Of course, improvements to the evolutionary algorithm, and fine tuning for its parameters (e.g., crossover/mutation rate) might yield faster execution times.

We also implemented a simple multi-threaded version of our approach, that helps improving its performance. Basically, at each iteration we make t copies of the current population, where t is the number of available threads, and evolve each of the population replicas independently of the others. After all the threads have finished, we keep the $100/t$ fittest individuals of the population evolved by each thread, and use them to build the population for the next iteration of the algorithm.

4.4 Reducing the Search Space by Observers Classification

We say a routine is an observer if it never modifies the parameters it takes, and never generates a non-primitive value as a result of its execution. Column Obs? in Table 1 (Sect. 2) indicates whether each NCL method is an observer or not. Clearly, an observer cannot be used to modify nor build new objects, and therefore can never belong to a minimal set of builders. Hence, if we can classify them correctly beforehand, we can remove the observers from the search to significantly reduce the search space, without losing precision. For example, in the NCL API (Table 1) there are 13 observers out of 34 methods, so by removing observers we prune more than one third of the search space.

To detect observers we run another customized Randoop version before our evolutionary algorithm. This time, we check for each method whether it modifies its inputs at each test sequence generated by Randoop involving the method, by canonicalizing the objects before and after execution of the method, and checking if the field values of the objects change after execution. If this is the case, the method is marked as a builder (not an observer). For return values, if in any test sequence generated by Randoop the method returns a non-primitive value, then we mark it as a builder as well. We run this custom Randoop until it generates a large number of scenarios for each method. Ten to twenty seconds was enough for our case studies. At the end of the Randoop execution, methods not marked as builders are considered observers and discarded before invoking the evolutionary algorithm.

Other approaches exist for the detection of pure methods [15,31] (similar to our observers). Note that our evolutionary algorithm is not dependent on the method classification algorithm, so any of them could be useful for our purposes.

5 Experimental Results

In this section, we experimentally assess our approach. The evaluation is based on a benchmark of data structure implementations, including: `NCL` from Apache Collections [36]; `BinaryTree`, `BinomialHeap`, `FibonacciHeap`, `RedBlackTree` taken from [35]; `UnionFind`, an implementation of disjoint sets taken from JGrapht [38]. We also evaluate our technique on components of real software projects

such as `Lits` from the implementation of Sat4j [3], taken from [20], which consists of a variable store that monitors when a guess was last made about a value of a variable, and whether listeners are watching the state of that variable; and `Scheduler`, an implementation of a process scheduler taken from [10]. All the experiments were run on 3.4 GHz quad-core Intel Core i7-6700 machines with 8 GB of RAM, running GNU/Linux.

The evaluation consists of two parts. First, we ran our approach (Algorithm 1) on the whole module APIs of the aforementioned classes, to compute sets of builders for each case study. The goal is to assess how good are the builders identified, and the time it takes our approach to compute them. For each case study we ran our approach 5 times. The results are shown in Table 2, including the number of routines in the whole API (#API), a sample of identified builders (some methods might be interchanged in different runs, e.g., `addFirst` and `addLast` in NCL), and the average running time (in seconds) of the 5 runs. We manually inspected the results, and found that the automatically identified sets of builders were in all cases sufficient (all the feasible objects for the structure can be constructed using the builders) and minimal (do not contain superfluous methods). The approach is reasonably efficient, taking about 30 min in the worst case.

The second part of the evaluation regards how helpful are the identified builders in the context of a program analysis, namely, the automated generation of test cases. These objects might be used, for example, as inputs in parameterized unit tests. For the case studies that provide mechanisms to measure the size of objects and to compare objects by equality (i.e., the *size* and *equals* methods of data structures), we generated tests with Randoop using all the methods available in the API (API), and then we generated tests with Randoop using only the builder methods (BLD) identified by our approach in the previous experiment (Table 2). We then compare the number of different objects (No. of Objs.), and the size of the largest object (Max Obj. Size) created by the tests generated from the API, against the tests generated using methods from BLD only. We set three different test generation budgets: 60, 120 and 180 seconds (Budget). The results are summarized in Table 3. In addition, we consider another approach, API+, that involves the generation of tests using the API for a budget that encompasses the test generation budget (Budget) plus the time it takes our approach to identify builders for the corresponding case study. The results show that in the same test budget BLD generates in average 1280% more objects than API. Furthermore, when builders identification time is added to the test generation budget for API (API+), BLD can generate 568% more objects in average (w.r.t API+). In all cases, BLD also generates significantly larger objects than API and API+. In view of these results, it is clear that automated builders identification pays off for the automated generation of structures for stateful classes.

The experiments can be reproduced by following the instructions in the paper website [27]. Furthermore, in the site we experimentally show that the builders identified by our approach can be employed to build efficient drivers for software model checking. We don't show these results here due to space constraints.

Table 2. Builders computation results

	Sample Builders	Time
NCL	NCLinkedList(int)	
	addFirst(Object)	1744
#API: 34	removeFirst()	
UFind	UnionFind()	
	addElement(int)	215
#API: 9	union(int,int)	
FHeap	FibonacciHeap()	
	insert(int)	72
#API: 7	removeMin()	
RBT	TreeMap()	
#API: 8	put(int)	73
BTree	BinTree()	
#API: 7	add(int)	73
BHeap	BinomialHeap()	
#API: 10	insert(int)	121
Lits	Lits()	
#API: 26	getFromPool(int)	
	forgets(int)	1229
	setLevel(int,int)	
	setReason(int)	
Sched.	Schedule()	
#API: 10	addProcess(int)	
	blockProcess()	377
	quantumExpire()	

Table 3. Assessment of using the identified builders (BLD) vs the whole API (API) in test case generation. API+ involves test case generation with the whole API, with budget = (Budget + builders computation time)

	Budget	Max Obj. Size			No. of Objs.		
		API	BLD	API+	API	BLD	API+
NCL	60	8	16	11	1442	42021	13119
#API: 34	120	8	18	11	2423	69017	13247
#BLD: 3	180	9	18	11	3166	91647	13505
UFind	60	8	13	9	3388	34250	8351
#API: 9	120	9	13	9	5180	56418	8574
#BLD: 3	180	9	13	9	6695	74425	9387
FHeap	60	11	15	12	6989	32639	11499
#API: 7	120	12	17	13	11447	54264	17202
#BLD: 3	180	12	17	13	15344	72413	20775
RBT	60	8	15	8	1812	23034	3041
#API: 8	120	8	15	8	2678	35635	3698
#BLD: 2	180	8	15	8	3358	44807	3940
BTree	60	8	15	8	3600	24908	6019
#API: 7	120	8	15	8	5471	39239	7387
#BLD: 2	180	8	15	9	6975	50671	9247
BHeap	60	9	26	10	3874	65915	8076
#API: 10	120	10	29	10	5970	111402	9708
#BLD: 2	180	10	29	11	7638	147260	10606

6 Related Work

As mentioned throughout the paper, the problem of identifying sufficient builders is recurrent in various program analyses, including but not limited to software model checking and test generation. In works like [18,23], in the context of software model checking, and [5,24,32,33], in the context of automated test generation, and just to cite a few, the problem of identifying part of an API and provide it for analysis is present. Typically the problem is dealt with manually.

The use of search-based techniques to solve challenging software engineering problems is an increasingly popular strategy, which has been applied successfully to a number of problems, including test input generation [11], program repair [19], and many others. As far as we are aware of, this is a novel application of evolutionary computation in software engineering. An approach that tackles a related, but different, problem, is that associated with the SUSHI tool [5]. The aim with SUSHI is to feed a genetic algorithm with a *path condition*, produced by a symbolic execution engine, so that an input satisfying the provided path condition can be reproduced using a module's API. This approach assumes that the API (or the subset of relevant methods) is provided, as opposed to our work, that precisely tackles the provision of the restricted API.

Our technique requires a mechanism for identifying *observers*, which we have solved within the work in the paper, resorting to random test generation,

and instrumentation for state monitoring. Approaches to the identification of observers, or more precisely *pure methods*, exist in the literature [15,31]. Regarding these lines of work, notice that the focus of our evolutionary algorithm is not the identification of observers, but the construction of minimal and sufficient set of builders. Moreover, our approach is in fact independent of the mechanism used to identify observers/pure methods, and thus could be combined with the works just cited (i.e., replacing our random testing based approach by an alternative one).

7 Conclusions

In this work, we presented an evolutionary algorithm for automatically detecting sets of builders from a module's API. We assessed our algorithm over several case studies from the literature, and found that it is capable of precisely identifying sets of builders that are sufficient and minimal, within reasonable running times. To the best of our knowledge, this is the first work that addresses this problem, which is typically dealt with manually.

We also showed preliminary results indicating that our approach can be exploited by test case generation tools to yield larger and more diverse objects. Other techniques, like software model checking, can benefit as well by using the identified set of builders to automatically construct efficient drivers. More experimentation needs to be done, but given the results in this paper our approach looks very promising.

One of the biggest challenges of this work was the construction of a tool to allow us to generate all the bounded structures, for a given maximum number k of objects, from the methods of the program API. The proposed solution worked well enough for our case studies, but avoiding randomness in the process would be desirable. Using bounded exhaustive generation tools rather than random generation would better fit our purposes [4], but unfortunately none of the tools for bounded exhaustive test generation produce inputs from a module's API. We believe that a promising research direction, that we plan to further explore in future work, is to adapt our presented approach for bounded exhaustive test generation.

Some aspects of our genetic algorithm can be further improved. For instance, a more powerful classification for argument types, in the prioritization of methods according to their complexities, can be defined. Moreover, one may also incorporate other dimensions, such as *code complexity*, to favor simpler methods. We will explore this direction as future work. Also, our genetic algorithm implementation is, for most parts, a default evolutionary implementation of the JGap Java library [37]. Of course, improvements to the evolutionary algorithm, and fine tuning for its parameters (e.g., crossover/mutation rate) might yield faster execution times, so we plan to investigate this further in future work.

References

1. Abad, P., et al.: Improving test generation under rich contracts by tight bounds and incremental SAT solving. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, 18–22 March 2013, pp. 21–30 (2013)
2. Andoni, A., Daniliuc, D., Khurshid, S.: Evaluating the small scope hypothesis. Technical report, MIT Laboratory for Computer Science (2003)
3. Berre, D.L., Parrain, A.: The Sat4j library, release 2.2, system description. J. Satisf. Boolean Model. Comput. **7**, 59–64 (2010)
4. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2002, pp. 123–133. ACM, New York (2002)
5. Braione, P., Denaro, G., Mattavelli, A., Pezzè, M.: SUSHI: a test generator for programs with complex structured inputs. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 21–24. ACM (2018)
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011)
7. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: ARTOO: adaptive random testing for object-oriented software. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 71–80. ACM, New York (2008)
8. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pp. 268–279. ACM, New York(2000)
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
10. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empirical Softw. Eng. **10**(4), 405–435 (2005)
11. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011, pp. 416–419. ACM, New York (2011)
12. Galeotti, J.P., Rosner, N., López Pombo, C.G., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA 2010, pp. 25–36. ACM, New York (2010)
13. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, pp. 225–234. ACM, New York (2010)
14. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
15. Huang, W., Milanova, A., Dietl, W., Ernst, M.D.: Reim & ReImInfer: checking and inference of reference immutability and method purity. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 879–896. ACM, New York (2012)

16. Itzhaky, S., Bjørner, N., Reps, T., Sagiv, M., Thakur, A.: Property-directed shape analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 35–51. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_3
17. Khalek, S.A., Yang, G., Zhang, L., Marinov, D., Khurshid, S.: TestEra: a tool for testing Java programs using alloy specifications. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, pp. 608–611. IEEE Computer Society, Washington, DC (2011)
18. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_40
19. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)
20. Loncaric, C., Ernst, M.D., Torlak, E.: Generalized data structure synthesis. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 958–968 (2018)
21. Meyer, B., Ciupa, I., Leitner, A., Liu, L.L.: Automatic testing of object-oriented software. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 114–129. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69507-3_9
22. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs, 3rd edn. Springer, Heidelberg (1996). https://doi.org/10.1007/978-3-662-03315-9
23. Nori, A.V., Rajamani, S.K., Tetali, S.D., Thakur, A.V.: The Yogi project: software property checking via static analysis and testing. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_17
24. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for Java. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, OOPSLA 2007, pp. 815–816. ACM, New York (2007)
25. Ponzio, P., Aguirre, N., Frias, M.F., Visser, W.: Field-exhaustive testing. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 908–919. ACM, New York (2016)
26. Ponzio, P., Bengolea, V., Brida, S.G., Scilingo, G., Aguirre, N., Frias, M.: On the effect of object redundancy elimination in randomly testing collection classes. In: Proceedings of the 11th International Workshop on Search-Based Software Testing, SBST 2018, pp. 67–70. ACM, New York (2018)
27. Ponzio, P., Bengolea, V.S., Politano, M., Aguirre, N., Frias, M.F.: Replication package of the article: automatically identifying sufficient object builders from module APIs. https://sites.google.com/view/objectbuildergeneration/
28. Păsăreanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of Java bytecode. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE 2010, pp. 179–180. ACM, New York (2010)
29. Rosner, N., Geldenhuys, J., Aguirre, N., Visser, W., Frias, M.F.: BLISS: improved symbolic execution by bounded lazy initialization with SAT support. IEEE Trans. Softw. Eng. **41**(7), 639–660 (2015)
30. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall Press, Upper Saddle River (2009)
31. Sălcianu, A., Rinard, M.: Purity and side effect analysis for Java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_14

32. Tillmann, N., De Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10
33. Tillmann, N., de Halleux, J., Xie, T.: Parameterized unit testing: theory and practice. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, pp. 483–484. ACM, New York (2010)
34. Visser, W., Mehlitz, P.: Model checking programs with Java PathFinder. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, p. 27. Springer, Heidelberg (2005). https://doi.org/10.1007/11537328_5
35. Visser, W., Păsăreanu, C.S., Pelánek, R.: Test input generation for Java containers using state matching. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis, ISSTA 2006, pp. 37–48. ACM, New York (2006)
36. Website of the Apache Collections library. https://commons.apache.org/proper/commons-collections/
37. Website of the Java Genetic Algorithms Package. http://jgap.sourceforge.net
38. Website of the JGrapht library. https://jgrapht.org/

Author Index