HAL
archives-ouvertes.fr

System-Level State Equality Detection for the Formal Dynamic Verification of Legacy Distributed Applications

Marion Guthmuller, Gabriel Corona, Martin Quinson

▶ **To cite this version:**

Marion Guthmuller, Gabriel Corona, Martin Quinson. System-Level State Equality Detection for the Formal Dynamic Verification of Legacy Distributed Applications. 2015. hal-01558049

HAL Id: hal-01558049
https://hal.inria.fr/hal-01558049

Preprint submitted on 7 Jul 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

System-level State Equality Detection for the Formal Dynamic Verification of Legacy Distributed Applications

Marion Guthmuller[a], Gabriel Corona[a], Martin Quinson[a,b,*]

[a] *LORIA laboratory (Université de Lorraine, CNRS, Inria Veridis), Nancy, France.*
[b] *IRISA laboratory (ENS Rennes, CNRS, Inria Myriads), Rennes, France.*

Abstract

The ever increasing complexity of distributed systems mandates to formally verify their design and implementation. Unfortunately, the common approaches and existing tools to formally establish the correctness of these systems remain hardly applicable to most legacy HPC applications, that are commonly written in Fortran or C/C++, using the MPI standard.

This work addresses the problem of automatically detecting at system-level the equality of the application's state. This allows to automatically verify safety and liveness properties on legacy HPC applications. We present how this state equality detection can be achieved without any source code static analysis, but at runtime using memory introspection and classical debugging techniques.

We demonstrate the effectiveness of our approach through the exhaustive verification of several programs from the MPICH3 test suite and through the partial termination analysis of some applications from the Competition on Software Verification (SV-COMP).

1. Introduction

Model checking is an appealing automated technique to establish the correctness of distributed systems. But applying this technique to legacy applications requires a complete model of the application. Manually building such models is error-prone and labor-intensive. Keeping the resulting model up to date when the real application is modified constitutes another challenge. A guided approach such as the CEGAR abstraction/refinement methodology [1] can ease this modeling step, but the user still needs a high level of expertise in formal methods. Static code analysis [2] can automatically reconstruct a model, thus removing the burden induced by the modeling step. This interesting approach is used in many existing tools [3, 4, 5]. Our work is in line with another approach called *formal dynamic verification* (or execution-based model checking), where

*Corresponding author.

the model is not explicitly known but only implicitly explored through the actual execution of the real application. The verification is thus performed on the concrete implementation of the application. In some sense, it is orthogonal with the static analysis, as static analysis and dynamic verification gather complementary but not redundant information: these approaches could be combined in the same tool.

Ultimately, our goal is to dynamically verify unmodified legacy distributed applications composed of sequential processes interacting through message passing. We do not aim at *certifying* such applications, but at finding issues. We dynamically verify real applications toward bug finding through falsification.

We build upon the SimGrid framework, initially intended to assess the performance of distributed applications [6], and upon the SimGridMC module to dynamically verify the correctness of these applications [7]. It exhaustively explores the execution paths that the verified application could follow from a initial situation provided by the user. The considered non-deterministic choices consist in the message reception order: when two messages A and B are in flight, the verification consists in first exploring the execution path where the message A is delivered before B. The application is then rewinded to explore the execution path where B is delivered first. Since we work with legacy applications, the state consists in the heap, the stacks and all global variables. Each transition encompasses a message reception by a process, and its sequential execution up to the next message exchange. The sequential execution blocks are supposed to be deterministic. We believe that these assumptions are realistic and sufficient for mono-threaded MPI-like computational applications, which outcome typically only depends on the input data provided in the initial state and on the message delivery order. Leveraging the simulator architecture to fold the verified application into a single local process makes it easier to inspect, checkpoint and rewind the complete system state. The state space explosion problem is mitigated through Dynamic Partial Ordering Reduction (DPOR) [7].

In the current work, we tackle the problem of comparing system-level states of arbitrary legacy applications written in Fortran or C/C++. Detecting state equalities is mandatory to detect infinite executions and cycles, and makes it possible to verify liveness properties. Specifically, this article makes the following contributions: we detail the challenges posed by the system-level state equality detection. We propose solutions to mitigate each of those difficulties, leveraging debugging techniques to retrieve semantic information on the application. We show the practical effectiveness of our proposal through several sets of experiments: we detect liveness violations in a custom MPI code; we show the absence of infinite execution paths in programs from a Software Verification Competition; we exhaustively explore an infinite-time MPI application as well as several of the MPI applications from the official MPICH3 testsuite.

The remainder of this article is organized as follows: Section 2 presents why state equality is an important problem for the formal verification of legacy applications. Section 3 details our contribution, which is evaluated in Section 4. Section 5 presents the related work while Section 6 concludes this article.

2. State Equality for the Formal Verification of Legacy Applications

Prior to this work, SimGridMC was *stateless*: only the system's initial state was checkpointed. When rewinding the application, the initial state was first restored and then all transitions leading to the desired state were replayed. This section presents several use cases of the state equality in the context of the formal verification of legacy distributed applications.

2.1. Efficiently Verifying HPC Programs and State Equality Reduction

The stateless approach was satisfying in the considered context of Peer-to-Peer (P2P) protocols, but it is less adapted to HPC applications, as the computations make the execution path highly time-consuming in this case. It is then interesting to checkpoint more states to directly restore the desired state instead of reconstructing it iteratively. Another advantage of the stateful exploration is to reduce the size of the explored state space by cutting the exploration when reaching a state that was already explored. In principle, this reduction technique is complementary to other ones such as DPOR.

The first limitation of the existing stateful verification tools is that they either only save parts of the system state after decomposition [8] or only save a bitstate hash of each reachable state rather than the full state [9]. This is probably due to classical memory space restrictions, but not saving the intermediate states forces to recompute them on need. Nowadays, some computers are equipped with terabytes of memory. This makes it possible (and thus appealing) to checkpoint and analyze the whole memory state of medium-size applications, or smaller instances of legacy HPC applications.

Another limitation of the existing stateful verification tools is that they target abstract models. Adapting these techniques to legacy HPC applications is technically very challenging because we usually lack the abstract model of these applications that are typically written in Fortran or C/C++.

Instead of adapting bitstate hash techniques to legacy applications, we checkpoint the full application state: the heap, the stacks and the sections containing the global variables are copied. Capturing thousands to million of states, as required by typical verifications, can rapidly exhaust the available memory. In many applications, only a small part of the memory changes between consecutive states. In some applications 99% of the memory pages do not change between consecutive states. Our snapshots are thus compacted by exploiting the similarity between states: identical memory pages (ie, memory segments of size 4KiB) are shared between snapshots.

2.2. Verifying Arbitrary Liveness Properties on Legacy Code

The execution loops detected by the state equality mechanism constitute *non-progressive cycles*. They play a central role in the verification of liveness properties, since the counterexample to liveness properties are infinite paths. If the application state size is bounded (in particular, if the stack size is bounded), most infinite paths contain such an execution loop. Liveness properties are then

verified through the search of acceptance cycles in the Cartesian product of the application with a Büchi automaton encoding the negation of the verified property. If found, such an acceptance cycle denotes an infinite execution path which constitutes a counterexample to the property.

This approach can sometimes be used to falsify the program termination using the property *"Always, Eventually, the program terminates"*. If it is naturally impossible to solve the Halting Problem in all generality, it remains sometimes possible to prove whether a given program terminates or not [10]. Dynamic verification can check the termination of any finite protocol, although sometimes inefficiently) It could also correctly diagnose applications that do not terminate because of non-progressive cycles, provided that these cycles are detected. This approach fails on applications which do not terminate but whose state changes infinitely often, which is consistent with the indecidability of the Halting Problem.

MaceMC [11] is the verification tool that comes closest to our work since it is able to verify some liveness properties on concrete implementations of distributed systems. Instead of detecting the acceptance cycles, it looks for the so-called critical transition that plunged the property into a violation. The exploration performance is then improved using state hashing. This approach remains however limited to the restricted set of liveness properties that can be expressed as $\Box\Diamond p$ (*Always Eventually* p) where p is a logical predicate, while detecting acceptance cycles can be used to verify arbitrary LTL_{-X} formula.

2.3. Verifying (infinite-time) Cyclic Protocols

Non-progressive cycles constitute an inherent part of a whole class of applications, such as the cyclic protocols which react to external periodic events, as most P2P protocols do.

Stateless DPOR cannot study such systems because even if it reduces the amount of explored interleavings by detecting the independent actions, it does not detect all execution cycles. The state space is still infinite after the DPOR and thus cannot be explored explicitly during a dynamic verification. On the contrary, a state equality reduction cuts the repetitive patterns of the application behavior, which permits the exhaustive verification of cyclic protocols.

3. System-level State Equality Detection

The stateful verification mandated by the previously presented use cases requires to efficiently determine at system-level whether two given application states are semantically equal or not. All relevant memory area must be included in this comparison, including global variables, the heap, the simulated processes' stacks, as well as the simulator's network state.

Comparing memory byte per byte is not sufficient to detect the state equality, as it detects many syntactic differences that are not significant to the application semantic. For example, the numerical values of pointers on different memory areas are syntactically different while the pointed memory areas could

be semantically equivalent. Such false negative must be avoided by all means, as the tool could fail to detect some state equalities, possibly leading to the non-detection of property violations. The key to a better state equality detection lies in the ability to introspect the memory and to reconstruct the semantic of bits.

For Java applications, the whole memory is handled by the virtual machine. Using the meta-data known to the JVM, it is thus possible to reconstruct the semantic of each byte in memory, making it possible to detect system state equalities. However, this would probably require to modify the JVM directly, as Java introspection is more intended to explore the values of objects' content rather than exploring the meaning of memory locations. Adding the needed meta-data and keeping them in sync in the whole JVM source code constitutes a daunting task, but remains feasible [12].

However, Java is rarely used in the context of HPC applications using MPI, calling for another approach that would be applicable to C/C++ applications. In [13], the user must provide a hashing function that can be used to detect the state equalities. Detecting system state equalities is easier when the complete data semantic is known, but this remains a burdensome and highly error-prone task, hardly adaptable to complex systems. In the following, we propose a generic solution which operates at the operating system level. Instead of specifying the segments of memory that are relevant to the system's semantic, we start by considering the whole system memory, and optionally ignore some sections that are known to be irrelevant.

The system state that we consider aggregates the application's global variables, the application's heap and the stack of each process (Fig. 1). The global variables of the simulator are also included in the comparison, as they contain the network's state during the simulation. At the OS-level, these data are stored in several memory *segments* that must be considered separately. In the following, we detail our approach using Linux as an example, but our implementation was successfully ported to FreeBSD. Porting our code to other operating systems is certainly possible, even if it would be somehow harder for non-Unix systems.

In the remainder of this section, we detail the causes of the syntactic differences that defeat the byte per byte comparison of memory segments. For each of these challenges, we show how additional information could be retrieved from the system, and leveraged to rebuild the needed semantic information. For some of these difficulties, we must rely on heuristics that could conclude on the difference of semantically equal states. This must be avoided when possible, as this could jeopardize the soundness of the verification process. That is why we do not aim at certifying the verified applications, but rather at *finding bugs* that are hard to detect on real applications.

3.1. Uninitialized Data and Memory Overprovisioning

The first source of syntactic differences that are not relevant to the semantic equality lies in uninitialized data. In contrast to Java, the memory handed over to the user is not initialized by the C/C++ runtime because of performance

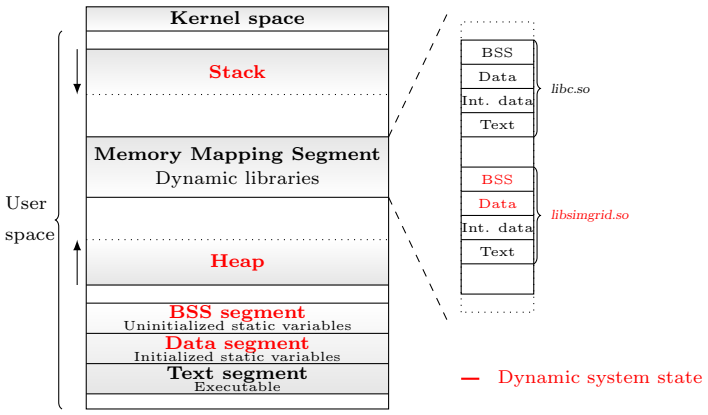

Figure 1: Memory layout of a process.

concerns. This is true for the blocks allocated on the heap (that still contain the content of their previous use), and also for the stack (which still contains the frames of returned functions).

In addition, most dynamic memory allocation libraries (such as `malloc` implementations) allocate memory chunks whose sizes are powers of two to avoid the memory fragmentation. For example, a memory area of 64 bytes will be used to serve a request of 48 bytes. It is expected that the application only uses the requested area while the extra area remains unused (Fig. 2). In our context, this memory overprovisioning may result in irrelevant byte differences that are not trivial to address at the application level, as the extra memory area are not directly known to the application but still contain the arbitrary data, written by the previous use of that chunk.

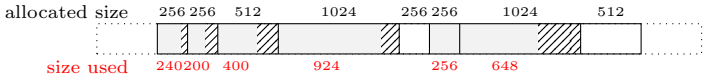

Figure 2: Memory allocation with overprovisioning.

SimGridMC provides a specific implementation of `malloc` to address both problems of uninitialized heap area and heap over-provisioning. A first approach for heap over-provisioning is to only consider the user-requested area during the state comparison and to ignore the extra memory. But this would not work for applications that exhibit small buffer overflows. Indeed, if the application accidentally overflows from the requested area while remaining within the bounds of the corresponding malloc chunk, the problem will remain usually harmless, but our analysis would fail to capture the whole state. Instead, we address this issue by filling each newly allocated area with zeroes before handing it to the

application. This ensures that every value is specified while remaining robust to limited buffer overflows. Although SimGridMC is not initially intended to detect this kind of bug, we could add an option to detect and report heap overflows, *e.g.* with `mprotect`ed pages located after each allocated block. Similarly, zeroing memory areas may hide bugs resulting from undefined behaviors and other tools and approaches should be used to track these issues.

Uninitialized data and memory over-provisioning also occur within the processes' stacks. The local variables may remain unset by the user program while the memory occupied by a stack frame is not zeroed out by the system when a function returns. We circumvent this problem by only considering the stack area that is in active stack frames and optionally by zeroing out the stack frame at the beginning of each function. This is currently implemented by modifying the assembly code generated by the compiler before giving it to the assembler: a script estimates the size of the stack frame of each function by parsing its assembly code, and adds assembly instructions that zero out the stack frame at the beginning of the function. This script has been used with both GCC and Clang for C, C++ and Fortran input. It is currently implemented for x86_64 only but could easily be ported to other targets. This post–processing must be done for the whole application and the simulator, or it could be completely omitted if the user code initialize every local variables when declaring them.

3.2. Sparse Memory and Padding Bytes

The compiler enforces memory alignment constraints to speed up the data movements between the memory and the CPU registers. The address of each variables must be a multiple of its size. For example, short integers (of size 2 bytes) must reside at even addresses. This results in sparse memory layouts with *padding bytes* added between the variables (as depicted on Fig. 3), leading to irrelevant syntactic differences if the values of those padding bytes are compared.

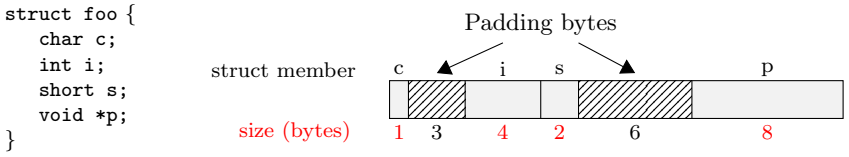

Figure 3: Data memory alignment is preserved by adding unused padding bytes.

On some platforms, it is possible to disable these padding bytes, using the `#pragma pack(1)` compiler directive. But the resulting major performance impact mandates for other solutions because misaligned data requires several cycles to move from memory to the CPU registers.

In the heap, our solution to the previous issue happens to address that problem too, since the padding bytes are zeroed out and thus specified. This remains problematic in the stack despite zeroing it out for two reasons. First, each stack frame is protected by the system against buffer overflow and *stack*

smashing security attacks through the addition of arbitrary data in the stack by the system. Any change to this data is interpreted by the system as a security attack. If integrated to the comparison, this arbitrary data could defeat the state equality detection. In addition, several local variables can share the same location in the stack memory if they do not exist at the same time of the program flow. This may lead to temporally unused memory areas in the stack.

Our approach is to focus on the actual data stored on the stack and to ignore the unspecified values and padding bytes. We retrieve the needed information from the debugging symbols using two complementary libraries: libunwind[1] is used to analyze the content of each stack frame and retrieve the address and type of local variables. The memory layout of each encountered data type (size and alignment constraints) is in turn retrieved from the DWARF[2]-formatted debugging information. This information is generated by the compiler so that debuggers such as `gdb` can reconstruct the memory layout of the debugged process. Our implementation works on Linux and FreeBSD only, but this approach is portable to other systems. The DWARF format is also used on MacOS X while Windows provide the same information under another format.

The DWARF information alone is sufficient to rebuild a fine-grain semantic of each byte in the global segment. Coupled with libunwind, it is sufficient for the stack segments. It is however not sufficient to reconstruct the semantic for the heap, as the user can cast the allocated data to arbitrary types. This difficulty is addressed in the following section.

3.3. Heap Dynamic Semantic Comparison

The last and most difficult technical lock of system-state state comparison is due to the fact that the order in which memory blocks are allocated rarely matters to the application semantic while it greatly impacts their actual location in memory. Fig. 4 depicts two heaps that are semantically equivalent despite their different block orderings. The content of blocks `0x30`, `0x40` and `0x50` are syntactically different, but these differences come from the fact that the block `0x30` of one heap corresponds to the block `0x40` of the other heap. The semantic of these heaps is perfectly equivalent for the application.

Such situation often occurs in our context because the block ordering stems from the order of malloc requests, which changes when the process execution order changes, as in the dynamic verification.

Moreover, the numerical value of a pointer remains unchanged in C when the pointed area is freed. As the system may reallocate the previously freed memory, the pointed data may be completely different. It is a good habit as a programmer to set the pointer variables to NULL in this case, but this is not requested by the C standards. The manual detection of these *dangling pointers* by the user is however essential to the soundness of our approach, as

[1] `http://www.nongnu.org/libunwind/`
[2] `http://www.dwarfstd.org/`

the comparison of unrelated memory areas leads to the non-detection of the system state equality.

Even if there is no dangling pointer, detecting the semantic equality between heaps in which blocks were reordered remains hard. We cannot directly leverage DWARF debugging information, as the considered memory is allocated during the execution and thus unknown at compile time when this information is produced. Type aliasing of heap data is also a common practice that defeats the formal identification of the data types.

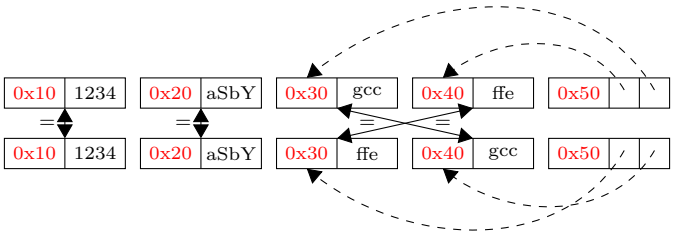

Figure 4: Two heaps syntactically different but semantically identical.

In [14], the author presents a *heap canonicalization algorithm* that can reorder the blocks in a canonical form to ease heap comparison. This approach relies on a garbage collecting mechanism in a language where all references to allocated data are clearly known to the system. A mark-and-sweep algorithm is then conducted from all variables down to the blocks. The graph traversal is deterministic by construction, and the blocks are reordered on-the-fly into a canonical form.

This approach cannot be applied as is to our context because we cannot assume that the verified application uses one of the existing garbage collector for C/C++. We can retrieve the references located in local and global variables according to their DWARF signature, but we will probably miss some references located in heap blocks as we lack any semantic information on the content of these blocks.

Because of these missing references, it is impossible to move memory blocks: any reference still pointing to an old block location would cause the application to abort immediately.

It is still possible to compare two given states on the fly using a mark-and-sweep approach. Starting from the global and local pointer variables (identified through libunwind and DWARF), all heap blocks that are reachable through identified pointers are marked as equivalent in both states. We perform a partial mark-and-sweep, starting from the variables we know and iterating on data for which we have the debugging information. This traversal can have several outcomes. It first stops when an inexplicable difference is found, indicating that both states differ. It may also manage to match all blocks, despite their

potentially different location in both heaps, proving the equality of both states' semantic.

But this traversal can only be a heuristic because of the missing typing information. First and foremost, we lack any typing information about the data on the heap. Then, the actual datatype may be unknown or hidden by `void*` type aliasing. Finally, local variables may be hidden at some execution points because of scoping rules within the function, hiding some valuable starts for the mark-and-sweep traversal.

It is thus possible that some blocks allocated on the heap were not considered during the traversal, because the pointer to them were not identified as such because of missing typing information. The remaining blocks are then compared byte per byte. If we find a difference, we attempt to explain this as a pointer difference: in each heap, we read the aligned 8 bytes that contain the differing byte as a pointer. If each value corresponds to the address of a valid heap block, the comparison iterates recursively on the designated blocks.

The comparison algorithm can detect dangling pointers in some cases (when they point to unallocated heap blocks or above the top of the stack). Better detection of dangling pointers could be achieved by avoiding to reallocate heap blocks as long as they are reachable.

The practical effectiveness of this heuristic is evaluated in Section 4.

3.4. User-identified Irrelevant Differences

Specific variables and memory areas can also be explicitly ignored during the comparison. The user indicates to ignore the step number in a cyclic protocol, the value of an iterator in a loop or any value that he considers irrelevant to its state. A different system behavior resulting from the value of the counter will be detected in the other memory areas, while this counter could be ignored.

The same mechanism is used to mask irrelevant differences caused by the simulator internals, such as the total amount of messages sent during the simulation or other similar statistics. Determining the sections to be manually marked as irrelevant can be very tedious. In the future, we plan to issue a warning when two states differ only by a very little amount of bytes, allowing the user to check whether this difference must be marked manually or not.

3.5. Summary

Table 1 summarizes the difficulties encountered during the system-level comparison of state equality, and the solution that we propose in each case for both the heap and the stack. As a final optimization, we test first and foremost several efficient criteria such as the total amount of allocated blocks, allocated data, and stack sizes of all processes. This allows to eliminate many candidate states even before traversing the heaps.

4. Experimental Evaluation

This section evaluates our contribution through three sets of experiments which illustrate the motivating examples of Section 2 that leverage state equal-

Issue	Heap solution	Stack solution
Overprovisioning	`memset` 0	Stack pointer detection
Uninitialized data	`memset` 0	Stack cleaner by binary code modification
Padding bytes	`memset` 0	DWARF + libunwind
Syntactic differences	Dynamic semantic comparison	N/A (sequential access)
Irrelevant differences	Ignore explicit areas	DWARF + libunwind + ignore

Table 1: Summary of issues and solutions for the system-level state equality detection.

ity detection in the context of dynamic verification of legacy applications. Section 4.1 focuses on the exhaustive verification of several applications distributed as part of the MPICH3 testsuite. Section 4.2 then evaluates the tool's ability to detect two kinds of liveness violation: on a custom MPI application and the termination of a program. Section 4.3 demonstrates how state equality detection enables the exhaustive exploration of cyclic applications whose behavior is regular but not bounded in time.

We used SimGrid (60,000 lines of code – git version d8710e), as our contribution is integrated to the public version of this framework. These experiments were conducted on a Intel(R) Xeon(R) CPU E7540 @ 2.00GHz, RAM 512GiB, 48 cores, with debian wheezy environment 3.2.0-4-amd64 and 3 extra packages (cmake 2.8.9, libunwind7, and gfortran 4.7.2).

In all subsequent tables, '#P' is the total amount of processes in the studied application, '# States' corresponds to the number of expanded states before finding the counterexample (or the total number of states in the case of exhaustive exploration), and 'Depth' is the depth in which the counterexample has been found (only for the verification of a liveness property).

4.1. Exhaustive Verification of MPICH3 Testcases

This first experiment focuses on the MPICH3 testsuite[3]. This testsuite allows to test the standard compliance of any MPI implementations. Not all of the tested features are currently implemented in SimGrid so we eliminated the corresponding tests. We verified several applications from this testsuite, each of them lasting about 1,300 lines written in C or Fortran. These tests were not modified from their original version. In particular, we did not manually mark any sections as irrelevant, as presented in Section 3.4. Since our tool is written in C and C++ itself, these experiments demonstrate our ability to verify programs written in C/C++ or Fortran.

Our verification algorithm consisted in exploring every orders of network message delivery that respect the causal constraint. We consider the blocks between message exchanges as atomic. We took one snapshot of the whole application after each message exchange, sharing identical pages between snapshots to reduce the memory consumption. We compared each snapshot to all

[3]The MPICH project: `http://www.mpich.org`

Application	# P	Stateless exploration			Stateful exploration		
		# States	Time	Memory	# States	Time	Memory
bcasttest (C)	3	> 1 million	> 24 h	-	4,823	25 min 23 s	1.01 GiB
bcastzerotype (C)	5	12,135,948	1 h 22 min	0.35 GiB	4,734	6 min 50 s	0.84 GiB
	6	> 263 millions	> 24 h	-	56,054	10 h 02 min	7.16 GiB
commcreate1 (C)	4	102,289	44 s	0.35 GiB	1,556	1 min 28 s	0.49 GiB
	5	12,710,034	1 h 23 min	0.35 GiB	8,359	25 min	1.48 GiB
	6	> 274 millions	> 24 h	-	99,235	24 h 55 min	14.18 GiB
dup (C)	2	907	2 s	0.35 GiB	81	2 s	0.35 GiB
	3	138,678	43 s	0.35 GiB	405	7 s	0.36 GiB
	4	78,082,843	7 h 36 min	0.35 GiB	2,352	1 min 16 s	0.62 GiB
	5	> 276 millions	> 24 h	-	39,263	7 h 06 min	5.83 GiB
groupcreate (C)	4	102,289	31 s	0.35 GiB	1,205	44 s	0.44 GiB
	5	12,710,034	1 h 22 min	0.35 GiB	6,237	11 min 21 s	1.21 GiB
	6	> 272 millions	> 24 h	-	80,878	17 h 35 min	11.47 GiB
inplacef (Fortran)	3	> 182 millions	> 24 h	-	2,941	1 min 15 s	0.73 GiB
op_commutative (C)	3	358	2 s	0.35 GiB	94	2 s	0.35 GiB
	4	102,289	31 s	0.35 GiB	1,545	1 min 20 s	0.48 GiB
	5	12,710,034	1 h 23 min	0.35 GiB	10,998	53 min 29 s	1.79 GiB
sendrecv2 (C)	2	> 156 millions	> 24 h	-	1,877	30 s	0.49 GiB

Table 2: Exhaustive verification of MPI applications from MPICH3 testsuite.

others using the contribution presented in this work, and then searched for non-progressive cycles (*i.e.,* livelocks) in the resulting graph.

Table 2 compares the performance of the stateless and stateful explorations of these applications. The DPOR reduction was not activated in these experiments since our current DPOR implementation remains stateless and is incompatible with the state equality reduction due to technical reasons. We managed to exhaustively explore the state space of these applications for up to 6 processes (no error has been found during these explorations). These results clearly demonstrate the necessity of the state reduction to exhaustively explore the state space of even basic tests with few processes.

4.2. Dynamic verification of liveness properties

This experimentation part is divided into two experiments. We demonstrate firstly the ability of our tool to detect a liveness violation on a custom MPI application. Then we study the termination on some programs from the Competition on Software Verification (SV-COMP).

4.2.1. Custom Implementation of Centralized Mutual Exclusion

The first experiment uses a custom MPI implementation of the centralized mutual exclusion algorithm (about 100 lines of code), where a coordinator grants infinitely often a mutex to the clients that request it. We introduce an error so that one of the clients never gets the requested critical section (its requests are discarded). We verify the following liveness property: $\square(r \rightarrow \Diamond cs)$ (*any process that requests (r) must obtain the critical section (cs)*). Since one client never gets the cs, this property is actually violated.

This example is the only case where we had to manually mark a given variable as irrelevant to the comparison (see Section 3.4). We ignored a field in a data structure returned by MPI. In this specific case, this could probably be automatized in the future.

A counterexample is accordingly found by our tool. The duration of this exploration can vary from few seconds if the violation in on the first explored branch, to several hours if the violation is in another branch. Table 3 presents the worst case results, when the buggy process is the last one. With such infinite-time applications, a stateful exploration is mandatory to detect and avoid non-progressive cycles that could prevent the verification from terminating.

# P	**# States**	**Time**	**Memory**	**Depth**
3	101	2,5 s	0.35 GiB	94
4	287	3 s	0.37 GiB	267
5	960	5,5 s	0.62 GiB	890
8	68,128	18 min 40 s	5 GiB	62,831
10	$>$ 200,000	$>$ 1 h	$>$ 16 GiB	-

Table 3: Verification of the property $\square(r \rightarrow \Diamond cs)$ on a bugged mutual exclusion (worse case).

4.2.2. Program Termination of SV Competition

The second experiment for the dynamic verification of liveness properties focuses on a very specific liveness property: *"Always, Eventually, the program terminates"*. As explained in section **??**, detecting a non-progressive cycle (that is, a cycle of infinite length composed of a finite amount of states) is enough to determine that the program will never terminate. Note that a program without any non-progressive cycle can still fail to terminate, which is consistent with the fact that the Halting Problem is undecidable.

To study this liveness property, we selected some test cases of the Software Verification Competition (SV-COMP), which aims at assessing the state of the art in the field of software verification. The competitors of SV-COMP are usually based on static analysis, but we wanted to evaluate whether the same results could be obtained through dynamic analysis. Since our tool considers the code between MPI communications to be atomic, we had to modify the proposed benchmarks to add some fake MPI communications to break down the execution path into separate transitions.

The *Program Termination* category of SV-COMP'15 entails 18 test cases that exhibits a non termination. 14 of these tests present an infinite state space, for example with an integer variable that is incremented at each step. Even if static analysis tools can handle such programs, our approach is not effective as it is not possible to explore explicitly and exhaustively such state space. As a result, these cases were not included in our evaluation.

The remaining tests are rather simple: two of them (WhileTrue and Madrid) are infinite loops containing only constant affectations. The third one (Rotation180) permutes three values while alternating their sign. In the fourth one (NonTerminationSimple5), a variable is randomly either incremented or decremented at each step of an infinite loop.

Our tool manages to find the non-progressive cycle in a matter of seconds in each of these cases involving only a few states, but for the last program. NonTerminationSimple5 presents several different infinite execution paths: the variable can iterate over an infinite amount of values if it is constantly incremented (resp. decremented). The variable can also infinitely loop over only 2 values if it is alternatively incremented and decremented. Our tool detects the second case when limiting the maximum exploration depth so that the second branch of the second alternative gets explored (increasing after decreasing, or the opposite).

Similarly to the 14 tests that were not included in this evaluation, the first kind of infinite execution path presented by NonTerminationSimple5 entails an infinite amount of different states. This defeats our approach of dynamic verification of termination, as it is based on the detection of execution loops leading to *previously evaluated* states. This is a clear limitation over the scope of our verification methodology, somewhat reducing our impact on this use case. This limitation should be balanced with the versatility of liveness properties that can be verified, which can be much more complex than the termination property. In addition, this limitation does not reduce the effectiveness of the state equality detection presented in this article, that can also be leveraged in the other use cases and that perfectly works in this evaluation when it is applicable.

4.3. Stateful exhaustive exploration of infinite-time application

The goal of this experiment is to prove the applicability of our approach to cyclic applications. To that extend, we use a modified version of the centralized mutual exclusion algorithm used previously. The bug was removed, and the processes were modified to loops forever, requesting (resp. granting) the critical section at each step.

We could exhaustively verify this infinite protocol in four minutes for 3 processes (8,216 states found, 1 GiB) and almost 18h for 4 processes (480,997 states found, 49 GiB).

5. Related work

Since this work leverages dynamic analysis techniques, we do not discuss approaches based on static analysis. These orthogonal approaches could be used

jointly to benefit both several sources of information during the verification.

Many tools exploit dynamic analysis either for the verification of C programs [15, 4, 16] with abstract interpretation in some cases and focused on memory-related errors [17, 18] or in particular for the termination analysis [19, 20]. This approach is also commonly used for Java [12] or object-based program analysis, with a garbage collection mechanism [21]. As explained in Section 3, it is easier to implement this approach for languages served by a virtual machine, as the meta-data known to the virtual machine are precious in this context. The semantic of any byte of memory is known to the VM, that can in particular retrieve and follow pointers. This operation is much harder in our case. Our heuristic must deal with generic pointers, invisible variables, dangling pointers and otherwise unknown pointers, that cannot happen with a garbage collector.

To the best of our knowledge, our contribution is the first extension of the heap canonicalization algorithm proposed in [14] to languages without automatic garbage collection. Strictly speaking, this is not a heap canonicalization algorithm, as it does not change the order of blocks in memory, but it fulfills the same need for heap semantic comparison. Our approach allows to perform the verification on the actual program memory without using state vectors. This direct use of the actual state can reduce the risk of abstraction errors.

This state equality comparison is a major technical lock to the formal verification of HPC applications. Although the need of such methods is well acknowledged in this context [22], there exist few verification tools for MPI applications. To the best of our knowledge, MPI-CHECK [23] is the only verification tool of Fortran 90 MPI programs. Thanks to compile-time and runtime tests, it detects deadlocks and some inconsistencies in MPI calls such as negative message lengths. But the exploration is not exhaustive and the achieved tests are limited. Gauss [24] and MPI-Spin [25] are model extractors for MPI applications which can then be checked with Zing [26] and SPIN [27] respectively.

Finally, ISP [28] and its distributed counterpart DAMPI [29] are dynamic verifiers specifically tailored for the verification of MPI applications written in C. They check for deadlock and local assertion violations without requiring users to manually model their code. ISP hijacks the PMPI profiling interface that is normally intended for tracing tools to gather information about the MPI calls. ISP mediates these MPI calls and performs the dynamic verification at that level. A distributed protocol is then used between nodes to determine which messages should be delayed within the "profiling" call, and which ones can be proceed (after being rewritten).

This approach leads however to two major drawbacks. First, collective operations are seen as atomic calls from the profiling interface perspective. The point-to-point communications that compose these collectives are completely invisible at this level. It means that unlike SimGridMC, ISP cannot properly verify collective operations, that must be assumed intrinsically correct. This seems unfortunate given the current momentum on asynchronous group collectives, that are known to be error-prone to implement [22]. Moreover, the real execution of the application on a real distributed platform may pose subtle

challenges to ensure that the simulation is reproducible. Likewise, rewriting the MPI calls may change the synchronization semantic, that may depend on the buffer sizes in some border cases [30]. SimGridMC is based on the versatile SimGrid framework instead which is reproducible by design. It is much easier to observe and control the distributed system when it is folded into a unique system process, resulting in simpler and thus more robust setups.

6. Conclusion and Future Work

System state equality detection is essential for the dynamic verification of applications. It is important to verify arbitrary liveness properties, to explore exhaustively infinite cyclic protocols and constitutes an efficient reduction mechanism during stateful explorations. Detecting system state equalities is however much harder with legacy distributed applications than with abstract models.

In this article, we detailed the root causes of these difficulties, and proposed various solutions leveraging debugging information and tools. We presented a heuristic to detect the state equality of an application at system-level. Even if missing typing information may lead to false negative, this memory introspection technique is to the best of our knowledge the first solution to reconstruct semantic information about systems that use programming languages without automatic garbage collection.

Despite its apparent simplicity, our heuristic proves efficient in practice. It was evaluated on the dynamic formal verification of several properties on various test cases from the official MPICH3 testsuite. We were able to actually verify and exhaustively explore these legacy distributed applications.

This contribution is integrated in the SimGrid framework[4], making it possible to jointly evaluate the correctness and performance of distributed applications.

Our approach cannot be used for certification but only for bug finding as it relies on a heuristic. The improvement beyond prior work on verifying unmodified legacy HPC programs is many-fold: any safety or liveness property can be verified, and some class of infinite time-applications can be verified. These properties can be assessed on arbitrary C/C++ or Fortran mono-threaded applications based on MPI, using only the debugging symbols. This may be applicable even if the source code is not available.

In the future, we want to combine our approach with other classical methods to increase the amount of available information during the reduction. For example, visibility information that can be extracted from instrumentation are mandatory to combine the DPOR and stateful reductions [31]. Memory graphs that can be reconstructed from static analysis would help extending the applicability of our approach to multithreaded MPI applications. We also want to evaluate complex MPI code, such as the asynchronous collective calls implemented in MPICH and OpenMPI or even full MPI applications.

[4]SimGrid is freely available from `http://simgrid.org/`

Acknowledgments

This work was partially funded by the ANR project SONGS (ANR-11-INFRA-13). Some experiments were carried out using the Grid'5000 experimental testbed, being developed under the Inria ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr/`). The authors would like to thank Matthieu Volat for his precious help to port this work to the FreeBSD operating system.

[1] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-Guided Abstraction Refinement, in: 12th International Conference on Computer Aided Verification, Springer, Berlin, Heidelberg, 2000, pp. 154–169.
URL `http://dx.doi.org/10.1007/10722167_15`

[2] D. Engler, Static Analysis Versus Model Checking for Bug Finding, in: 16th International Conference on Concurrency Theory, CONCUR 2005, Springer, Berlin, Heidelberg, 2005, pp. 1–1. doi:10.1007/11539452_1.
URL `http://dx.doi.org/10.1007/11539452_1`

[3] T. Ball, V. Levin, S. K. Rajamani, A Decade of Software Model Checking with SLAM, Communications of the ACM (2011) 68–76.

[4] D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar, The Software Model Checker Blast: Applications to Software Engineering, International Journal on Software Tools for Technology Transfer (2007) 505–525.

[5] A. Gurfinkel, O. Wei, M. Chechik, YASM: A Software Model-checker for Verification and Refutation, in: 18th International Conference Computer Aided Verification, CAV '06, Springer, Berlin, Heidelberg, 2006, pp. 170–174.

[6] H. Casanova, A. Giersch, A. Legrand, M. Quinson, F. Suter, Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms, Journal of Parallel and Distributed Computing 74 (10) (2014) 2899–2917.
URL `http://hal.inria.fr/hal-01017319`

[7] S. Merz, M. Quinson, C. Rosa, SimGrid MC: Verification Support for a Multi-API Simulation Platform, in: Joint 13th IFIP International Conference on Formal Techniques for Distributed Systems, FMOODS / FORTE 2011, Springer, Berlin, Heidelberg, 2011, pp. 274–288.
URL `http://dx.doi.org/10.1007/978-3-642-21461-5_18`

[8] H. Saissi, P. Bokor, C. A. Muftuoglu, N. Suri, M. Serafini, Efficient Verification of Distributed Protocols Using Stateful Model Checking, in: 32nd International Symposium on Reliable Distributed Systems, IEEE Computer Society, Los Alamitos, CA, USA, 2013, pp. 133–142.
URL `http://dx.doi.org/10.1109/SRDS.2013.22`

[9] G. Holzmann, An Analysis of Bitstate Hashing, Formal Methods in System Design 13 (3) (1998) 289–307.
URL http://dx.doi.org/10.1023/A:1008696026254

[10] B. Cook, A. Podelski, A. Rybalchenko, Proving Program Termination, Communications of the ACM (2011) 88–98.
URL http://dx.doi.org/10.1145/1941487.1941509

[11] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, A. M. Vahdat, Mace: Language Support for Building Distributed Systems, in: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, ACM, New York, NY, USA, 2007, pp. 179–188.
URL http://doi.acm.org/10.1145/1250734.1250755

[12] K. Havelund, T. Pressburger, Model Checking Java Programs using Java PathFinder, International Journal on Software Tools for Technology Transfer 2 (4) (2000) 366–381.
URL http://dx.doi.org/10.1007/s100090050043

[13] P. Fonseca, C. Li, R. Rodrigues, Finding Complex Concurrency Bugs in Large Multi-threaded Applications, in: Sixth Conference on Computer Systems, EuroSys'11, ACM, New York, NY, USA, 2011, pp. 215–228.
URL http://doi.acm.org/10.1145/1966445.1966465

[14] R. Iosif, Symmetry Reductions for Model Checking of Concurrent Dynamic Software, International Journal on Software Tools for Technology Transfer 6 (4) (2004) 302–319.
URL http://dx.doi.org/10.1007/s10009-004-0154-9

[15] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, The ASTREÉ Analyzer, in: M. Sagiv (Ed.), 14th European Symposium on Programming Languages and Systems, Held as Part of the Joint European Conferences on Theory and Practice of Software, ESOP / ETAPS 2005, Springer Berlin Heidelberg, 2005, pp. 21–30.
URL http://dx.doi.org/10.1007/978-3-540-31987-0_3

[16] R. Bagnara, P. M. Hill, A. Pescetti, E. Zaffanella, On the Design of Generic Static Analyzers for Modern Imperative Languages, arXiv preprint (2008).
URL http://arxiv.org/abs/cs/0703116

[17] K. Dudka, P. Müller, P. Peringer, T. Vojnar, Predator: A Tool for Verification of Low-Level List Manipulation, in: 19th International Conference Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the European Joint Conferences on Theory and Practice of Software, TACAS / ETAPS 2013, Springer, Berlin, Heidelberg, 2013, pp. 627–629.
URL http://dx.doi.org/10.1007/978-3-642-36742-7_49

[18] E. Clarke, D. Kroening, F. Lerda, A Tool for Checking ANSI-C Programs, in: 10th International Conference Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the Joint European Conferences on Theory and Practice of Software, TACAS / ETAPS 2004, Springer, Berlin, Heidelberg, 2004, pp. 168–176.
URL http://dx.doi.org/10.1007/978-3-540-24730-2_15

[19] S. Falke, D. Kapur, C. Sinz, Termination Analysis of C Programs Using Compiler Intermediate Languages, in: 22nd International Conference on Rewriting Techniques and Applications (RTA'11), Vol. 10 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2011, pp. 41–50. doi:http://dx.doi.org/10.4230/LIPIcs.RTA.2011.41.
URL http://drops.dagstuhl.de/opus/volltexte/2011/3123

[20] C. Alias, A. Darte, P. Feautrier, L. Gonnord, Rank: a tool to check program termination and computational complexity, in: IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013, Luxembourg, 2013, pp. 238–238.
URL https://hal.inria.fr/hal-00801571

[21] N. H. M. Aan de Brugh, V. Y. Nguyen, T. C. Ruys, MoonWalker: Verification of .NET Programs, in: 15th International Conference Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the Joint European Conferences on Theory and Practice of Software, TACAS / ETAPS 2009, Springer, Berlin, Heidelberg, 2009, pp. 170–173.
URL http://dx.doi.org/10.1007/978-3-642-00768-2_15

[22] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, G. Bronevetsky, Formal Analysis of MPI-based Parallel Programs, Communication of the ACM 54 (12) (2011) 82–91.
URL http://doi.acm.org/10.1145/2043174.2043194

[23] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, Y. Zou, MPI-CHECK: a Tool for Checking Fortran 90 MPI Programs, Concurrency and Computation: Practice and Experience 15 (2) (2003) 93–100.
URL http://dx.doi.org/10.1002/cpe.705

[24] R. Palmer, S. Barrus, Y. Yang, G. Gopalakrishnan, R. M. Kirby, Gauss: A Framework for Verifying Scientific Computing Software, Electronic Notes in Theoretical Computer Science 144 (3) (2006) 95 – 106, proceedings of the Workshop on Software Model Checking (SoftMC 2005).
URL http://dx.doi.org/10.1016/j.entcs.2006.01.007

[25] S. F. Siegel, Model Checking Nonblocking MPI Programs, in: 8th International Conference Verification, Model Checking, and Abstract Interpretation, VMCAI'07, Springer, Berlin, Heidelberg, 2007, pp. 44–58.
URL http://dx.doi.org/10.1007/978-3-540-69738-1_3

[26] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, Y. Xie, Zing: A Model Checker for Concurrent Software, in: 16th International Conference Computer Aided Verification, CAV'04, Springer, Berlin, Heidelberg, 2004, pp. 484–487.
URL http://dx.doi.org/10.1007/978-3-540-27813-9_42

[27] G. Holzmann, the Spin Model Checker: Primer and Reference Manual, 1st Edition, Addison-Wesley Professional, 2003.

[28] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, R. Thakur, Formal Verification of Practical MPI Programs, SIGPLAN Notices 44 (4) (2009) 261–270.

[29] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, G. Bronevetsky, A Scalable and Distributed Dynamic Formal Verifier for MPI Programs, in: 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10, IEEE Computer Society, 2010, pp. 1–10.

[30] D. H. Ahn, G. L. Lee, G. Gopalakrishnan, Z. Rakamarić, M. Schulz, I. Laguna, Overcoming Extreme-scale Reproducibility Challenges Through a Unified, Targeted, and Multilevel Toolset, in: 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering, SE-HPCCSE '13, ACM, New York, NY, USA, 2013, pp. 41–44.
URL http://doi.acm.org/10.1145/2532352.2532357

[31] Y. Yang, X. Chen, G. Gopalakrishnan, R. M. Kirby, Efficient Stateful Dynamic Partial Order Reduction, in: Model Checking Software: 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008 Proceedings, Springer, Berlin, Heidelberg, 2008, pp. 288–305.
URL http://dx.doi.org/10.1007/978-3-540-85114-1_20