NetworCat: Applying Analysis Techniques of Shared Memory Software on Message-Passing Distributed Systems

Levente Bajczi [1*] and Vince Molnár [1*]

[1*]Department of Artificial Intelligence and Systems Engineering, Budapest University of Technology and Economics, 2, Magyar tudósok körútja, Budapest, 1117, Hungary.

*Corresponding author(s). E-mail(s): molnar.vince@vik.bme.hu;

Abstract

Communication models are a key aspect in the design and implementation of distributed system architectures. Application logic must consider the guarantees of these models, which fundamentally influence its correctness. Modern multi-core processor architectures face a similar problem when it comes to accessing shared memory: the guarantees of an architecture have a fundamental impact on the observable behavior of software. The formalization of these guarantees in a declarative way has led to powerful tools and algorithms to define reusable constraints on patterns of memory access events and their relationships, enabling the efficient description and automatic formal analysis of software properties with respect to a specific architecture. The Cat memory modeling language provides a standard means of specifying these constraints. Despite the parallels, the axiomatic modeling and analysis of communication models in distributed systems remain a relatively unexplored area. In this paper, we address this gap and demonstrate how communication models can be mapped to the Cat language. We create a standard library of reusable patterns and demonstrate our approach, called NetworCat, on the simple examples of UDP and TCP, and we also present its applicability to the vastly configurable OMG-DDS service. This adaptation-based approach enables the use of ever-improving verification tools built for shared memory concurrency on distributed systems. We believe this not only benefits distributed system analyses by broadening the toolset for verification but also positively impacts the field of memory-model-aware verification by widening its audience to another domain.

Keywords: distributed systems, systems modeling, formal verification, cat, OMG-DDS

1 Introduction

Efficient formal verification and modeling of distributed systems and applications has long been a primary target of formal methods research [1–5]. In addition to the highly concurrent, asynchronous setting, these applications generally rely on stacks of complicated communication models that aim to compensate for unreliable communication channels.

The complexity of these models is often on par with the application implemented on top of them – directly encoding their operational semantics together with the application is, therefore, impractical. A common practice is to verify that models indeed fulfill their guarantees, then verify the distributed application modulo these guarantees. Furthermore, components of the system can sometimes be analyzed in isolation, composing the results of these smaller sub-problems into a verdict [2, 6].

1.1 Analysis of Distributed Systems

An idea that systems could be decomposed into communicating components, which in turn can be defined and analyzed with relative ease, led to the creation of *process algebras* [7–9]. Process algebras enable the system designer to specify sequential and parallel composition of components, along with message-based communications and synchronizations between them.

One downside of process algebras is their "imperative" approach to specifying component behavior when the communicational model is more complex than a simple synchronous message-passing action (e.g., retransmission or lossy sending of data necessitates either ghost participants or duplication of actions)[1]. In this work, we aim to *decouple* the semantics from the execution model, thus allowing the analysis of a system with respect to many different execution models. There has been substantial work on moving towards a *declarative* approach for communication models in distributed algorithms [1, 4, 5, 10], which proved successful in finding previously unknown bugs in distributed models and algorithms. Furthermore, there are other approaches such as TLA+ [11], IronFleet [12], EventML [13], Verdi [14] or Disel [15], which provide frameworks and languages for the analysis of distributed systems.

However, most of such works rely on a well-established communication link among the components, at best distinguishing between synchronous and asynchronous channels and considering complete channel failures [4]. In some exceptions to this, the semantics of channels were encoded as a separate *communication model* using custom extensions to the declarative specification framework (such as Datalog or Network Datalog) used to model the distributed algorithms themselves [3, 16]. The granularity of the proposed specification languages is usually low, capturing only simple properties such as channel failure (message loss), duplication, and delays.

1.2 Similarities to Weak Memory Verification

During the last decade, a similar direction has been emerging in the field of multicore processor architectures [17]. Modern CPUs are themselves small systems with

[1] There are extensions to this model, but we liken them more to the other *declarative* approaches rather than process calculi.

multiple components, including the computation cores, various controllers, caches, and even software components like the operating system. The complex interplay between these subsystems and the external physical memory results in similar complexity as in distributed systems.

In order to improve the speed and efficiency of memory operations, modern CPUs – like many communication models – also relax some of the usual guarantees when accessing shared memory. This can lead to "exotic" executions that many programmers would find unexpected, like the reordering of memory operations. The exact way the usual guarantees are relaxed depends on the architecture and is described by a *memory model* [17].

Historically, memory models were not defined explicitly [18]. Chip vendors usually gave a textual description of the intended behavior or the outline of the underlying architecture, and sometimes a few demonstrative examples called *litmus tests* [18]. The need for a more precise description of *what is allowed* in a certain architecture has led to the development of axiomatic memory models and the Cat language, along with a set of tools to *simulate* litmus tests on a given architecture [19]. Soon after, the first software verification tools that handle these axiomatic descriptions as a parameter followed as well [20, 21].

1.3 Goals of this Paper

This work has been born out of the recognition that the modeling and verification of distributed systems modulo communication models, and parallel programs modulo memory models share a large number of similar challenges. According to our observation, the highly advanced and dynamically improving solutions of the shared memory domain have not yet found their way into the distributed systems domain, even though many of the techniques and even the tools themselves would be suitable to solve both kinds of problems. Of course, everybody has at least some intuition of the relationship between those two fields (e.g., a reverse transformation has been shown in [22]), but the point of this paper is to formalize this intuition and to adapt tools from shared-memory to work with message-passing distributed systems. Our solution is called NetworCat, and it includes a minor extension of the Cat language as well as a set of standard primitives to declaratively model the functional aspects of communication models. We hope that our work will facilitate the application of the newest results of the software verification community in the functional verification of distributed applications.

To this end, the paper will introduce the reader to the basics of shared memory modeling and analysis in Section 2; lay the foundations of how to model axiomatic semantics for message-based communication in Section 3; define and present a standard library of axiomatic constraints as primitives to declaratively model the functional aspects of communication models in Section 4; demonstrate the applicability of the concept and the standard library in the context of two common communication models (UDP and TCP) as well as a complex industrial middleware (OMG-DDS) in Section 5; then conclude in Section 6. The paper is accompanied by a small tool[2] to

[2] http://ftsrg.mit.bme.hu/networcat/

$$\begin{array}{l|l} \mathbf{x} := 0, \mathbf{y} := 0 & \\ \mathbf{x} := 1 & i := \mathbf{y} \\ \mathbf{y} := 1 & j := \mathbf{x} \end{array} \quad \text{(2w2r)} \qquad \begin{array}{l|l} \mathbf{x} := 0, \mathbf{y} := 0 & \\ \mathbf{x} := 1 & \mathbf{y} := 1 \\ i := \mathbf{y} & j := \mathbf{x} \end{array} \quad \text{(wrwr)}$$

Fig. 1: Programs showcasing the differences between various memory models

demonstrate how NetworCat can be used with the Herd tool [23]. The demonstrational programs and models used in the context of this work have been published as loadable, read-to-try examples in our proof-of-concept tool.

2 Analysis of Shared Memory Software

In this Section, we introduce the core concepts of memory modeling and their operational and axiomatic semantics, as well as three common families of memory models to showcase the common constraints and rules of the models.

2.1 Memory Models

Consider a simple two-threaded program (2w2r in Figure 1), where one thread writes variables x then y, and the other reads them back in reverse order.

Here, the initial values in memory are depicted in the first line using assignments to variables, while the threads of the program are divided by vertical lines. The order of the instructions on any given thread follows their respective order in the source code of the program.

Assuming no further synchronization among the threads, the intuitive set of possible outcomes for (i, j) include $(0, 0)$ and $(1, 1)$ when either of the two threads runs before the other, and $(0, 1)$ when the second thread starts reading between the two *Write* instructions. We could naively say that $(1, 0)$ is never possible because for that, the *Write* to $\mathbf{y}$ had to have finished before the *Read* from $\mathbf{y}$ did, while delaying the *Write* to $\mathbf{x}$ after the *Read* from $\mathbf{x}$, yet still ordering it before the *Write* to $\mathbf{y}$ based on the instruction order. Therefore, the first instruction should happen *before itself*. It is easy to conclude that such an execution is impossible.

If we look at the above-described execution for $(1, 0)$ on an *execution graph* in Figure 2a, we can see this *happens-before* loop. An execution graph is a directed and labeled graph, where nodes are memory accesses (reads and writes), and edges show *relations* among these accesses. In Figure 2a, the *rf* (read-from[3]) edges denote data flowing from *Write* operations to *Read* operations, and the *po* (program order) edges show the instruction order in the program.[4] In addition, the *fr* (from-read[5]) edge is also added to show that a *Write* has to be ordered after a *Read*, because the *Read* received an older value; hence the write's effect is not yet visible.

If we follow the edges starting from $x := 1$, the loop is clearly visible through the *po*, *rf*, *po* and *fr* edge sequence. As we assume that all these relations denote a

[3] "Read" in past participle, denoting where the value *has been read from*.

[4] We can say an operation is po-*before* another operation (or po-*previous*) when there is a directed path labeled with *po* between the two.

[5] "Read" as a noun denoting the operation, i.e., we know *from the* Read *operation* that we have an implicit ordering constraint.

happens-before relation (i.e., the target instruction (of the relation) happening *after* the source) – which is a transitive property – $x := 1$ should happen before $x := 1$. Based on our reasoning so far, this execution is never observed when running on actual hardware.

However, should we run this small program (also called *litmus test*) on some actual multicore architecture, we may very well find that the outcome $(i, j) = (1, 0)$ will occur. This does not contradict the visualized relationships in Figure 2a, but it seems some of them are not ordering constraints on real hardware. Indeed, the hardware executing the test has an associated *memory model*, which will govern which execution patterns are allowed or forbidden. For example, a *sequentially consistent* memory model (SC) would not allow this execution to happen, as it ensures that all the relations introduced so far are indeed treated as *happens-before* relations and, therefore, should be respected by any valid execution.

Memory models can be approached either based on *operational* semantics or *axiomatic* semantics. The former is useful in terms of understanding *why* memory models are so complex (and why we need them in the first place), and the latter can be efficiently utilized for reasoning about the *observable outcomes* when applying a memory model to a problem. We will now introduce three common families of memory models based on their operational semantics: *sequential consistency*, *total store ordering*, and *weak memory*.

2.1.1 Sequential Consistency (SC)

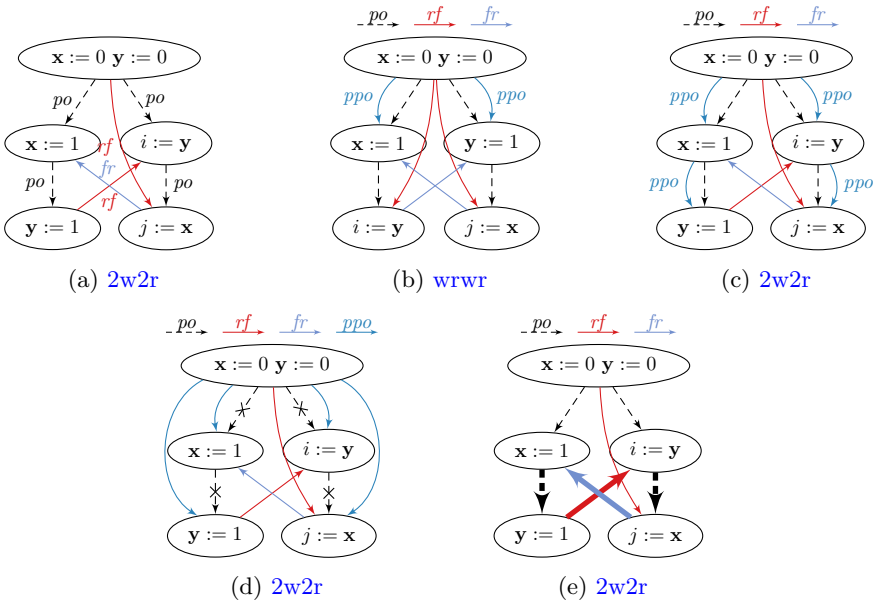
The operational semantics of a memory model describes the steps the hardware-software system will take when presented with an input program such as 2w2r. In this view, the easiest memory model to describe is *sequential consistency* (SC), the strictest memory model when it comes to restricting the number of observable outcomes. In this model, the system will loop through the following steps (let T be the pool of threads in a program, and $PC(t)$ the program counter for thread $t \in T$):

1. Choose a thread $t \in T$, load instruction i at $PC(t)$
2. Load the inputs to instruction i from memory
3. Execute instruction i
4. Write the output of instruction i to memory

While this model is certainly easy to understand from the programmer's point of view, it is very wasteful of CPU cycles. Steps 2 and 4 both have to access the main memory, which is potentially orders of magnitude slower than just using values already in the CPU's register. For loading the data (step 2), caching might be useful, which makes the step significantly shorter (when no cache miss occurs). However, writing to the main memory (step 4) has to invalidate caches in all threads, which is again complicated and slow. Without invalidation, the value is not visible to other threads, and hence a successive load on another core would return an older value.

2.1.2 Total Store Ordering (TSO)

To further improve performance, in addition to caching, the strict rule of waiting for *Write*s to finish can also be relaxed (hence, getting a *relaxed* memory model). A *store*



(a) 2w2r (b) wrwr (c) 2w2r

(d) 2w2r (e) 2w2r

Fig. 2: Execution graphs of programs

buffer can be introduced to the architecture, where step 4 can place its value immediately after execution, and some other mechanism in the architecture will propagate it to caches and eventually to main memory. This will introduce the above-mentioned "anomaly" where other threads will see the value later than with SC. To showcase the impact of this change, see wrwr.

Considering a round-robin scheduling, $\mathbf{x} := 1$ and $\mathbf{y} := 1$ would go first in some order, both putting a new value (1) to a store buffer associated with $\mathbf{x}$ and $\mathbf{y}$, respectively. However, it is not guaranteed that these values are propagated to shared memory before running the next two instructions, $i := \mathbf{y}$ and $j := \mathbf{x}$, which will be able to read the old values (0). This execution is summarized in Figure 2b, with visible *fr* edges to denote *Write*s which should be ordered *later* than respective *Read*s, because at the time of reading the value an older *Write* was still in effect. If all relations denote a *happens-before* notion, this execution should not be observable either, as there is a loop among the edges: $x := 1$ should happen before itself here as well. The solution to this problem is recognizing the *po* edge in itself does not define a total order, as *store buffering* will enable *Write*s to succeed without the global shared memory (and hence, the view of the reading thread) changing.

To show which *po* edges to consider in the ordering by a certain architecture (via its abstraction as a memory model), the *ppo* (preserved program-order) edges are added to the graph. This relation may be seen as a filtered *po* relation, which *relaxes* the strict ordering of sequential consistency.

This behavior is unintuitive for many reasons, and programmers without deep knowledge about the architecture rarely take such behaviors into account, which might

lead to hard-to-find bugs. However, such architectures are very prevalent, with one of the most used examples being the *total store ordering* (TSO) architecture of Intel CPUs. Due to *store buffering*, *po* edges between *Write*s and successive *Read*s on the same thread do not imply a *happens-before* relation, as these *Read*s can be performed before the completion of the *Write* (before the value is written to the main memory). This means that we will have a *ppo* edge for every *po* edge that does *not* go from a *Write* to a successive *Read* – hence the filtered view.

It is important to note that generally, all implemented memory models *intend*[6] to preserve ordering of same-variable memory accesses, so relaxation only matters for foreign-address instruction pairs.

Going back to 2w2r, the execution shown in Figure 2a is still not observable over TSO. Figure 2c showing the *ppo* edges explains this phenomenon – we have a *ppo* edge for all *po* edges, because none of them leads from a *Write* to a successive *Read*. Therefore, the happens-before loop that prevents the realization of this execution still exists.

2.1.3 Weak Memory

Even though TSO already makes unintuitive executions possible, there are even more relaxed (*weaker*) memory models. For example, ARM uses a form of *weak* memory model that allows any non-dependent instructions to be reordered, effectively shrinking the size of *ppo* to only include edges between dependent instructions.

Note that dependency in this context is a generic term to denote any relationship where the output of an instruction influences the behavior of another later on. Such dependencies include *Write*s and loads with the same variable, or loading a value and then loading another using the first result as an address, etc.

Over weak memory, 2w2r will only include *ppo* edges starting in the initial *Write*s (see Figure 2d). Without considering *po* edges between non-initial instructions as *happens-before* relations, the execution resulting in $(1, 0)$ is now *allowed* by the memory model.

2.2 Declarative Semantics

So far, we have explained widely used memory models via their operational semantics (i.e., the steps an architecture will go through when executing an instruction). Real memory models are usually so complex that a full operational specification would be very hard to read. For example, the RISC-V ISA defines the *RVWMO* (RISC-V Weak Memory Ordering) model as the least strict model an implementation might use [24]. As RISC-V permits open-source hardware implementations for its adopters, one may be able to see the source of the entire memory subsystem in such implementations. While such a description could be used to co-simulate with programs and analyze the precise behavior of parallel programs, the complexity of the architectures would make this prohibitively resource-intensive.

[6] One of the main goals of memory modeling is to be able to verify that the hardware complies with these kinds of specifications[19].

A better option is to use the *axiomatic* semantics of memory models to provide a specification. In this semantics, the concept of the shared memory (and auxiliary hardware components, such as caches) is omitted, and rather, memory accesses will be handled as *events*, whose exact behavior and side effects depend on the memory model.

Notice that execution graphs say nothing about the exact scheduling of threads, only denoting (possibly order-enforcing) relations when necessary. Most of these relations *explain* why a specific execution may or may not be observable, but the outcome of the execution itself is determined only by the executed instructions *and* the *rf* edges, which denote *data-flow* (as opposed to inferred ordering constraints only). Therefore, the analysis of a program must focus on what instructions can be executed with what data-flows – while respecting the constraints implied by all the other relations. Therefore, in the axiomatic semantics, *memory models describe constraints on the* rf *relation based on the static and dynamically inferred relations implied by the program.* Static, in this sense, means relations such as *po*, which are directly mapped from elements in the program source.

2.3 The CAT Specification Language

The most popular specification language for axiomatic memory models is Cat [25]. Using *event sets* and *relations* over events, the possible *rf* relations are restricted using *constraints*. Any number of event sets and relations may be defined using operations over other elements or primitives, but there are a few built-in basic event sets and relations that are grounded in the input program's source (the *static* relations mentioned above). Furthermore, deduced elements may be defined using operations over other elements. These elements and operations include the listed items in Figure 3 and Figure 4. Most of these operations (or *derivation rules*, as new event sets or relations may be derived with such operations) are self-explanatory as they stem from set theory. A notable exception may be the *sequence* operation, which has the following definition: $(e_1, e_2) \in a \wedge (e_2, e_3) \in b \implies (e_1, e_3) \in (a; b)$.

So far, memory models have been introduced to solve problems where, given an input program and memory model, the task is to determine what executions are allowed. Notice, however, that some pre-defined elements do not make sense when considering *programs* as inputs – namely, the FW (final writes), *co* (coherence order, i.e., global order of same-variable write events), and *rf* (read-from). These are properties of an *execution*, rather than the input program. This is a deliberate choice of Cat, as the semantics of the language is given for *validating candidate executions*, rather than *generating consistent executions with a memory model.* Candidate executions are execution graphs to be checked for consistency with a memory model during the exploration of possible executions. Then, tools consuming the memory model described using Cat are capable of checking whether the execution is *consistent* (and therefore observable when the program is executed) or *inconsistent* and therefore *forbidden.*

To validate the consistency of candidate executions, Cat allows the declaration of certain kinds of *constraints*. These act on event sets and relations, and may be of the following kinds: *(a)cyclic* (relations), *(ir)reflexive* (relations) and *(non-)empty*

W All *Write* events	∼*a* Complement
IW Initial *Write* events	*a* | *b* Union
R *Read* events	*a* & *b* Intersection
M All memory events (R ∪ W)	*a* \ *b* Difference
F Fence events	*a* * *b* Cartesian product

Fig. 3: Event sets and derivation rules from events sets

	a+ Transitive closure
	*a** Reflexive-transitive closure
po Instruction order	*a^-1* Relation inversion
amo Atomic read-modify-write	*a?* Reflexive closure
id Identity relation	∼*a* Complement
loc Same-variable relation	*a* | *b* Union
int Same-thread relation	*a ; b* Sequence
rf Dataflow (Write-to-Read)	*a* & *b* Intersection
co Coherence order of writes	*a* \ *b* Difference
	dom Domain of a relation
	range Range of a relation

Fig. 4: (Binary) relations over events and derivation rules from relations

<CAT>	::=	<assert>	‖	<let>	‖	<CAT> <CAT>		
<assert>	::=	acyclic <rel>	‖	irreflexive <rel>	‖	empty <rel>		
<rel>	::=	<named-rel>	‖	dom(<rel>)	‖	range(<rel>)	‖	
		<rel>?	‖	<rel> | <rel>	‖	<rel> & <rel>	‖	
		<rel> \ <rel>	‖	<rel>^-1	‖	<rel>+	‖	
		<rel>*	‖	<rel>; <rel>	‖	<evt> * <evt>		
<evt>	::=	<named-evt>	‖	∼ <evt>	‖	<evt> | <evt>	‖	
		<evt> & <evt>	‖	<evt> \ <evt>				
<let>	::=	let <id> = <rel>	‖	let <id> = <evt>				
<named-rel>	::=	po ‖ amo ‖ id ‖ loc ‖ int ‖ rf ‖ co ‖ <id>						
<named-evt>	::=	M ‖ W ‖ R ‖ IW ‖ F ‖ <id>						

Fig. 5: Cat syntax (simplified)

(relations or event sets). For example, to realize *sequential consistency*, the acyclicity of the happens-before relations (*fr*, *po*, *rf*, *co*) can be asserted:

```
acyclic (po | co | rf | fr)
```

When applied to the execution in Figure 2e, the thick cycle consisting of the relations is detected, and therefore this candidate execution is discarded.

The core concrete syntax of Cat is summarized in Figure 5. The full language specification (including language elements not utilized in the current work) and the precise semantics can be found in [25].

2.4 Analysis Tools

For shared memory concurrency, various tools exist that can analyze a program together with a memory model and answer queries on their exhibited properties (e.g., error state reachability). The historically most significant tool is most probably Herd

[17], which enabled rigorous analysis of *litmus test* behavior, thus validating the idea of axiomatic modeling and analysis of memory-model-aware software.

2.4.1 Herd

HERD is a memory model simulator [17]. It expects a memory model specification written in the CAT language [25] and a litmus test.

For a given memory model and litmus test, the question is whether the forbidden behavior is *observable* on the target architecture. To answer this question, HERD will first generate all candidate executions of the litmus test. This is done in an enumerative way: for each primitive relation, every semantically correct combination will be explored [17]. After enumeration, the candidate executions are filtered based on whether they are consistent with the specified memory model. If any consistent execution graph of the litmus test produces the forbidden outcome, the specified behavior is *observable* and the litmus test fails.

The number of *candidate executions* is generally much higher than the number of *consistent executions*, especially for larger programs. However, the goal of HERD is not general program verification but rather architectural verification. Litmus tests are, by definition, small programs, and therefore it is unnecessary to optimize the algorithm in HERD for input size. For anything larger than an ordinary litmus test, HERD will most likely time out while enumerating the candidate executions. This prompted the development of smarter *candidate execution* generation, such as RCMC [26].

2.4.2 RCMC

The novelty of RCMC is its smart exploration algorithm. In each step of its algorithm, RCMC will only generate consistent execution graphs, and no execution graph is ever explored twice. The implemented *stateless* model checking algorithm receives a concurrent C/C++ program with optional assertions, and enumerates all consistent executions as its output. If in any of the execution graphs the assertion is violated, or a non-atomic concurrent access occurs, the tool reports the program as *unsafe* immediately. Note that the memory model is *not* an input, as the C/C++ concurrency model (as formalized in the repaired RC11 memory model [27]) is hard-coded into the algorithm. This significantly reduces the applicability of the tool for custom architectures and potentially yields false positive results.

2.4.3 GenMC

As an improvement to RCMC, GENMC promises to deliver a memory model-aware, stateless model checking algorithm. This enables the verification of software running on custom memory models with an approach very close to that of RCMC. However, the boundedness of the algorithm is still a considerable drawback (as it is with RCMC as well).

2.4.4 Dartagnan

Most of the concerns above are addressed by DARTAGNAN, a bounded model checker that uses memory models as modules [21, 28]. DARTAGNAN expects a concurrent

program and a memory model as inputs, and using the conjunction of SMT-encoded expressions determines whether an unsafe state is reachable within a given bound. To achieve this, Dartagnan unrolls and encodes the concurrent program as an SMT-expression; encodes the unsafe state as another SMT-expression; and encodes the input memory model as an SMT-expression. If the conjunction of the expressions above is *satisfiable*, the unsafe state is *reachable*, and therefore, the concurrent program is *unsafe*.

Dartagnan is a software verification tool, complete with an integration to Smack [29], an LLVM-based program transformation tool that allows Dartagnan to work on formal models rather than source-level programs. The gap between the higher-level LLVM-IR and the ISA of the target architecture is bridged by using *compiler mappings* for translating e.g. memory ordering primitives [21]. This is a conventional procedure [19], but special attention has to be paid to ensure the compiler mappings represent an actual compiler's behavior that might be used to compile the examined program later on.

In comparison with Herd and rcmc, Dartagnan (and its companion tool, Porthos [21]) is not capable of enumerating consistent executions. Even though as a reachability checker, Dartagnan is not expected to provide this feature, it could be useful to provide a way to use the tools embedded into other verification algorithms for handling concurrent parts of an otherwise independent set of threads. In this case, an unsafe state might not only be dependent on the concurrent parts of the program, and therefore Dartagnan could not handle it on its own.

As shown in the sections above, many practically proven tools and algorithms exist to verify shared memory software with weak memory semantics. Our goal in this paper is to bring these tools to distributed systems as well – hopefully without the tools' developers ever needing to modify their source code. Thus, shared memory verification tools could benefit from a wider audience and user base, while the domain of distributed system verification could benefit from this ever-improving litany of analysis methods.

3 Modeling Message-Based Communication in CAT

As discussed in Section 2, the axiomatic semantics for concurrency relies on connecting *Read* events to value-supplying *Write*s via the *rf* relation. This completely eliminates the need to use the state of the shared memory to reason about possible executions of programs, instead focusing on the possible data flow links between instructions. The state of the execution is, therefore, implicitly captured in the graph.

As long as there are clearly defined operations that can be interpreted as *Write* and *Read* events in the sense that one provides data to the other, we can use the techniques developed for concurrent software to model and analyze the functional aspects of other types of parallel systems as well. Note that these techniques do not aim to cover timing and other quantitative aspects – the goal is to formally prove the *correctness* (e.g., fulfillment of some safety properties) of a parallel system with respect to a chosen ruleset governing the interaction of parallel components.

In this paper, we apply these techniques to distributed systems using models of communication with *Receive* and *Send* operations, where *Receive* events will be related with *Send* events to model inter-node communication. While these operations roughly map to *Read*s and *Write*s, there are some fundamental differences in modeling the two kinds of communication models.

3.1 Modeling Send and Receive Semantics in Cat

We assume that participants using the communicational model (i.e., software instances enabled to communicate with each other) are connected to some common network and can either:

1. Address each other (thus being able to *Send* and *Receive* directed messages, possibly via *scoped* or *named* channels), or
2. Address some *scoped* or *named* network resource or shared variable (thus being able to *Publish* and *Subscribe* to *Topics*).

An example of the former might be a point-to-point message-passing model, such as sending IP packets to each other with a channel name that can be used as a filter. An example of the latter might be a publish-subscribe middleware such as Message Queue Telemetry Transport (MQTT) [30] or Data Distribution Service (OMG-DDS) [31]. See Section 5.3 for more detail on the latter.

Most of the differences stem from *Receive*s not being able to always acquire a value, as opposed to *Read*s, where eventual access to a value is always provided (because operationally, it just needs to read a value from memory, which is never inaccessible). Furthermore, no initial implicit *Send* event (analogously to initial *Write* events) should exist in a network setting because the channels start empty. This is important to distinguish because in some communicational models, *Receive* operations will block on an empty channel instead of returning a default value. There is a similar problem with *Send* events, because they may need to get an acknowledgment message from a *Receive* operation (depending on the underlying protocol) – therefore, they might have to wait until another participant becomes ready to communicate. This also invalidates the expectation we had for *Write* operations (i.e., they will always be able to store a value in memory, even with no future potential readers present), because *Send* operations can also block.

For the latter problem (i.e., a *Send* not succeeding if no suitable participant is available), partial support already exists in Cat. An example of such an instruction can be observed in the ARM ISA for *load-exclusive* and *store-exclusive* operations [17, 25]. In that case, depending on the behavior of other threads, a *store* might fail, i.e., not commit anything to memory. We can leverage a similar mechanism for disallowing *Send*s to succeed without a partnering *Read*, should the communicational model we are utilizing require this behavior.

However, the former problems (i.e., a *Receive* not being able to read a value when no suitable *Send*s exist, and disallowing initial *Send*s) require one of two solutions:

1. We forbid reading from the initial *Send* events[7], and we forfeit the implicit expectation of CAT (and verification tools built with CAT in mind) that a *Receive* must have an associated rf-edge in all consistent executions (to allow non-blocking *Receive*s); or
2. We allow reading from the initial *Send* events, but assign the special meaning to such rf-edges that the *Receive* did not actually receive a value from any participant (e.g., used a default value as its output).

As it would be impractical to have multiple interpretations of a language, we do not deviate from the defined semantics of CAT, and opt to use the 2^{nd} option.

With this slightly modified semantics for *Read* and *Write* events we successfully mapped *Send* and *Receive* operations to aspects of conventional concurrent programming. However, as discussed in Section 2, memory models need to have access to information on the *locality* of events, i.e., the variables which *Write*s and *Read*s operate on. In the case of *Send* and *Receive* operations, however, messages may be passed without any explicit definition of locality: if a sender can connect to a receiver, it will be able to send a message. Furthermore, in many cases, a sender will have an intended receiver it tries to send data to, instead of storing a value in memory for all future readers to see. Therefore, the aforementioned *same-variable (loc)* predicate will have to be reinterpreted:

- If a *Send* event *could* in theory supply data to a *Receive* event, their locality is the same.

Consequently:

- If a *Receive* event can receive from multiple *Send* events, the localities of such *Send*s are the same.
- If a *Send* event can supply data to multiple *Receive* events, the localities of such *Receive*s are the same.

In all other cases, the locality of a pair of events must be different.

In this context, the possibility for data transfer means that there could be a situation where the semantics of the communicational model would allow for the two events to connect, but not necessarily in all cases. One such example is when a *Send* using a simple Internet Protocol (IP) based communicational model correctly addresses the participant executing a *Receive* event – if the address is correct, then connection to the receiver is possible, even though an inspection of the software executing these instructions is necessary to determine whether this scenario could actually happen.

In practice, this means that we distinguish logical *topic*s of communication, which, depending on the underlying communicational model, might be communication on a certain port, a message queue, a channel, or even an actual topic in models such as Message Queue Telemetry Transport (MQTT) [30] or Data Distribution Service (OMG-DDS) [31]. These *topic*s will behave as variables would in the case of shared memory local concurrency.

It is also important to note that we need to support different access types of topics (i.e., variables) because a part of the program might use a stricter communicational

[7] This is easy in CAT: $empty\ (IW * R)\ \&\ rf$

model to access one topic, while another part will use a more relaxed one. Strictness, in this sense, is governed not only by the communicational model itself but also by the model's Quality of Service (QoS) settings. This is not unlike atomics in C11 [32], where different memory ordering primitives can be used to ascertain the rules the execution hardware has to enforce. CAT supports *tag*s that can be used in the definition of custom event sets and relations to help distinguish the different levels of strictness – we can also take advantage of this approach by defining tags for communicational models and their respective QoS settings.

3.2 Formal Model of Network Communication in CAT

To give a precise definition to the model of network communication in the context of this work, we provide an instruction set definition for the communicating software as a specialization of LISA, a generic instruction set architecture that can be used with CAT; and mappings from concepts of the generic communication model to concepts of the CAT language:

- One or more software instances (hereinafter *participant*s) are communicating with each other solely via *Send* and *Receive* operations.
- *Send* operations take a value as input, and have no effect on the local execution of the participant. *Send* operations are *Write* instructions from the LISA instruction set tagged with "*send*" and possibly further, communicational-model-specific tags.
- *Receive* operations return a value sent by a *same-location Send* operation. *Receive* operations are *Read* instructions from the LISA instruction set tagged with communicational-model-specific tags.
- The locality of operations is defined by allocating a variable to each of them, respecting the locality rules outlined above.
- Default values to failing *Receive* operations are modeled using initial values of variables.

With this customized instruction set and corresponding CAT set and relation definitions, one can now reason about distributed programs communicating via messages organized in topics. Note that these modifications do not modify the grammar of CAT but rather extend it by providing a library, defining the required primitives. The reference implementation of CAT (in HERD [17]) natively supports all the additions.

4 Modeling Common Patterns of Communication

After introducing the formal foundations of NETWORCAT, we now propose a *standard library of communication patterns* to effectively model real-life communicational models, which should aid the modeling of communications by eliminating the need for boilerplate parts in the model. It is not our goal to handle every edge-case, but rather to create an extendable set of rules that cover a wide range of common patterns. In an effort to accomplish this purpose, first, we identified common phenomena in network communication, then we gathered a list of patterns that various models either allow or forbid. Thereby, representing a communicational model only necessitates selecting

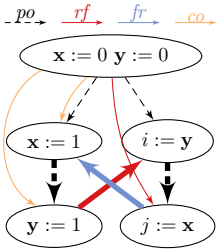


Fig. 6:
Forbidden reordering

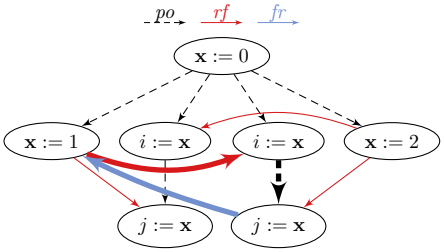


Fig. 7:
Allowed reordering

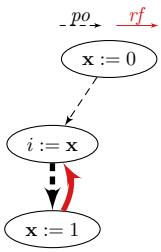


Fig. 8:
Acausal *Read*

patterns in accordance with the model's guarantees and then, if needed, extending the rules to accommodate any special behavior patterns.

For the definition of syntax constructs used throughout this section, see Figure 5 in Section 2.3.

4.1 Patterns for Common Phenomena in Network Communication

When it comes to shared memory communication, the reordering of memory accesses was the main source of complexity. For network communication, however, multiple types of problems can occur:

- Two messages are sent in one order but observed in another by at least one receiver
- A single message is sent, but multiple messages are received with identical contents
- A message is sent, but no one received it

Note that the last two problematic events are normal when we talk about memory accesses but might be unusual and/or unwanted when network communication is used. Based on the most common phenomena in network communication, the list of behavior patterns we chose to model in our standard library is the following:

1. Reordering: does the order of received messages respect the ordering of the respective *Send* operations?
2. Duplication: can multiple receptions by the same participant occur from a single sent message?
3. Reliability: is the reception of a sent message guaranteed?
4. Broadcasting: can multiple participants receive a single sent message?[8]
5. Sending synchronicity: can a participant issue new events before a sent message is received?
6. Blocking reception: can a participant fail and receive a default value, or will all receive instructions block until successful?

[8] In general, a message is *broadcasted* if it is delivered to all participants. In contrast, we characterize *broadcasting* as *if a participant receives a message from a sender, then it must receive all messages from that sender.*

4.1.1 Reordering Messages

The reordering of events is a phenomenon that both concurrent software and distributed systems exhibit.

No reordering

With no reordering, we expect all *rf* edges to conform to a strictly *sequential* execution – which can be achieved using the following memory model constraint:

$$\begin{array}{c} let\ fr\ =\ (rf\hat{}\text{-}1\ ;\ co) \\ acyclic\ (po\ |\ co\ |\ rf\ |\ fr)\ as\ sc \end{array} \qquad \text{(no-reordering.cat)}$$

As demonstrated in Figure 6, global reordering (akin to *sequential consistency*) can be forbidden by disallowing cycles in the subgraph consisting of the program order (*po*), coherence order (*co*), read-from (*rf*) and from-read (*fr*) edges.[9] The candidate execution in Figure 6 contains such a cycle, drawn with thicker edges, and is thus *forbidden* under the no-reordering constraint.

The syntactic element *as sc* is helpful in identifying the constraints in the output of verification tools (here, *sc*).

Locally no reordering

Another option is that if two *Write*s to the same variable happen in a particular order on one participant, then no other participant shall read these values out-of-order. However, if such *Write*s were to happen on different participants, then reading may happen in any order. Thus:

$$\begin{array}{c} let\ fr\text{-}local\ =\ (rf\hat{}\text{-}1\ ;\ (co\ \&\ (po\ |\ IW * W)))\ \backslash id \\ acyclic\ (po\ |\ co\ |\ rf\ |\ fr\text{-}local)\ as\ sc \end{array} \qquad \text{(locally-no-reordering.cat)}$$

The execution in Figure 7 shows a 4-threaded program, where two of the threads *Write* the variables **x**, while two other threads read them twice. While the reader threads may perceive the changes to **x** in any order, it may be required that all readers should agree on one particular order. Therefore, we may not want to see an execution where one thread reads 1 then 2, while the other reads 2 then 1. This candidate execution is shown in Figure 7. Notice the from-read edge: while (no-reordering.cat) would draw that edge in and therefore invalidate the candidate (due to the cycle with thick edges), (locally-no-reordering.cat) will not do so. It would only draw in the from-read(-local) edges (*fr-loc*) towards such *Write*s which are *co*-after some other *Write* on the same participant. Therefore, in Figure 7 there will be no cycles, and the execution is allowed.

[9] As a reminder, the from-read (fr) edge denotes an inferred happens-before relation between a *Read* r and a *Write* w when there is another *Write* w' *co*-before w and there is an *rf* edge from w' to r (i.e., w could not have happened before r, otherwise r would have read the value written by w).

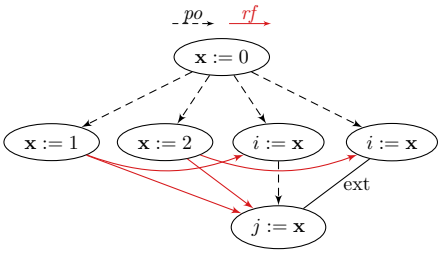


Fig. 9:
Message duplication

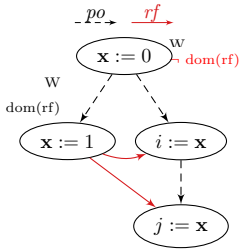


Fig. 10:
Message loss

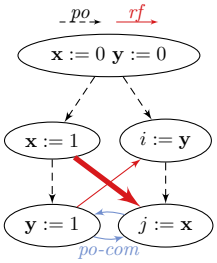


Fig. 11:
Synchronicity

Causality

If a communicational model wants to allow reordering, we still need to pay attention to causality – meaning a previous reception should never read a value published by a *po*-later send. With no other constraint, this can be realized by disallowing cycles in the *po* and *rf* relations:

$$\textit{acyclic}\ (po \mid rf)\ \textit{as causality} \qquad \text{(causality.cat)}$$

The execution in Figure 8 shows a single-threaded candidate execution, where a po-later *Write* event should supply the value received by the previous *Read* event. This constitutes a cycle in the $(po \cup rf)$-subgraph of the execution graph, and therefore the candidate is rejected by the constraint (causality.cat).

Of course, finer control over reordering messages *can* be expressed using Cat – but that problem is well-covered by the conventional memory modeling of concurrent programs [17].

4.1.2 Duplicating Messages

With concurrent software, it is not unusual to have a *Write*'s value passed to multiple *Read*s – if that value is in-memory, any number of *Read*s might receive it. However, with message-based communication, each message is restricted to be received once in most cases (either globally, or by participant). To model global uniqueness using Cat, we can say that *there shall not be such a* Write *that connects to two* Read *events*, by constructing all sequences of the *rf* relation with its inverse (therefore, this path of length 2 would go from a *Read* to a *Write* and then to a *Read*). We need to exclude self-loops from this rule (i.e., the two *Read* events must be different):

$$\textit{empty}\ (rf\hat{}\text{-}1;\ rf) \setminus id\ \textit{as noDup} \qquad \text{(global-no-dup.cat)}$$

This rule can be relaxed by allowing each *participant* to read the value once by also excluding all *Read*-pairs from this constraint that are on different participants (*ext*):

$$\textit{empty}\ (rf\hat{}\text{-}1;\ rf) \setminus id \setminus ext\ \textit{as intNoDup} \qquad \text{(participant-no-dup.cat)}$$

Figure 9 shows these two patterns in detail. Two types of duplication occur in this candidate execution: $x := 1$ supplies values to two *Read*s on the same participant, and $x := 2$ supplies values to two *Read*s on different participants. Applying the constraint (global-no-dup.cat) will flag all pairs of *Read*s with the same rf-origin *Write* as inconsistent with the model, while (participant-no-dup.cat) will also check if the pairs of *Read*s are on the same thread (i.e., they lack an *ext*-edge), allowing external *Read*s to have the same rf-origin *Write*. Note that not all *ext* edges are drawn in Figure 9 for clarity.

If duplication should be allowed, no constraint is necessary – this is the most relaxed model.

4.1.3 Losing Messages

In general, it is not forbidden for messages to be *lost*, i.e., have no associated *rf*-edge. It might be required, however, that all messages must be received by at least one participant: a send operation cannot be considered successful, if no-one received (and acknowledged) it. This requirement can be formalized in the following constraint:

$$\textit{empty } W \setminus \textit{domain(rf) as noSendLost} \qquad \text{(no-loss.cat)}$$

The constraint asserts that no such *Write* should exist that does not have an associated rf-edge, i.e., that is not in the *domain* of the rf-relation. Demonstrating this pattern, Figure 10 contains two *Write*s, the initial *Write* with value 0 and a subsequent *Write* with value 1. As it is shown in the figure, the initial *Write* has no DOM(RF) label because no rf-edge exists for the *Write*. Therefore, the constraint does not hold, and the execution candidate is rejected.

4.1.4 Broadcasting Messages

To further constrain the execution seen in no-loss.cat, it can also be a requirement that *all* other participants must receive a value from a particular *Write*:

$$\begin{gathered}\textit{let rf-int} \;=\; \textit{rf; int} \\ \textit{empty } (W * R) \;\&\; \textit{loc} \;\&\; \textit{ext} \setminus \textit{rf} \setminus \textit{rf-int as allMustReceive}\end{gathered} \qquad \text{(broadcast.cat)}$$

The pattern works the following way. First, we relate all *Write*s to all *Read*s of the same variable on different participants ($(W * R)$ & *loc* & *ext*), then take away all (w, r) that already have an rf-edge between them ($\setminus$ *rf*), as well as those (w, r') where r' is on the same participant as r ($\setminus$ *rf-int*). If the remaining set is not empty, then there is a participant that has at least one *Read* from the variable, but none of these *Read*s read from the *Write*, i.e., the broadcast did not reach this participant.

Instead of constraining the *Write* events to have associated rf-edges going to all participants, the constraint asserts that there shall be no *Read* event with no edge to a *Write* event, without at least one other *Read* event on the same participant reading from the *Write*. This rephrasing of the constraint is useful to prevent unnecessary candidate rejections if there are other participants that do not attempt to read. As

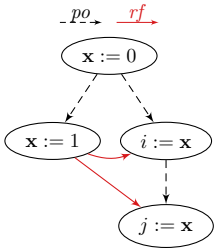


Fig. 12:
Blocking *Read*

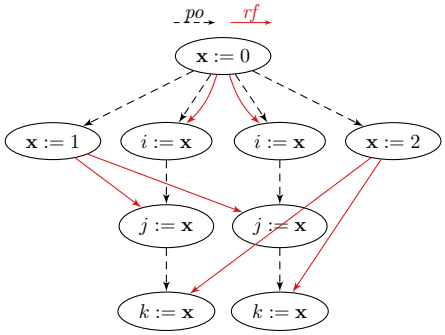


Fig. 13:
Broadcasting write

an example, Figure 13 shows a consistent execution with the pattern, where each of the *Write*s supply a value to each reading participant, without needing to also supply values to non-reading participants (such as the other writing participant).

4.1.5 Sending Synchronously

Some communicational models require all *send* operations to find all its respective *receive* operation(s) before proceeding with execution. This is called *synchronous* messaging, while *asynchronous* refers to the case where the sender can proceed independently from whether the receivers receive the message or not. Normally, concurrent programs are asynchronous and therefore no extra constraint has to be placed over such executions, but synchronicity requires the use of the following construct:

$$\begin{aligned} \mathit{let}\ \mathit{fr} &= (\mathit{rf}\hat{}\text{-}1\ ;\ \mathit{co}) \setminus \mathit{id} \\ \mathit{let}\ \mathit{po\text{-}com} &= \mathit{po} \mid \mathit{co} \mid \mathit{rf} \mid \mathit{fr} \\ \mathit{empty}\ \mathit{rf} \,\&\, (\mathit{po\text{-}com}\ ;\ \mathit{po\text{-}com}+)\ \mathit{as}\ \mathit{allRWCoincide} \end{aligned} \qquad \text{(synchronous.cat)}$$

Consider the execution candidate in Figure 11. The candidate shows an inconsistent execution of the program given synchronicity because the program may not advance after any of the *Write*s before their respective *Read*s finished, thus there exists an ordering relation (*po-com*) between the reads in both directions. This constitutes a happens-before loop in the graph, making the execution forbidden. The pattern detects the existence of alternative paths to rf-edges in the happens-before subgraph of the program because that signals that events exist which are after the *Write*, but before the *Read* (thus violating synchronicity). In the example, the thick rf edge has an alternative path over $y := 1$ and $i := y$, meaning these events were ordered between the supposedly synchronous rf edge.

4.1.6 Blocking Reception

Normally, a *Read* from memory could never fail, as the value in the memory is always established. However, in the case of network communication, it is possible to receive

nothing as the result of an attempted reception. In this case, the implementation might *block* and wait for a valid value, which is handled implicitly, without modeling any timing information (and therefore arbitrary time differences are allowed among any two events). It also might just fail and continue execution normally (with possibly a random value as their received payload, or a default value as placeholder). For the blocking case, the following constraint can be used to disallow reading from the (default) initial value:

$$empty\ rf\ \&(IW * R)\ as\ everyReadReads \qquad \text{(blocking.cat)}$$

The execution in Figure 12 shows the only consistent execution of the program with blocking *Read*s, as neither of the *Read*s receive their values from the initial *Write* events.

4.2 Generalizing the Standard Model

As mentioned in Section 3, the standard model could include topics, QoS settings, etc.. As discussed, each segment of the network under different QoS settings can be tagged accordingly.

Notice that the modeled patterns above use the *rf*, *po*, etc. relations directly, even though different QoS settings might necessitate these rules to be applied differently. To solve this, all rules above are placed inside procedures [25], which receive these relations indirectly as parameters. For example, the rule (no-loss.cat) will look like this:

$$\begin{aligned} &procedure\ no\text{-}loss(local\_W,\ local\_rf)\ = \\ &\ empty\ local\_W \setminus domain(local\_rf)\ as\ noSendLost \\ &end \end{aligned} \qquad \text{(no-loss.cat)}$$

Thus, if only certain *Send*s should have delivery guarantees, the filtered set of *Write*s (e.g., via a "*no-loss*" tag) can be passed to this procedure. This greatly reduces the complexity of models employing several different levels of strictness on ordering.

4.3 Compositions of Patterns

It is natural to have doubts about the composability of the patterns above. They all constrain similar yet distinct behaviors, and using them side-by-side feels like asking for unintentional side effects. However, each pattern constrains the executions *declaratively*, not *operationally* – there is not a pipeline of filters that must be applied on the executions, but rather, given any candidate execution, we can check any number of patterns for consistency.

However, it is entirely possible to compose unsatisfiable models. If two patterns (given a program) contradict each other, it is possible that no execution will be found consistent with all constraints. However, that is a sign of a misconfigured formal model rather than a flow in the approach.

One example of such a problem may be combining *blocking* reads with *non-duplicating* writes, and having fewer reads than writes in the program's execution.

```
UDP

include "causality.cat"
call causality(po, rf)

include "blocking.cat"
call blocking(R, rf)
```

Fig. 14: udp.cat

```
send(x, 1) | i = receive(x)
send(x, 2) | send(x, i)
           | j = receive(x)
```

Fig. 15: udp.litmus

That will inevitably be unsolvable, as the two patterns cannot be satisfied at the same time, given the number and nature of communication events – this means that the execution prescribed by the program will halt earlier than expected before a superfluous *Read* could advance the program's flow.

For examples on composability, see Section 5, where we show a memory model for UDP [33], TCP [34] and DDS [31].

5 Evaluation of the Applicability of the Approach

To demonstrate the applicability of the presented approach, we set out to answer two research questions:

RQ1 Can we adapt the approach to communicational models employed by *message-passing networked systems*?
RQ2 Can we adapt the approach to *industrially used, complex, configurable, distributed-memory-based* communicational models?

To answer these questions, we model two simple message-passing examples, UDP and TCP, as well as a significantly more complex communicational model, OMG-DDS [31].

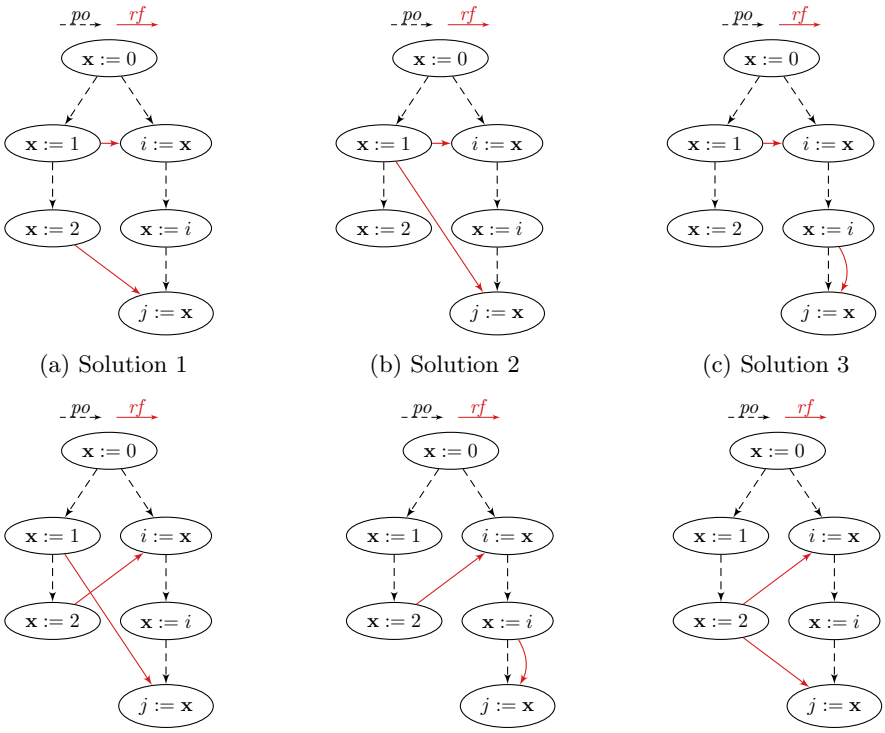
5.1 User Datagram Protocol (UDP)

UDP [33] is a simple, best-effort point-to-point or point-to-multipoint communication protocol. It allows reordering, duplication, and message loss. We assume that the content of the messages are preserved during transmission (or changes are detected by some error-correction code and the affected message is dropped, resulting in message loss). In this example, we opted to demonstrate the point-to-point version for simplicity.

Based on its semantics, a model for UDP needs the following patterns:

- Reordering, but causality
- Duplication
- Loss
- No broadcasting
- Asynchronous
- Reception is blocking

This selection of patterns is realized by the memory model in Figure 14. To showcase the properties above, we would like to generate executions of the litmus test in



(a) Solution 1 (b) Solution 2 (c) Solution 3

(d) Solution 4 (e) Solution 5 (f) Solution 6

Fig. 16: Solutions to Figure 15 over Figure 14

Figure 15. The observed executions are presented in Figure 16. The executions show that the memory model works as intended:

- Solution 2 shows *duplication* and *loss*
- Solution 4 shows *reordering* and *asynchronous*-ness
- No solution has been generated that contradicts *causality*, even though the second participant could have read from its own future
- No solution has been generated that contradicts the *blocking* effect, as no *Read* received its data from an initial send

5.2 Transmission Control Protocol (TCP)

In contrast to UDP, TCP [34] is a reliable, mainly point-to-point communication protocol. It guarantees that all sent messages are delivered without reordering or duplication. Therefore, it needs to abide by the following patterns:

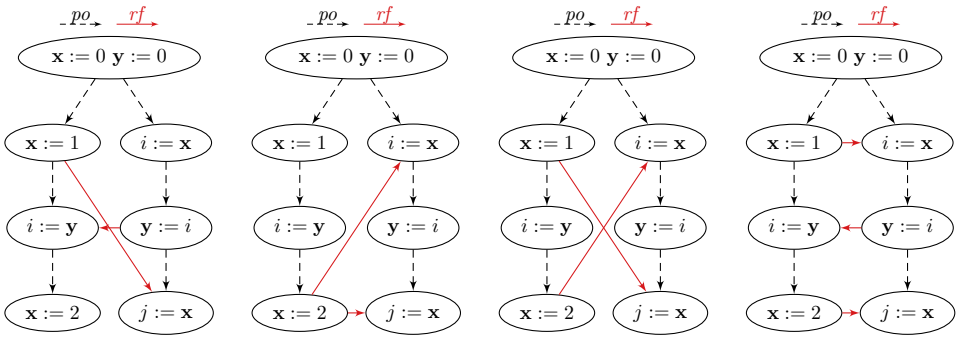
- No reordering
- No duplication
- No loss
- No broadcasting
- Synchronous
- Reception is blocking

```
TCP

include "no-reordering.cat"
call no-reordering(rf, co, id, po)

include "global-no-dup.cat"
call global-no-dup(rf, id)

include "no-loss.cat"
call no-loss(W, rf)

include "synchronous.cat"
call synchronous(rf, co, id, po)

include "blocking.cat"
call blocking(R, rf)
```

Fig. 17: TCP memory model

```
send(x, 1)      | i = receive(x)
i = receive(y) | send(y, i)
send(x, 2)      | j = receive(x)
```

Fig. 18: TCP Litmus test



(a) Forbidden 1 (b) Forbidden 2 (c) Forbidden 3 (d) Only allowed

Fig. 19: Some potential solutions to Figure 18

The memory model for TCP can be seen in Figure 17.

And the representative litmus test (We did not use the same litmus test (Figure 15), as the mismatched number of send and receive events would make it impossible to generate any solutions) can be seen in Figure 18

Notice that only a single execution was generated (seen in Figure 19d): this is because the strict rules of TCP disallow any other execution. See Figure 19 for all the solutions that were eliminated by its rules.

Consider the forbidden execution in Figure 19a. It would violate a number of patterns: it loses the value of the last *Write* (*loss*), there is a *Read* that received no value (*non-blocking*), and it is not possible that the two pairs of *Read*-*Write* events were executed synchronously (*asynchronous*). In addition, Figure 19b shows *duplication*, and Figure 19c shows *reordering* of events. This shows that the litmus test *could* produce forbidden results, but the constructed memory model (rightly) eliminates them.

5.3 Data Distribution Service (DDS)

Data Distribution Service (DDS) [31] is a middleware specifically designed to facilitate data exchange between distributed systems via *samples* (messages). It is used extensively across various domains including defense, aerospace, healthcare, and industrial automation due to its capabilities in supporting real-time communication, reliability, scalability, and interoperability.

DDS adopts a publish-subscribe communication model, where data producers (writers) and data consumers (readers) can exchange data without requiring direct knowledge of one another. This decoupling allows for flexible and dynamic data sharing. Additionally, DDS provides Quality of Service (QoS) mechanisms that enable users to configure parameters like reliability, message durability, resource allocation, and security according to the specific requirements of their applications.

It is worth to note that in some aspects, DDS is closer to shared-memory systems than message-based models, as client applications will generally read and write variables transparently on different participants, using local caches, and it is the job of the middleware to ensure cache synchronization among participants. This makes modeling the policies of DDS more in-line with the concepts of Cat than, e.g., TCP or UDP in Section 4.

DDS defines the concept of domains, which serve as logical groupings of DDS applications that can communicate with each other through DDS. Furthermore, an important feature of DDS is its support for partitions. Partitions allow data to be filtered based on content and associated metadata. This enables logical segregation of data within a domain, enabling different groups of readers to subscribe to specific subsets of data based on their interests. Only readers and writers that are *compatible* (i.e., they are in the same domain and have non-excluding partition filters) may communicate.

In the context of this work, we will focus on modeling a subset of the QoS policy options for DDS that directly impact the safety of a user program. These are the following:

- **History:** Specifies whether to keep all samples accessible for readers (*keep-all*), or the last N (*keep-last*). For the value of N, see *DurabilityService*.
- **DurabilityService:** Specifies the history depth for keeping samples.
- **Ownership:** The *exclusive* setting ensures that only one writer at a time is allowed to modify the data (akin to a *mutex*). The *shared* setting allows all publishers to publish data, even simultaneously.
- **Presentation:** Specifies how changes to data instances are presented to subscribing applications. The setting *ordered_access* can make sample propagation always respect their creation order.
- **Partition:** Enables a logical separation of data within a domain. Only compatible Writers and Readers may communicate.
- **Reliability:** Ensures that data sent by a writer is reliably delivered to its intended readers. DDS supports two reliability modes: *reliable* and *best-effort*.
- **Durability:** Allows data to persist even after the writer has gone offline. DDS supports two durability modes: volatile and transient.

- **DestinationOrder:** Specifies the order in which data is delivered to readers, based on either the source or the reception timestamp of the message (*by_reception_timestamp*, *by_source_timestamp*).

Note that we discarded certain resource-related (e.g., *ResourceLimits*) and timing-related QoS settings (e.g., *Deadline*) because they are out-of-scope for our approach. We aim to use the models for *qualitative* analysis rather than *quantitative* analysis, meaning we can reason about all possible timing- and resource-allocations at once, independent of their exact constraints. Furthermore, some QoS settings have no direct effect on the delivery of messages (e.g., *UserData*), and are thus excluded.

5.3.1 Formal System Model

Systems utilizing DDS for communication may be very heterogeneous in terms of programming languages, operating systems, architectures, and runtimes. To establish exactly what common features this approach supports, we create a simplified version of a system communicating via DDS. We opted to use *bell* files for this specification, which can serve as a preamble for Cat [25].

First, we must define the possible values for the QoS settings. This may be done with *enum*s the way shown in Figure 20 (each entry is given in the order of increasing strictness, using the syntax of Cat [25]).

We also need to support the partitioning of the data space into *domains* and *partitions*, so that *Write*s and reads may only communicate if they are in the same domain, and compatible partitions.

We expect the user to provide the layout of the data space using two special relations, `compatible_partition` and `same_domain`. These must relate compatible memory accesses with each other, so that the relation in Figure 21 will include all pairs of accesses that are compatible.

We also expect the enumerations `domains` and `partitions` to be defined. Using these and the enumerations defined above, we can specify the blueprint of the instructions available for use. These are shown in Figure 22.

It is worth to note that some QoS settings are applied to Topics, DataWriters, DataReaders, Publishers, Subscribers, and DomainParticipants. However, in Cat, we only get the option to add tags to *instructions*, therefore we must encode all settings at the level of instructions. For example, if a Topic has a certain QoS setting, then all *Write* and read events to that topic will need to include the QoS setting.

Note that instructions tagged with a certain value of an enumeration will be available in the corresponding capitalized event set (e.g., `'shared` → `SHARED`).

5.3.2 Assembling the Memory Model

To create the memory model of DDS (as formalized in Section 5.3.1), we need to apply constraints on the tagged instructions. We make use of the composability of the rules (as most QoS settings are independent of each other), and define the model aspect by aspect.

```
enum history           =     'keeplast     ||      'keepall
enum ownership         =      'shared      ||     'exclusive
enum presentation      =  'orderedaccess   ||  'noorderedaccess
enum reliability       =     'reliable     ||    'besteffort
enum durability        =     'volatile     ||     'transient
enum destinationorder  =    'reception     ||      'source
```

Fig. 20: DDS QoS options

```
let compatible_rf = (compatible_partition & same_domain)
empty rf \ compatible_rf as domOrPartIncompatible
```

Fig. 21: Compatibility relation

```
W[domains,partitions,history,ownership,presentation,
    reliability,durability,destinationorder]
R[domains,partitions,history,ownership,presentation,
    reliability,durability,destinationorder]
```

Fig. 22: Instructions

```
let EW = EXCLUSIVE & dom(rf) \IW
empty (EW * EW) & loc & ext
```

Fig. 26: Ownership

```
include "no-reordering.cat"
call no-reordering(
  rf & (ORDEREDACCESS*ORDEREDACCESS),
  co & (ORDEREDACCESS*ORDEREDACCESS))
```

Fig. 27: Presentation

```
include "locally-no-reordering.cat"
call locally-no-reordering(rf, co)
```

Fig. 23: Baseline

```
include "no-reordering.cat"
call no-reordering(
  rf & (KEEPLAST * KEEPLAST),
  co & (KEEPLAST * KEEPLAST))
```

Fig. 24: History

```
include "no-reordering.cat"
let co3 = co; co; co
call no-reordering(
  rf & (KEEPLAST * KEEPLAST),
  co3 & (KEEPLAST * KEEPLAST))
```

Fig. 25: DurabilityService

```
include "broadcast.cat"
call broadcast(
  W & RELIABLE,
  R & RELIABLE)
```

Fig. 28: Reliability

```
include "synchronous.cat"
call synchronous(
  rf & (VOLATILE * VOLATILE))
```

Fig. 29: Durability

```
include "no-reordering.cat"
call no-reordering(
  rf & (SOURCE * SOURCE),
  co & (SOURCE * SOURCE))
```

Fig. 30: DestinationOrder

Baseline model

There are some QoS-independent guarantees that DDS provides its users. Most importantly, out-of-order messages sent by the same participant will be dropped independent of any ordering constraints (i.e., older values will not "overwrite" later values), and read instructions need to completely finish before execution may continue (i.e., there is no reordering of read instructions). These two constraints may be formalized in the memory model the way shown in Figure 23.

Note that some elements of the list of patterns in Section 4 are not directly relevant to DDS, independent of QoS settings:

- *Duplication* is always allowed because (akin to shared-memory systems), a reader will always be able to read at least the last submitted sample.
- *Blocking* reception is not usually allowed because the semantics of a receiver resemble shared-memory *reading* instead of actual message reception, thus a value is always retrieved. In the initial case when no previous samples have been submitted by a compatible writer, the *Read* fails – i.e., either a safe default is returned, or an exception is raised, rendering the instruction unsuccessful, thus not included in the (candidate) execution.

History and DurabilityService

History can either keep all samples for later use (therefore any read may read from any write), or only the last N samples (given by the value given to DurabilityService).

If any read may read from any *Write*, we need not place any additional constraints on the executions, as this is the most relaxed model. For KEEP_LAST, however, we need to make sure that older *Write*s may not write to newer reads. This coincides with the effect of the no-reordering.cat rule because if a newer *Write* is observed, no older *Write* shall be accessible (see Figure 24).

For DurabilityService values larger than 1, we need to use a longer *co*-chain, such as the one in Figure 25 (for $N = 3$).

Ownership

By default, topics are *shared*, i.e., multiple writers may publish samples at the same time. If, however, the *exclusive* setting is used instead, the model needs to constrain the executions such that no two foreign *Write*s may succeed (see Figure 26).

Notice that we must allow initial *Write*s to succeed, as those *Write*s are inherently external to all other *Write*s.

Presentation

If *ordered_access* is set, the instructions shall not be reorderered. This is implemented in Figure 27.

Reliability

If samples are sent with the setting *reliable*, no potential compatible readers may ignore the *Write* – therefore, the rule (broadcast.cat) may be used, shown in Figure 28.

Durability

By default, topics are *transient*, i.e., reads may receive a value from a writer that is no longer active. If the *volatile* setting is used instead, sample reception is *synchronous*, as only currently active *Write*s may interact with *Read*s, meaning the Reader and Writer both need to be active at the same time. This rule is implemented in Figure 29.

DestinationOrder

Depending on the ordering constraint, samples are either ordered by their reception timestamp (by default), or their source timestamp, which results in a global *no-reordering* constraint in Figure 30.

5.4 Summary of the Applicability

As demonstrated by the examples for UDP (Section 5.1), TCP (Section 5.2), and DDS (Section 5.3), the proposed approach of modeling network-based communication with the toolset (namely, Cat [25]) for shared- and weak-memory systems works well for real-life communicational models and QoS settings. While complex communicational models will inevitably include settings not covered in Section 4 as part of the standard library, these situations are easily overcome, such as in Figure 26.

5.5 Threats to Validity

As we do not perform a performance-based evaluation of the proposed approach (because the effect of our contributions is not measurable with a comparative evaluation), we only discuss the relevant *internal* and *construct validity* of the evaluation above. We chose the three examples (UDP, TCP and OMG-DDS) as three vastly different models, hopefully covering most aspects of relevant communicational models. We deliberately chose both message-passing (UDP, TCP) and shared-variable-based models (OMG-DDS), as we set out to show the applicability on simple models far removed from the concept of shared memory (UDP, TCP), and complex models more aligned with its concept (OMG-DDS). We could not test complex models that are not aligned with the shared-memory concept, as to the best of our knowledge, no such widely used solution exists. We have shown that the patterns introduced in Section 4 are useful in modeling real-life models, but the completeness of this pattern-collection has not been evaluated. Furthermore, the proof-of-concept tool [23], while able to handle litmus-test-sized examples for all models in the evaluation, has not been thoroughly tested with bigger, more realistic example programs. We plan to continue our research bridging the two fields together, and will hopefully rectify these shortcomings when future research enables us to pursue them. However, we are confident that despite these limitations, the research questions posed in this paper could both be answered positively:

RQ1 We can adapt the approach to communicational models employed by *message-passing networked systems.*
RQ2 We can adapt the approach to *industrially used, complex, configurable, distributed-memory-based* communicational models.

6 Conclusion and Future Work

In this paper, we introduced an approach that bridges the fields of distributed systems and weak-memory concurrency. Using the presented techniques, the qualitative, functional verification of networked systems is opened up for tools conventionally used for shared-memory concurrency, therefore furthering both fields:

- distributed systems by adding a set of new tools to their verification portfolio; and
- shared-memory concurrency verification by adding a new domain, with potential for new optimizations to existing approaches.

Furthermore, it widens the respective visibility of the fields by researchers and users of both.

While the proposed approach works well for the subset of QoS settings of complex communicational models that are directly responsible for the qualitative *safety* of a system (see Section 5.3), further work is necessary to extend the presented technique to also support *quantitative* metrics, possibly related to timing- and resource-guarantees of these communicational models. Furthermore, a comparative evaluation of verification tools is necessary to show that they perform well over the presumably different characteristics of models sourced from distributed systems as well.

However, the goal of this paper was not to introduce a single tool or algorithm that solves the problem of distributed system verification. Rather, we set out to *unify* efforts in two separate domains: analysis of software with weak memory semantics and analysis of distributed message-passing systems. We believe that the ongoing development of verification techniques, which have already benefited the analysis of weak memory concurrency, may be of use in the field of distributed system verification. Therefore, we have introduced an *adaptation framework*, using the specification language Cat, for a seamless transformation among these domains.

To the best of our knowledge, similar work has not yet been performed, and other efforts – e.g., those introduced in Section 1.1 – all rely on purpose-built modeling frameworks such as process calculi. This constrains the audience (and hence limits the development effort) to such solutions. We are confident that a transformation-based approach is superior not because of its immediate usability or performance (which our proof-of-concept implementation [23] almost certainly lacks) but because of the opportunities it enables. In the broad field of systems verification, such approaches have seen widespread praise from the community, e.g., in the case of bridging hardware- and software verification by transforming C programs to BTOR2 circuits [35], or vice versa [36], or transforming Constrained Horn Clause (CHC) logic problems to C programs [37].

We believe that the best-case outcome of our work is having distributed system analysis research – enabled via our transformation framework – join the efforts of shared memory concurrency verification. Therefore, verification tools could receive additional development effort; research results could receive additional attention, and – hopefully – previously unsolvable problems could finally receive solutions.

In the future, we plan to explore the requirements of safety-critical distributed systems in greater detail to show how our proposed approach can be adapted to work with models using various communication models. We also plan to look at the scalability of our solution to show that real-world examples can successfully be mapped and verified using the techniques presented in this paper. Furthermore, we plan to start working on tool support for creating the memory models automatically from communication model descriptions, such as OMG-DDS configuration files.

This research was partially funded by the ÚNKP-24-3 New National Excellence Program of the Ministry of Innovation and Technology, from the National Research, Development and Innovation Fund of Hungary (grant no. ÚNKP-24-3-BME-213). The Doctoral Excellence Fellowship Programme (DCEP) is funded by the National Research Development and Innovation Fund of the Ministry of Culture and Innovation and the Budapest University of Technology and Economics, under a grant agreement with the National Research, Development and Innovation Office. This work was partially supported by the National Research, Development and Innovation Fund of Hungary, financed under the 2022-1.2.4-EUREKA-2023-00013 scheme.

References

[1] Loo, B.T.: The design and implementation of declarative networks. PhD thesis, University of California at Berkeley (2006)

[2] Guerraoui, R., Yabandeh, M.: Model checking a networked system without the network. NSDI'11: Proceedings of the 8th USENIX conference on Networked systems design and implementation (2011) https://doi.org/10.5555/1972457.1972481

[3] Lobo, J., Ma, J., Russo, A., Le, F.: Declarative distributed computing. Correct Reasoning, 454–470 (2012) https://doi.org/10.1007/978-3-642-30743-0_31

[4] Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. Communications of the ACM **52**(11), 87–95 (2009) https://doi.org/10.1145/1592761.1592785

[5] MA, J., LE, F., WOOD, D., RUSSO, A., LOBO, J.: A declarative approach to distributed computing: Specification, execution and analysis. Theory and Practice of Logic Programming **13**(4–5), 815–830 (2013) https://doi.org/10.1017/s1471068413000513

[6] Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco, CA, United States (1996)

[7] Hoare, C.A.R.: Communicating Sequential Processes. Commun. ACM **21**(8), 666–677 (1978) https://doi.org/10.1145/359576.359585

[8] Milner, R.: A Calculus of Communicating Systems. Lecture Notes in Computer Science, vol. 92. Springer, Berlin, Germany (1980). https://doi.org/10.1007/3-540-10235-3

[9] Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, I. Inf. Comput. **100**(1), 1–40 (1992) https://doi.org/10.1016/0890-5401(92)90008-4

[10] Wang, A., Basu, P., Loo, B.T., Sokolsky, O.: Declarative network verification. Practical Aspects of Declarative Languages **5418**, 61–75 (2008) https://doi.org/10.1007/978-3-540-92995-6_5

[11] Kapus, T.: Improved Formal Verification of SDN-Based Firewalls by Using TLA$^+$. IEEE Access **11**, 107126–107134 (2023) https://doi.org/10.1109/ACCESS.2023.3320050

[12] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: Miller, E.L., Hand, S. (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015, pp. 1–17.

ACM, Monterey, CA (2015). https://doi.org/10.1145/2815400.2815428

[13] Rahli, V., Guaspari, D., Bickford, M., Constable, R.L.: Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **72** (2015) https://doi.org/10.14279/TUJ.ECEASST.72.1013

[14] Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, pp. 357–368. ACM, Portland, OR, USA (2015). https://doi.org/10.1145/2737924.2737958

[15] Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. Proc. ACM Program. Lang. **2**(POPL), 28–12830 (2018) https://doi.org/10.1145/3158116

[16] Mogk, R., Drechsler, J., Salvaneschi, G., Mezini, M.: A fault-tolerant programming model for Distributed Interactive Applications. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–29 (2019) https://doi.org/10.1145/3360570

[17] Alglave, J., Maranget, L., Tautschnig, M.: Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. ACM Trans. Program. Lang. Syst. **36**(2), 7–1774 (2014) https://doi.org/10.1145/2627752

[18] Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and Litmus tests. In: Cohen, A., Vechev, M.T. (eds.) 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pp. 467–481. ACM, Barcelona, Spain (2017). https://doi.org/10.1145/3062341.3062353

[19] Trippel, C., Manerkar, Y.A., Lustig, D., *et al.*: TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In: Chen, Y., Temam, O., Carter, J. (eds.) 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, pp. 119–133. ACM, Xi'an, China (2017). https://doi.org/10.1145/3037697.3037719

[20] Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: McKinley, K.S., Fisher, K. (eds.) Conference on Programming Language Design and Implementation, PLDI 2019, pp. 96–110. ACM, Phoenix, Arizona, United States (2019). https://doi.org/10.1145/3314221.3314609

[21] León, H.P., Furbach, F., Heljanko, K., *et al.*: BMC with Memory Models as Modules. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, pp. 1–9. IEEE, Austin, TX, USA (2018).

https://doi.org/10.23919/FMCAD.2018.8603021

[22] Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1), 124–142 (1995) https://doi.org/10.1145/200836.200869

[23] Bajczi, L., Molnár, V.: NetworCat: Axiomatic Analysis of Distributed Systems. https://doi.org/10.5281/zenodo.8147152 . Budapest University of Technology and Economics

[24] Waterman, A., Asanović, K.: The RISC-V Instruction Set Manual Version 2.2 (2017). https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[25] Alglave, J., Cousot, P., Maranget, L.: Syntax and semantics of the weak consistency model specification language cat. CoRR **abs/1608.07531** (2016) arXiv:1608.07531

[26] Kokologiannakis, M., Lahav, O., Sagonas, K., *et al.*: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. **2**(POPL), 17–11732 (2018) https://doi.org/10.1145/3158105

[27] Lahav, O., Vafeiadis, V., Kang, J., *et al.*: Repairing Sequential Consistency in C/C++11. In: 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2017, pp. 618–632. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062352

[28] Gavrilenko, N., León, H.P., Furbach, F., *et al.*: BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019. Lecture Notes in Computer Science, vol. 11561, pp. 355–365. Springer, New York, NY, USA (2019). https://doi.org/10.1007/978-3-030-25540-4_19

[29] Rakamaric, Z., Emmi, M.: SMACK: Decoupling Source Language Details from Verifier Implementations. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014. Lecture Notes in Computer Science, vol. 8559, pp. 106–113. Springer, Vienna, Austria (2014). https://doi.org/10.1007/978-3-319-08867-9_7

[30] MQTT Version 5.0. Technical report, OASIS Standard (March 2019). https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[31] OMG Data Distribution Service (DDS) Version 1.4. Technical report, Object Management Group (April 2015). https://www.omg.org/spec/DDS/1.4/PDF

[32] Programming languages — C. International standard, International Organization for Standardization, International Electrotechnical Commission (December 2010). https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf

[33] User Datagram Protocol. RFC Editor (1980). https://doi.org/10.17487/RFC0768 . https://www.rfc-editor.org/info/rfc768

[34] Eddy, W.: Transmission Control Protocol (TCP). RFC Editor (2022). https://doi.org/10.17487/RFC9293 . https://www.rfc-editor.org/info/rfc9293

[35] Chien, P.-C., Lee, N.-Z.: CPV: A Circuit-Based Program Verifier. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 365–370. Springer, Cham (2024)

[36] Beyer, D., Chien, P.-C., Lee, N.-Z.: Bridging Hardware and Software Analysis with Btor2C: A Word-Level-Circuit-to-C Translator. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 152–172. Springer, Cham (2023)

[37] Bajczi, L., Molnár, V.: Solving Constrained Horn Clauses as C Programs with CHC2C. In: Neele, T., Wijs, A. (eds.) Model Checking Software: 30th International Symposium, SPIN 2024. Lecture Notes in Computer Science. Springer, Luxembourg City, Luxembourg (2025)