Enabling Bounded Verification of Doubly-Unbounded Distributed Agreement-Based Systems via Bounded Regions

CHRISTOPHER WAGNER, Purdue University, USA
NOURALDIN JABER, Purdue University, USA
ROOPSHA SAMANTA, Purdue University, USA

The ubiquity of distributed agreement protocols, such as consensus, has galvanized interest in verification of such protocols *as well as* applications built on top of them. The complexity and unboundedness of such systems, however, makes their verification onerous in general, and, particularly prohibitive for full automation. An exciting, recent breakthrough reveals that, through careful modeling, it becomes possible to reduce verification of interesting distributed agreement-based (DAB) systems, that are unbounded in the number of processes, to model checking of small, finite-state systems.

It is an open question if such reductions are also possible for DAB systems that are *doubly-unbounded*, in particular, DAB systems that additionally have unbounded data domains. We answer this question in the affirmative in this work thereby broadening the class of DAB systems which can be automatically and efficiently verified. We present a novel reduction which leverages *value symmetry* and a new notion of *data saturation* to reduce verification of doubly-unbounded DAB systems to model checking of small, finite-state systems. We develop a tool, Venus, that can efficiently verify sophisticated DAB system models such as the arbitration mechanism for a consortium blockchain, a distributed register, and a simple key-value store.

CCS Concepts: • **Theory of computation** → **Verification by model checking**; **Automated reasoning**.

Additional Key Words and Phrases: Layered Verification, Data Saturation, Reduction

ACM Reference Format:
Christopher Wagner, Nouraldin Jaber, and Roopsha Samanta. 2023. Enabling Bounded Verification of Doubly-Unbounded Distributed Agreement-Based Systems via Bounded Regions. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 81 (April 2023), 29 pages. https://doi.org/10.1145/3586033

1 INTRODUCTION

A recent breakthrough in formal reasoning about distributed systems builds on the modularity inherent in their design to enable *modularity in their verification* [Griffin et al. 2020; Jaber et al. 2021; Sergey et al. 2017]. The central approach incorporates *abstractions* of common core protocols, such as distributed consensus, into verification of applications *built on top of* such core protocols. This approach inspires an interesting epiphany: modular models of distributed systems based on protocol abstractions may permit *fully-automated verification à la model checking*, even when their monolithic and *intricate* counterparts do not [Jaber et al. 2020, 2021]. Amenability to full automation is significant because the problem of algorithmically verifying correctness of systems with an unbounded number of processes, popularly known as the parameterized model checking problem (PMCP), is a well-known undecidable problem [Suzuki 1988]. For instance, recent work [Jaber et al.

Authors' addresses: Christopher Wagner, Computer Science, Purdue University, USA, wagne279@purdue.edu; Nouraldin Jaber, Computer Science, Purdue University, USA, njaber@purdue.edu; Roopsha Samanta, Computer Science, Purdue University, USA, roopsha@purdue.edu.

2475-1421/2023/4-ART81
https://doi.org/10.1145/3586033

2021] identifies a class of modular models of distributed services, based on *distributed agreement protocols*, for which PMCP can be reduced to model checking of small, finite-state systems. However, a limitation of this approach (and of most PMCP decision procedures [Esparza et al. 1999; Finkel and Schnoebelen 2001; Jaber et al. 2020]) is that it requires each process in the distributed system model to be finite-state. This stipulation is not surprising—the distributed system becomes *doubly-unbounded* if, in addition to an unbounded number of processes, it has infinite-state processes.

In this paper, we seek to substantially expand the class of unbounded distributed agreement-based (DAB) systems which can be automatically and scalably verified via *modular and bounded verification*. We characterize modular models of DAB systems, unbounded in the number of processes *as well as* variable domains, whose verification can be reduced to that of small, bounded systems.

We tackle PMCP for such doubly-unbounded DAB systems, one dimension at time. We first focus on the verification of DAB systems with *unbounded variable domains* and some *arbitrary, fixed number n of processes*. In particular, we identify conditions under which their verification can be reduced to model checking of n-process DAB systems with small, bounded variable domains. We then focus on model checking of DAB systems with *small, bounded variable domains* and *an unbounded number n of processes*. Fortunately, this is possible using existing techniques (e.g., [Jaber et al. 2021]) for reducing PMCP for such systems to model checking of small, finite-state systems.

Our approach is characterized by the following features.

Value Symmetry. We first analyze if a system exhibits *symmetry* in its use and access of variables with unbounded domains, specifically, if the system's correctness is preserved under permutations of values from these domains. As is standard, verification of such symmetric systems can exploit the induced redundancies, for instance, through the use of a smaller quotient structure.

Localized Data Saturation. While a quotient structure is not guaranteed to be finite-state in general, in some cases, the system exhibits *data saturation*. This means that the number of distinct values appearing in any of its reachable states is bounded, so the size of its symmetry-reduced quotient structure has a finite upper bound, even when the system's data domains are unbounded. However, it is often infeasible to establish such a bound for a system with an *arbitrary* number of processes. Hence, we instead seek to check if a *value-symmetric* system also exhibits a property we call *localized data saturation*. This means that the number of distinct values appearing in certain *influential* parts of the system's state space is bounded while the number of distinct values appearing in other parts may be unbounded; the *influence* of different parts of the system's state space is determined by the system's semantics as well as the target safety specification.

Domain Cutoffs. We automatically detect when systems exhibit localized data saturation for a given safety specification and infer appropriate bounds on the data domains of the system's processes. We call these bounds *domain cutoffs*. We argue that domain cutoffs are more *flexible* than upper bounds used by classical data saturation and prove a *domain reduction* result which ensures the system with bounded data domains is safe iff the system with unbounded domains is safe.

Process Cutoffs. With the data domains reduced to a fixed, finite space, it now becomes possible to leverage existing results [Jaber et al. 2021] for PMCP of DAB systems with an unbounded number of finite-state processes. Specifically, PMCP can now use *process cutoffs* to reduce verification to that of systems with a fixed, finite number of processes. Notably, our domain reduction and cutoffs are not restricted to be used only with the results of Jaber et al. [2021]; instead, our inference of domain cutoffs can facilitate flexible combination with varied techniques to obtain process cutoffs.

To summarize, we make the following contributions for DAB system models:

(1) Domain-Reducible Verification (Sec. 4): We characterize systems which exhibit value symmetry and localized data saturation with respect to a given safety specification to define the notion of domain-reducible verification problems.

(2) Domain Cutoff Computation (Sec. 5): We present a sound procedure to *infer* pertinent bounds, called *domain cutoffs*, on the data domains of systems in domain-reducible verification problems.
(3) Symmetry-Based Domain Reduction (Sec. 6, Sec. 7): We *prove* that systems which have domain cutoffs for a given safety specification will exhibit an error trace iff a corresponding error trace exists for the system reduced according to the associated domain cutoff. Thus, we reduce verification of a doubly-unbounded system to verification of a system with a fixed, finite domain.
(4) Venus (Sec. 8): We develop a tool, Venus, for verification of doubly-unbounded systems that efficiently computes domain/process cutoffs and verifies sophisticated DAB systems.

2 ILLUSTRATIVE OVERVIEW

```
process Consortium

variables
    int[1,2] data
events
      br inform : int[1,2]
  env rz initialize : int[1,2]
  env rz influence : int[1,2]
  env br reset : unit

initial location Engage
  on recv(initialize) do
    data := initialize.payld
    goto Election

location Election
on partition<elect>(All,2)
    win: goto Deliberate
    lose: goto Wait

location Deliberate
  on recv(influence) do
    data := influence.payld
  on consensus<vc>(elect.winS,1,data) do
    data := vc.decVar[1]
    goto Decided

location Wait
  on recv(inform) do
    data := inform.payld
    goto ReplicaDone

location Decided
 on partition<share>(elect.winS,1)
    win: goto Announce
    lose: goto LeaderDone

location Announce
  on _ do
    broadcast(inform[data])
    goto LeaderDone

location LeaderDone
  on recv(reset) do
    data := default(data)
    goto Engage

location ReplicaDone
  on recv(reset) do
    data := default(data)
    goto Engage
```

Safety Property: Actors in locations `ReplicaDone` and `LeaderDone` must have equal values of the variable `data`.

Fig. 1. Mercury model of a Consortium process.

We motivate and illustrate our contributions with an example distributed system, *Consortium*, that uses different types of distributed agreement to achieve trust-based consensus.

2.1 Motivation: Distributed Agreement-Based Systems

Consortium. The Consortium distributed system involves a set of actors who want to mutually make a decision based on information they gather individually. In order to do this efficiently, a subset of the actors is elected and trusted with making the decision and announcing it to the rest of the actors. This resembles scenarios where a trade-off between trust and performance is needed (e.g., a consortium blockchain [Amsden et al. 2020; Hyperledger 2021]).

We model each actor/process of Consortium in Mercury [Jaber et al. 2021], a modeling language with inbuilt primitives for distributed agreement. As can be seen in Fig. 1, Mercury facilitates a

clean, *modular* design of DAB systems such as Consortium with its encapsulation of the intricacies of agreement protocols into agreement primitives, (highlighted in blue).

An actor initially starts in the `Engage` location, where it may receive an initial input from the environment (Line 12) before moving to the `Election` location. Then, the process coordinates with all other actors (Line 17) to elect at most two among them to deliberate. Notice that this election is performed using a "**`partition`**" agreement primitive with identifier `elect` (Line 17); this instance of **`partition`** expresses that `2` actors are elected from among `All` actors. The elected actors move to the `Deliberate` location where they are trusted to make a decision for everyone. The remaining actors move to the `Wait` location (Line 19), where they wait to be informed once the elected actors agree on a decision (Line 28). The elected actors in `Deliberate` may be influenced (by the environment) to update their proposal, stored in variable `data` (Line 22). Further, the elected actors make a decision using a "**`consensus`**" agreement primitive with identifier `vc` (Line 24); this instance of **`consensus`** models agreement on `1` value proposed in the `data` variable of processes elected in `Election`. After storing the decided value in the `data` variable, the elected actors move to `Decided` where they elect one actor (Line 33) to announce the agreed-upon value to all actors (Line 39). Next, the elected actors move to `LeaderDone` and all other actors move to `ReplicaDone`. All actors then go back to the initial location, reinitializing their variable, to start further rounds (Lines 43, 48).

Correctness Specification and Parameterized Verification. The correctness specification for a Consortium distributed system $\mathcal{M}(n)$ with n instantiations of the above actor is a safety property: all n actors in locations `ReplicaDone` and `LeaderDone` agree on the value of the variable `data`.

We wish to ensure that a Consortium distributed system with an arbitrary, *unbounded number of actors* is correct. In particular, we are interested in *fully-automated parameterized verification* of Consortium that seeks to algorithmically check if $\mathcal{M}(n)$ satisfies its correctness specification for all values of n. Furthermore, we are interested in *modular verification* that effectively exploits the modularity of Mercury's Consortium model.

Prior Work: Unbounded Number of Processes and Bounded Domains. While there exist multiple algorithms and tools for parameterized verification, we are aware of only two fully-automated, modular approaches [Jaber et al. 2020, 2021] that can tackle DAB systems, of which only one (QuickSilver [Jaber et al. 2021]) can tackle systems modeled in Mercury. In fact, QuickSilver can perform parameterized verification of Consortium, as shown in Fig. 1, *efficiently*, in under a second.

Sadly, QuickSilver is limited to Mercury systems composed of *finite-state processes* with a relatively modest number of states. Thus, even though QuickSilver can do verification of systems parameterized with an unbounded number of processes, it cannot handle systems with an unbounded number of processes *and* unbounded variable domains. Notice that a Consortium actor is finite-state with integer datatypes over small subranges. If the integer subrange datatype of the `data` variable in Fig. 1 is replaced with true integers (or even standard 32-bit integers), QuickSilver is unable to perform parameterized verification of the new system.

This work: Unbounded Number of Processes and Unbounded Domains. The inspiration for this work is two fold: (a) the effectiveness of Mercury in enabling modular design and verification of DAB systems and (b) the confirmation provided by QuickSilver that *fully-automated parameterized verification of such systems is possible!* We ask a natural question to help address QuickSilver's limitations:

> *Is fully-automated, modular, and scalable parameterized verification possible for DAB systems with large, unbounded, or infinite-state processes?*

In particular, can we algorithmically and efficiently perform parameterized verification of a *doubly-unbounded* version of Consortium where the integer subrange datatypes of variables and events are replaced with (unbounded) integers? In what follows, we continue to denote the Consortium distributed system with n instantiations of the finite-state actor in Fig. 1 as $\mathcal{M}(n)$. Further, we denote Consortium with n instantiations of the corresponding infinite-state actor (with unbounded integer datatypes) as $\mathcal{M}_\Delta(n)$.

2.2 Contributions

Domain-Reducible Verification. Our key contribution employs notions of *value symmetry* and *localized data saturation*, to enable a *reachability-preserving transformation* of Mercury systems with unbounded variable domains to Mercury systems with finite domains. We illustrate these concepts and our *domain reduction* on our target (doubly-unbounded) Consortium system.

Notice that an actor of $\mathcal{M}_\Delta(n)$ uses and accesses integer datatypes in a restricted way, in particular, as a *scalarset datatype* [Ip and Dill 1996]—expressions over this datatype are restricted to (dis)equality checks and variables. Notice also that the safety specification accesses integer-valued variables as scalarsets. This "value symmetry" in an actor of $\mathcal{M}_\Delta(n)$ as well as the safety specification implies preservation of correctness under permutations of scalarsets—an execution η of $\mathcal{M}_\Delta(n)$ violates the safety specification iff any execution corresponding to a permutation of the scalarset values in η violates the safety specification.

This preservation of reachability under scalarset permutations suggests that many system executions are redundant for verification, and not all of them need to be explored. Quotient structures are used commonly to exploit such redundancies to verify properties of symmetric systems more efficiently. For value-symmetric systems, if the number of *distinct* values appearing in any reachable state has a *finite upper bound*, the size of the quotient structure can have a finite upper bound, even when the systems' scalarset domains become unbounded or infinite; this property is called *data saturation* [Ip and Dill 1996]. In general, it can be difficult to establish such an upper bound for *every* reachable state in systems with an unbounded number of processes. However, we argue that if such an upper bound can instead be established on the number of distinct scalarset values appearing in *particular* process states where this distinctness affects the system's behavior, then safety verification of $\mathcal{M}_\Delta(n)$ can be reduced to safety verification of a system with finite integer domains!

By applying this insight to the unbounded Consortium system $\mathcal{M}_\Delta(n)$, we now assert that a bound of *two* distinct values can be established for $\mathcal{M}_\Delta(n)$, thereby reducing safety verification of $\mathcal{M}_\Delta(n)$ to safety verification of $\mathcal{M}(n)$ with integer subrange `[1,2]`. Any execution of $\mathcal{M}_\Delta(n)$ is equivalent (w.r.t. the safety property) to one in which only two distinct integer values appear. This stems from the fact that (a) only *one* value at a time is needed to represent the most recent result of consensus, and (b) each actor has only *one* local variable: `data`. While, in general, different actors may have distinct values in their `data` variable, because these values are never compared or communicated between actors (except during consensus), any execution in which actors have more than two distinct values in their local variables can be shown to imply the existence of an execution in which the actors have the same two values in their `data` variables: one value representing the most recent result of consensus, and one other value which is distinct from the result of consensus (although these two values may switch roles during the course of an execution). Ergo, to verify the unbounded-domain system $\mathcal{M}_\Delta(n)$, it suffices to verify the `[1,2]`-bounded-domain system $\mathcal{M}(n)$.

In this work, we identify conditions under which value symmetry and localized data saturation enable domain reduction for safety verification of Mercury systems with unbounded or infinite-state processes. In particular, if (a) all values from an unbounded scalarset domain held by a process in the system can be partitioned into *two* different *regions* at any point in a system execution and

(b) the maximum *sizes* of these two regions can be statically *bounded*, then the "large" unbounded scalarset domain can be replaced by a "small" finite one with size equal to the sum of the bounds of these two regions. In the first region, processes may be aware of a set of *globally distinguishable* values (where equality of values across processes affects the system's behavior). In the second region, processes may only have *local knowledge* of the values they hold (so equality of values across processes does not affect the system's behavior).

Domain Cutoff Computation. To support practical application of domain cutoffs, we present a procedure which analyzes Mercury processes to identify two regions as described above and their associated bounds. In doing so, we provide a concrete path to verification of Mercury models of DAB systems. The two regions yield Mercury systems with finite, bounded variable domains, which can then be verified using existing parameterized verification engines such as QuickSilver.

Symmetry-Based Domain Reduction. We prove the soundness of domain cutoffs for DAB systems by showing that if an unsafe execution exists in the "large" system, there must exist a related unsafe execution in the "small" system. This domain reduction provides a sound theoretical basis for verifying safety properties of DAB systems with both an unbounded number of processes and unbounded variable domains by first reducing their verification to that of a system with an unbounded number processes but small, finite variable domains.

Venus and Evaluation. We have implemented a tool, Venus, that combines our domain reduction and cutoffs with QuickSilver and can perform fully-automated, modular, and scalable parameterized verification of Mercury system models with large, unbounded, or infinite-state processes. In particular, Venus is able to verify, in under a second, that $\mathcal{M}_{\Delta}(n)$ satisfies its correctness specification for all values of n.

3 PROBLEM DEFINITION

Mercury is a modeling language for DAB systems consisting of an arbitrary number of identical processes and built on top of verified agreement protocols. As a native feature, Mercury includes two special event handlers, **`partition`** and **`consensus`**, which capture the behavior of common agreement protocols (e.g., leader election and consensus). We first review the syntax and semantics of Mercury systems, and then define the parameterized verification problem for such systems.

3.1 Mercury Systems

A Mercury process definition P has three main components: (i) a set of typed process-local variable declarations, (ii) a set of typed *event signatures* defining the set of coordination events which may occur during program execution, and (iii) a set of *action handlers* describing the behavior of a process when an event is initiated or received.

Variable Declarations. Each variable declaration consists of a variable name v and an associated variable domain which may be either a bounded integer range or the powerset of the set of process identifiers (i.e. v may store a set of process identifiers). We denote the set of variables as $\mathcal{V}$.

Event Declarations and Actions. Event declarations correspond to four different types of process coordination: (i) pairwise-rendezvous communication, denoted `rz`, where one process sends a message, and one other process receives it, (ii) broadcast communication, denoted `br`, where a process sends a message, and all other processes receive it, (iii) **`partition`** coordination, denoted `pc`, where a set of participating processes designates a finite subset of themselves as "winners", and all other participating processes as "losers", or (iv) **`consensus`** coordination, denoted `vc`, where each process in a set of participating processes proposes a value, and all participating processes agree on a finite subset of the proposed values.

Each event declaration consists of an event type (i.e., one of `rz`, `br`, `pc`, `vc`), event name `eID`, a payload data type (either a bounded integer range or **`unit`**), and optionally a keyword `env` if the event is an interaction with the environment. Each event declaration describes a set of events and a set of *actions*. An event $e := \texttt{eID}[val]$ corresponds to a particular payload value val of the associated payload data type.

An action is a *polarized* event where the polarity indicates if a process is initiating (denoted $e!$) or reacting to (denoted $e?$) the event. In particular, acting (resp. reacting) events are sends (resp. receives) of broadcasts and pairwise communication, and correspond to "winners" (resp. "losers") of agreement coordination[1]. We denote the sets of events, acting events, reacting events, and actions as $\mathcal{E}$, $\mathcal{E}!$, $\mathcal{E}?$, and $\mathcal{E}\text{↯}$.

Example 3.1. Consider the `inform` event declaration defined on Line 6 in Fig. 1 (i.e. `br inform : int[1,2]`), for the bounded-domain system $\mathcal{M}(n)$. The event has type `br`, name `inform`, and payload type `int[1,2]`. The set $\mathcal{E}$ of events associated with this event declaration (and induced by the values of the payload) is $\{\texttt{inform}[1], \texttt{inform}[2]\}$. For each event $e \in \mathcal{E}$, there is an acting and a reacting action. For instance, the acting and reacting actions for event `inform[1]` are `inform[1]!` and `inform[1]?`, respectively.

Action Handlers. Each location in the process definition is associated with a set of action handlers. An action handler comprises an action name, a guard, and a set of updates. A guard is a Boolean predicate over variables in $\mathcal{V}$ and the set of updates is a *parallel assignment* to each variable.

Mercury Semantics. We refer to the semantics of a Mercury process as *local* and the semantics of a Mercury system composed of multiple identical Mercury processes as *global*.

The local semantics of a Mercury process, P is given by a labeled state-transition system $M_P = (S, S_0, T)$ with a state space S, a set S_0 of initial states, and a set $T \subseteq S \times \mathcal{E}\text{↯} \times S$ of transitions induced by the set of action handlers. The state space S corresponds to all possible valuations of variables in $\mathcal{V} \cup \{v_{loc}\}$, where v_{loc} is a special variable defined to store the location. We denote the value of a variable v in a local state $s \in S$ as $s(v)$. The set S_0 of initial states consists of states where the location variable v_{loc} is set to the initial location (denoted ℓ_0) and all variables with the same domain are set to some default initial value. Action handlers for an event e induce acting and reacting transitions in T labeled with $e!$ and $e?$, respectively. In particular, T contains a transition $(s, e\text{↯}, s')$ for $e\text{↯} \in \{e!, e?\}$ iff there exists an action handler for $e\text{↯}$ such that the handler's guard is $true$ in s and s' is obtained by applying the handler's updates to s.

The global semantics of a Mercury system, consisting of n identical processes $P_1, \ldots, P_n$ and an environment process E communicating synchronously, is given by a labeled transition system $\mathcal{M}(n) = \langle Q, Q_0, R \rangle$ describing their parallel composition. Here, $Q = S^n \times S_E$ is the set of global states, $Q_0 = S_0^n \times S_{0,E}$ is the set of initial global states, and $R \subseteq Q \times \mathcal{E} \times Q$ is the global transition relation capturing the process coordination necessary for different event types and payloads. For instance,

1. R contains a "broadcast transition" (q, e, q') for broadcast event e iff (1) one process P_i has a local "broadcast send" transition $(q[i], e!, q'[i])$ and (2) all other processes P_j have corresponding local "broadcast receive" transitions $(q[j], e?, q'[j])$ with $q'[j]$ updated with the payload value.
2. R contains a "**`Consensus`** transition" $(q, \texttt{eID}[V'], q')$ for **`Consensus`** event $e = \texttt{eID}[V']$, participant set Prt, and set of winning values V' iff (1) each participating process in Prt has a consistent view of the other participants[2], (2) every value v in V' is proposed by some process in Prt with a local transition for $e!$, (3) each process in Prt has a corresponding local $e?$ local transition, and (4) the local states of all other processes remain unchanged.

[1] A process winning consensus means that it proposed one of the agreed-upon values.
[2] That is, each process stores the same participant set Prt in its own state locally.

Example 3.2. Consider an unbounded-domain Consortium system $\mathcal{M}_\Delta(4)$ with four processes p_1 through p_4. Fig. 2 shows a broadcast transition on event e = `inform[99]`. Process p_1 has local transition $(q[1], e!, r[1]) \in T$ due to the action handler on Line 37 of Fig. 1, where the process moves from location `Announce` to location `LeaderDone` while transmitting the value 99 from its `data` variable. Process p_2 has an implicit action handler for `inform[99]`? (not depicted in Fig. 1), which causes p_2 to have local transition $(q[2], e!, r[2]) \in T$ where it stays in the location `LeaderDone` without changing its value. Finally, processes p_3 and p_4 take transitions $(q[i], e?, r[i]) \in T$ where $i \in [3, 4]$, according to the `inform` receive handler on Line 27 of Fig. 1, which moves them from location `Wait` to location `ReplicaDone` while receiving the value 99 and storing it in the `data` variable. Thus, the global transition (q, e, r) is in the global transition relation R.

We refer the reader to App. A for the complete local and global Mercury semantics. An execution of a global transition system $\mathcal{M}(n)$ is defined in a standard way: a global execution is a (possibly infinite) sequence of global states, $q_0, q_1, \ldots$ in Q such that for each $j \geq 0$, $(q_j, e, q_{j+1}) \in R$ for some event e. Global state q is reachable if there exists a finite execution of $\mathcal{M}(n)$ that ends in q.

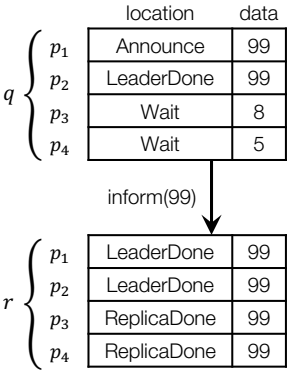


Fig. 2. A broadcast transition for the event e = `inform[99]`.

3.2 The Parameterized Verification Problem for Mercury Systems

The correctness of Mercury systems is given by a global specification Φ in the form of **`atmost`** clauses. Each **`atmost`** clause, written **`atmost`**(k, ϕ), is a Boolean predicate over the global states Q that no more than some threshold number k of processes are in some set $\{s \in S \mid s \models \phi\}$ of local states characterized by ϕ. An error state for such a clause is simple a global state which exceeds the threshold number of processes in those states. These **`atmost`** clauses are a natural fit parameterized distributed systems, as they define a set of error states for the system instantiated with every number of processes.

For a system $\mathcal{M}(n)$ with some number $n \in \mathbb{N}$ of *finite-state* processes P and a correctness specification Φ, we use $\mathcal{M}(n) \models \Phi$ to denote that the system $\mathcal{M}(n)$ satisfies Φ. The *parameterized model checking problem* (PMCP) targets the verification of a family $\mathcal{M}(\mathbb{N})$ of systems $\{\mathcal{M}(0), \mathcal{M}(1), \ldots\}$ w.r.t. correctness specification Φ. In particular, PMCP seeks to check if $\forall n.\ \mathcal{M}(n) \models \Phi$ [Bloem et al. 2015]. Note that this standard formulation of PMCP assumes that each process P has a finite state space. In order to enable reasoning about Mercury processes with unbounded and possibly infinite state spaces, we introduce new notation and a new formulation for the parameterized verification problem.

We denote a Mercury process with a set Δ of (possibly infinite) data domains as P_Δ and a process with a set $\overline{\Delta}$ of finite data domains as $P_{\overline{\Delta}}$. We denote a Mercury system with n instances of processes P_Δ (resp. $P_{\overline{\Delta}}$) as $\mathcal{M}_\Delta(n)$ (resp. $\mathcal{M}_{\overline{\Delta}}(n)$). In this paper, we target the parameterized verification problem over a family $\mathcal{M}_\Delta(\mathbb{N})$ of Mercury systems $\{\mathcal{M}_\Delta(0), \mathcal{M}_\Delta(1), \ldots\}$, defined as:

$$\forall n.\ \mathcal{M}_\Delta(n) \models \Phi.$$

The above problem generalizes PMCP for Mercury systems from the verification of an infinite family $\mathcal{M}_{\overline{\Delta}}(\mathbb{N})$ of finite-state systems $\{\mathcal{M}_{\overline{\Delta}}(0), \mathcal{M}_{\overline{\Delta}}(1), \ldots\}$ to an infinite family $\mathcal{M}_\Delta(\mathbb{N})$ of infinite-state systems $\{\mathcal{M}_\Delta(0), \mathcal{M}_\Delta(1), \ldots\}$.

4 DOMAIN-REDUCIBLE VERIFICATION

To enable parameterized verification over a family $\mathcal{M}_\Delta(\mathbb{N})$ of Mercury systems, we utilize value symmetry and localized data saturation to characterize verification problems, denoted $\langle \mathcal{M}_\Delta(n), \Phi \rangle$, that are *domain-reducible*. These problems permit a reachability-preserving transformation that replaces the set of data domains Δ of $\mathcal{M}_\Delta(n)$ with a set $\overline{\Delta}$ of finite, bounded domains, which we call the *domain cutoff*. For now, we consider $\mathcal{M}_\Delta(n)$ with only a single variable (and thus, a single data domain). Under this assumption, we use the single domain δ (resp. $\overline{\delta}$) and the singleton set $\Delta = \{\delta\}$ (resp. $\overline{\Delta} = \{\overline{\delta}\}$) interchangeably. For the infinite-state Mercury system $\mathcal{M}_\Delta(n) = \langle Q, Q_0, R \rangle$, let $\mathcal{M}_{\overline{\Delta}}(n) = \langle \overline{Q}, \overline{Q}_0, \overline{R} \rangle$ denote the Mercury system with n finite-state processes $P_{\overline{\delta}}$, where $P_{\overline{\delta}}$ is obtained from P_δ by replacing P's domain δ with a finite domain cutoff $\overline{\delta}$.

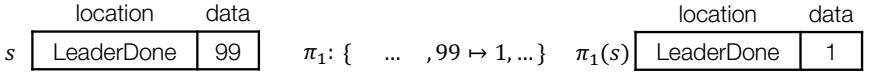
In this section, we first characterize value symmetry (Sec. 4.1). Then, we define *regions* of a process and their properties which enable domain reduction (Sec. 4.2). Finally, we combine these two notions to define domain-reducible verification problems (Sec. 4.3).

4.1 Permutations and Scalarsets

We first review standard notions of permutations and scalarsets. All permutations are w.r.t. *the* unbounded, symmetric domain $\delta \in \Delta$ which is the target of our reduction.

Permutations. A δ-permutation, $\pi : \delta \mapsto \delta$, is a bijection mapping the set δ onto itself. We define liftings of δ-permutations to local states and events as follows. An application $\pi(s)$ of a δ-permutation π to a local state s is defined so that the value of a variable v in the permuted state $\pi(s)$ is equal to the permutation of the value of v in the original state s (i.e., $\forall v \in \mathcal{V} : (\pi(s))(v) = \pi(s(v))$). An application $\pi(e)$ of a δ-permutation π to an event $e = \texttt{eID}[val]$ is $\texttt{eID}[\pi(val)]$, the permutation $\pi(e\wr)$ of an action of event e is $(\pi(e))\wr$, and the permutation of a local transition $(s, e\wr, s') \in T$ is $(\pi(s), \pi(e\wr), \pi(s'))$.

Example 4.1. We illustrate permutations on the unbounded-domain Consortium system $\mathcal{M}_\Delta(n)$. The figure below shows an example of applying a δ-permutation π_1 to the local state of process p_1 from Fig. 2 (here denoted s). Let δ denote the type `int`. Recall that the local variable `data` is of type `int`. The permutation π_1 maps the value 99 in state s to 1 in the permuted state $\pi_1(s)$ and maps all other values appropriately so that π_1 is a valid permutation (i.e., a bijection over the `int` domain). Notice that the value of the location variable v_{loc} is not of type `int` and, hence, it is not changed in the permuted state (i.e. $\pi_1(\texttt{LeaderDone}) = \texttt{LeaderDone}$).



Scalarsets. A scalarset domain [Ip and Dill 1996] is a set of distinct elements with restricted operations. Specifically:

(i) All valid scalarset terms are variable references; there are no scalarset constants.
(ii) Scalarset terms may only be compared using (dis)equality and only with terms of exactly the same scalarset type.
(iii) Scalarset variables may only be assigned values of the same scalarset type.

These restrictions ensure that scalarset values are interchangeable, so the local transition relation of a Mercury process is invariant under permutations of a scalarset:

Lemma 4.2. $\forall \pi : \forall (s,\ \alpha,\ s') \in T : \pi((s,\ \alpha,\ s')) \in T$.

Proof Sketch. Since scalarsets may only be used in direct assignment and equality comparisons, the existence of the transition is preserved by π, because π is a bijection. All equalities in the definition of T are preserved by π, so Lemma 4.2 follows by applying the permutation π to s, α, and s'.

The interchangeability of scalarset values also ensures that permuting scalarset values in an initial (resp. error) state results in another initial (resp. error) state. Further, we note that the bounded integer domains in Mercury can be treated as scalarsets if used according to the restrictions above. For instance, in $\mathcal{M}_\Delta(n)$, the `data` variable of Fig. 1 is of type `int` but can be treated as a scalarset variable because it conforms to the constraints (i) through (iii).

Definition 4.3 (Value Symmetry). A process P exhibits *value symmetry* with respect to a domain δ when P treats δ as a scalarset (according to constraints (i) through (iii) above).

Data Saturation. A classical result [Ip and Dill 1996] states that, in a system with only scalarset values, if the number of distinct values appearing in any reachable state of a transition system is finitely bounded, then the number of permutation-equivalent classes the systems states may belong to is also bounded, so the size of its symmetry-reduced quotient structure has a finite upper bound, even when the systems' data domains are unbounded.

We note that such a bound on the number of distinct values appearing in each reachable state often does not exist in systems composed of an unbounded number of processes with unbounded variable domains. Consider, for example, our Consortium system with an unbounded domain. In the location `Election`, there may be an arbitrary number of processes which have received a `initialize` event from the environment and updated their `data` variable to a value which is distinct from all values held by other processes in the system. Thus, for any candidate bound B on the number of distinct values appearing in any reachable state, there exists some reachable state in which at least $B + 1$ processes hold distinct values in the `data` variable. So, traditional data saturation cannot be applied to such a system.

4.2 Regions

To address this shortcoming of data saturation's applicability, we introduce *localized* data saturation. We observe that the distinctness of some values in the system may not affect the system's semantics. By dividing a system into two different regions—(i) one region where distinctness of values across processes impacts the system's semantics, but the number of values is bounded, and (ii) another region where the number of distinct values may be unbounded but distinctness of these values across processes does not impact the system's semantics—it may still be possible to verify the safety of this unbounded-state process by exploring a bounded system.

Definition 4.4 (Region). A *region* Ψ of a process P is a subset of the locations defined in P.

Example 4.5. For the Consortium system, any subset of {`Engage`, `Election`, `Deliberate`, `Wait`, `Decided`, `Announce`, `LeaderDone`, `ReplicaDone`} is a region.

In order to identify and refer to values for which equality across processes may (or may not) affect the system's semantics, we introduce some notation. Given a global state q, we denote the set of δ-values held by processes in q as $\texttt{proj}_\delta(q)$, and appearing in the local state $q[i]$ of the i^{th} process as $\texttt{proj}_\delta(q[i])$. Given a global state q and a region Ψ, we denote the set of δ-values held by processes in locations in Ψ as $\texttt{proj}^\Psi_\delta(q)$, i.e., $\texttt{proj}^\Psi_\delta(q) = \{s(v) \mid s(v_{loc}) \in \Psi \text{ and } \exists i : q[i] = s\}$. With this, we can characterize regions in which the number of distinct values present is bounded.

Definition 4.6 (Value-Stable Region). A region Ψ is *value-stable* if there exists a finite integer ρ, with $\rho > 0$, such that for all n and all reachable global states q of $\mathcal{M}_\Delta(n)$, $|\texttt{proj}^\Psi_\delta(q)| \leq \rho$.

Thus, for a value-stable region, there exists a finite upper bound ρ on the number of distinct values in $\texttt{proj}_{\delta}^{\Psi}(q)$ over all reachable global states q of $\mathcal{M}_{\Delta}(n)$ for *every* system size n. In other words, the bound ρ is independent of the number of processes in the system.

Example 4.7. For the Consortium system, the set {Decided, Announce, LeaderDone} is a value-stable region, because the vc consensus action ensures that only 1 value is present in the Decided location at a time, and any process which reaches the Announce or LeaderDone location must carry the value from the Decided location. In this case, the bound ρ is 1. Regardless of the number of processes in the system, no global state can be reached where 2 or more distinct values are simultaneously each held by some process in Decided, Announce, or LeaderDone. Determining such regions for a given process definition is explored in more detail in Sec. 5.

Finally, we characterize regions which capture all locations for which equality (or disequality) of values across processes impacts the system's semantics.

Definition 4.8 (Bounded Region). A value-stable region Ψ of process P is *bounded* w.r.t. specification Φ if it includes (i) the initial location, (ii) all destination locations of consensus handlers, (iii) all locations with handlers for sending actions with payloads, and (iv) any error locations identified by the specification Φ.

Example 4.9. For the Consortium system, the set {Engage, Decided, Announce, LeaderDone, ReplicaDone} is a bounded region with bound $\rho = 1$. How it is established that this region is value-stable is discussed in Sec. 5. Given that it is a value-stable region, this set satisfies condition (i) by including the Engage location, condition (ii) by including the Decided location, condition (iii) by including the Announce location, and condition (iv) by including the LeaderDone and ReplicaDone locations.

Global transitions of Mercury systems are defined according to a set of derivation-style rules in App. A. Some of these transition rules enforce (dis)equality between values held by separate processes in the *destination* state of the resulting transition as a result of data flow in the global semantics (e.g. from an assignment). Conditions (i) through (iv) ensure that, for any such global transition which enforces the (dis)equality of δ-values across processes, the appropriate δ-values for which that (dis)equality must hold are in the bounded region in the state of the transition where the (dis)equality must hold. As a result, the number of such values is limited by the (finite) global bound ρ of the bounded region Ψ. Additionally, no system transition requires any values *outside* of Ψ to be distinct across processes. Thus, any disequality of values across processes outside of the bounded region is unnecessary and may be ignored or eliminated. We defer to Sec. 5.2 for an example of a bounded region for the Consortium system.

Therefore, all transitions of the system $\mathcal{M}_{\Delta}(n)$ may depend upon at most ρ values (limited by the bound on region Ψ) which are *globally distinguished* across processes, and *one* additional value which serves as a stand-in for an arbitrary value outside of Ψ. So, for any trace in the system $\mathcal{M}_{\Delta}(n)$ with an arbitrarily-large domain Δ, there should exist a corresponding trace in a system $\mathcal{M}_{\overline{\Delta}}(n)$ with domain $\overline{\Delta}$ of size $\rho + 1$, even if the original domain is unbounded. Hence, we say that such a system exhibits localized data saturation.

Definition 4.10 (Localized Data Saturation). A system $\mathcal{M}_{\Delta}(n)$ exhibits *localized data saturation* w.r.t. the specification Φ if there exists a region Ψ which is bounded w.r.t. Φ.

4.3 Domain-Reducible Verification

Having defined value-symmetry and localized data saturation, we can now combine these notions to characterize verification problems which are amenable to domain reduction.

Definition 4.11 (Domain-Reducible Verification). A verification problem $\langle \mathcal{M}_\Delta(n), \Phi \rangle$ is *domain-reducible* if the system $\mathcal{M}_\Delta(n)$ exhibits value symmetry and localized data saturation with respect to the specification Φ.

Intuitively, the semantics of a domain-reducible verification problem $\langle \mathcal{M}_\Delta(n), \Phi \rangle$ only depends on a finite number of symmetric data values at a time. We show in Sec. 6 that the safety of $\mathcal{M}_\Delta(n)$ with respect to Φ can be determined by checking the verification problem $\langle \mathcal{M}_{\overline{\Delta}}(n), \Phi \rangle$, where the domain cutoff $\overline{\Delta}$ consists of the same domains as Δ, except that δ is replaced with another scalarset domain $\overline{\delta}$ of size $\rho + 1$.

Further, if we can then compute a process cutoff c for the reduced verification problem $\langle \mathcal{M}_{\overline{\Delta}}(n), \Phi \rangle$ using some cutoff-based procedure for parameterized verification of systems with *finite* variable domains (e.g., [Jaber et al. 2021]), then the verification problem $\langle \mathcal{M}_{\overline{\Delta}}(n), \Phi \rangle$ is reduced to checking the fully-finite verification problem $\langle \mathcal{M}_{\overline{\Delta}}(c), \Phi \rangle$. Ultimately, this two-step reduction allows one to verify the safety of $\mathcal{M}_\Delta(n)$ w.r.t. Φ by checking whether the finite system $\mathcal{M}_{\overline{\Delta}}(c)$ satisfies Φ.

Theorem 4.12. *If (i) $\langle \mathcal{M}_{\overline{\Delta}}(n), \Phi \rangle$ is domain-reducible with domain cutoff $\overline{\Delta}$, and (ii) c is a valid cutoff for $\mathcal{M}_{\overline{\Delta}}(n)$, then* $\left(\forall n : \mathcal{M}_\Delta(n) \models \Phi\right) \iff \left(\forall n : \mathcal{M}_{\overline{\Delta}}(n) \models \Phi\right) \iff \left(\mathcal{M}_{\overline{\Delta}}(c) \models \Phi\right)$.

5 COMPUTING DOMAIN-CUTOFFS

In this section, we discuss how we can *automatically* determine that a verification problem $\langle \mathcal{M}_\Delta(n), \Phi \rangle$ is domain-reducible. While value-symmetry can be determined using simple syntactic checks on the process definition P_δ, observing localized data saturation is more complex. In what follows, we discuss a procedure for establishing that $\mathcal{M}_\Delta(n)$ exhibits localized data saturation by computing a bounded region Ψ with bound ρ for the process definition P_δ. If the procedure succeeds, the domain cutoff is $\overline{\delta}$ with size $\rho + 1$.

5.1 Location Transition System (LTS)

To determine Ψ with bound ρ, we inspect the movement of values in the Mercury system. In particular, we wish to examine values that may *flow* into a set of locations *without* constructing the unbounded local semantics of P_δ. We begin by defining a finite-state transition system $\mathcal{A}$ capturing the *control-flow* of a Mercury process: $\mathcal{A} = \langle \mathcal{L}, \mathcal{E} \rangle$, where each node in $\mathcal{L}$ corresponds to a location in P_δ and there exists an edge (a, b) in $\mathcal{E}$ if there exists a transition in the local semantics of P_δ between states in locations a and b. Notice that this causes $\mathcal{A}$ to be an over-approximation of the local semantics of P_δ. Let the node corresponding to location ℓ be denoted $node(\ell)$, the location of node $x \in \mathcal{L}$ be denoted $loc(x)$, and the set of local states with $v_{loc} = loc(x)$ be denoted $[x]$. To track relevant *data flow*, we extend $\mathcal{A}$ with bookkeeping information for each edge, in particular, the set of action handlers yielding the edge. Let the set of edges derived from a handler h be $edges(h)$.

Example 5.1. The **partition** handler (Fig. 1, Line 33) is defined under location `Decided` and includes a `goto` statement which sends a process to location `Announce`, so $\mathcal{A}$ includes an edge $e = (x_1, x_2) \in \mathcal{E}$, where $loc(x_1) =$ `Decided` and $loc(x_2) =$ `Announce`.

Finally, we extend our notion of a region from Sec. 4.2 to LTSs. Let an *abstract region* $r \subseteq \mathcal{L}$ be a set of nodes representing a region composed of concrete locations. Let $[r]$ be the concrete region $\{loc(x) \mid x \in r\}$ that the abstract region r represents. We say an abstract region r is value-stable (resp. bounded) iff the corresponding concrete region $[r]$ is value-stable (resp. bounded). The bound $\rho(r)$ of an abstract region r is defined to be equal to the bound ρ of $[r]$.

Algorithm 1: Determining an abstract bounded region r.

```
1 procedure getAbstractBoundedRegion(𝒜, P_δ)
      Inputs : A process P_δ, and the LTS 𝒜
      Output : ⊥ or r, a BAR
2     regions = getInitialRegions(P_δ, 𝒜)
3     regions = expandRegions(regions, 𝒜)
4     regions = mergeRegions(regions, 𝒜)
5     r = getMinimalRegion(regions)
6     if r == null then
7         return ⊥
8     return r
```

Example 5.2. For an abstract region $r = \{node(\texttt{Engage}), node(\texttt{LeaderDone}), node(\texttt{ReplicaDone})\}$ with bound $\rho(r) = 3$, the concrete region represented by r is $[r] = \{\texttt{Engage}, \texttt{LeaderDone}, \texttt{ReplicaDone}\}$ with bound $\rho = 3$.

5.2 Determining Bounded Regions

We now present a procedure for determining a suitable *bounded abstract region* (BAR), defined in Algo. 1. If successful, Algo. 1 returns an abstract region (AR) r such that $[r]$ is value-stable and satisfies the conditions described in Def. 4.8, and the variable domain δ may be reduced. Otherwise, Algo. 1 fails to find such an AR and returns $\perp$, so verification does not proceed.

Initial Regions. The first step in Algo. 1 (Line 2) is to identify a set of initial ARs where for each AR r in that set, it is clear that *any* number of processes in $[r]$ hold a finite number of distinct values of type δ in their variables (i.e., $\rho(r)$ is finite). To that end, we identify three types of such initial ARs as defined in Algo. 2. First, for each **consensus** action vc, create an AR composed of all the nodes corresponding to locations where vc terminates (Line 3). The bound ρ of this AR matches the cardinality of the set of decided values of the **consensus** action vc. Intuitively, by the nature of **consensus**, we know that the result of **consensus** is consistent across participating processes in these locations. Second, for each location which can only be occupied by a single process at a time, as determined by a lightweight, sufficient check on $\mathcal{A}$, we create an AR (with $\rho = 1$) consisting of the singleton set containing the node associated with that location (Line 8). Finally, we create a single AR (with $\rho = 1$) encoding the initial location, where all processes hold some initial default value [3] (Line 12) Here, we denote by $gets(x, v, x', v')$ that the value in variable v' in location $loc(x')$ may come from variable v in location $loc(x)$ via a direct assignment or transmission. While operating under the assumption that $\mathcal{M}_\Delta(n)$ has only one variable v, we will sometimes use $gets(x, x')$ as shorthand for $gets(x, v, x', v)$. Thus, by $r = \{x \mid \nexists\, x' : gets(x, x')\}$ we mean that r is assigned the set of nodes x such that no transition may bring a new value into location $loc(x)$.

Example 5.3. For the Consortium system (Fig. 1), Algo. 2 proceeds as follows. The set *regions* is initially empty (Line 2). Then, the only consensus action is vc, so $consActions(P_\delta) = \{\texttt{vc}\}$ and the loop variable a becomes vc (Line 3). The set of handlers for $a = \texttt{vc}$ contains only the handler in location Deliberate which goes to Decided (Fig. 1, Line 24). So, the region r becomes the singleton set containing the destination node from the edge $(node(\texttt{Deliberate}), node(\texttt{Decided})) \in \mathcal{E}$. That is, $r = \{node(\texttt{Decided})\}$ (Algo. 2, Line 5). The cardinality of action a is $card(\texttt{vc}) = 1$, so $\rho(r) = 1$ (Line 6). The region $r = \{node(\texttt{Decided}\}$ with bound 1 is then added to *regions* (Line 7). There are no more consensus actions in $consActions(P_\delta)$, so control proceeds to Line 8. The

[3] While the initial value is consistent across processes, this value is arbitrary. Thus, each value in the variable domain yields a separate, valid initial global state.

Algorithm 2: Constructing initial regions.

```
1  procedure getInitialRegions(P_δ, 𝒜)
       Inputs : A Mercury program P_δ and an LTS 𝒜
       Output : a set of value-stable ARs
2      regions = {}                                  ▷ initialize the set of regions
3      foreach a ∈ consActions(P_δ) do
4          H = handlersOf(a)
5          r = {dst(e) | h ∈ H ∧ e ∈ edges(h)}       ▷ get destination nodes of consensus handlers
6          ρ(r) = cardOf(a)                          ▷ set bound ρ to cardinality of consensus action a
7          regions = regions ∪ r
8      foreach s ∈ singleProcRegion(𝒜) do
9          r = {s}                                   ▷ region contains just one node
10         ρ(r) = 1                                  ▷ bound is 1, because there is only 1 process
11         regions = regions ∪ r
12     r = {x | ∄ x' : gets(x, x')}                  ▷ create region for initial node with no incoming data
13     ρ(r) = 1                                      ▷ bound is 1, for the initial value
14     regions = regions ∪ r
15     return regions
```

Consortium system does not have any locations which are guaranteed to only be occupied by a single process at a time (i.e., $singleProcRegion(\mathcal{A}) = \emptyset$), so control moves directly to Line 12. At this point, a region is created for the initial node with no incoming data. While $\mathcal{A}$ does include edges $(node(\texttt{LeaderDone}), node(\texttt{Engage})) \in \mathcal{E}$ and $(node(\texttt{ReplicaDone}), node(\texttt{Engage})) \in \mathcal{E}$ due to the handlers on Line 43 and Line 48 of Fig. 1, these are *reset* handlers, which set all variables to their default value and do not propagate any non-initial values to the initial location, Engage. Thus, reset handlers do not affect the *gets* relation. Subsequently, r is assigned the singleton set {Engage} on Line 12. The bound $\rho(r)$ is 1, as only 1 initial value is present in the initial region (Line 13). The initial region r is added to the *regions* set (Line 14), and the resulting set $regions = \{\{node(\texttt{Decided})\}, \{node(\texttt{Engage})\}\}$ is returned (Line 15), with both regions having a bound of 1.

Expanding Regions. The second step in Algo. 1 (Line 3) is to expand the initial regions while maintaining the bound ρ. Intuitively, Algo. 3 follows the flow of values so that when the variable v may *only* get its value in a location $loc(x)$ from some AR r, the node x is added to r (Line 6).

Example 5.4. For the Consortium system (Fig. 1), Algo. 3 is called with the set $regions = \{\{node(\texttt{Decided})\}, \{node(\texttt{Engage})\}\}$ of initial regions. For the first iteration of the loop, let $r = \{node(\texttt{Decided})\}$ (Line 2). The only handler in the Decided location has two **goto** statements: one each to Announce and LeaderDone. Note that there is a **goto** LeaderDone statement in location Announce (Fig. 1, Line 37), and $node(\texttt{Announce}) \notin r$, so $node(\texttt{LeaderDone})$ cannot yet be added to r. Thus, only $node(\texttt{Announce})$ is added to r (Algo. 3, Line 7), and the *changeMade* flag is set (Line 8), so the loop continues. On the next iteration, $r = \{node(\texttt{Decided}), node(\texttt{Announce})\}$, so $node(\texttt{LeaderDone})$ can then be added to r (Line 7). Additionally, the inform broadcast from the Announce location (Fig. 1, Line 39) is handled in the Wait location (Line 28) such that all processes in Wait replace their data value with one received from a process in Announce as they move to the ReplicaDone location. Thus, $node(\texttt{ReplicaDone})$ can also be added to r (Algo. 3, Line 7). After this iteration, $r = \{node(\texttt{Decided}), node(\texttt{Announce}), node(\texttt{LeaderDone}), node(\texttt{ReplicaDone})\}$. On the next iteration, no more nodes can be added, as the reset handlers from LeaderDone and ReplicaDone do not propagate any stored values. So, control proceeds to the next iteration of the loop (Line 2) where the unexpanded region $r = \{node(\texttt{Engage})\}$. Since the value in the data

Algorithm 3: Expanding regions.

```
1  procedure expandRegions(regions, 𝒜)
       Inputs : The LTS 𝒜 and an initial set regions of regions
       Output : a set of value-stable ARs
2      foreach r ∈ regions do
3          do
4              changeMade = false                    ▷ initialize fixed-point iteration
5              foreach x ∉ r do
6                  if {x′ | gets(x, x′)} ⊆ r then
7                      r = r ∪ {x}                   ▷ add node which only gets data from inside region r
8                      changeMade = true             ▷ continue fixed-point computation
9          while changeMade;
10     return regions
```

variable is not preserved by the `initialize` handler (Fig. 1, Line 12), no nodes are added to r. After this iteration, no regions remain, so Algo. 3 returns $regions = \{\{node(\texttt{Decided}), node(\texttt{Announce}), node(\texttt{LeaderDone}), node(\texttt{ReplicaDone})\}, \{node(\texttt{Engage})\}\}$ (Line 10).

Merging Regions. After expanding individual ARs, the third step in Algo. 1 (Line 4) is to attempt to merge ARs which are mutually exclusive (i.e., processes are only in the locations of one of them when there are no processes in the locations of the other) according to Algo. 4. If two ARs r_1 and r_2 are mutually exclusive (Line 4), processes may not hold values in r_1 and r_2 simultaneously, so their bounds are independent and the larger bound applies to both regions. Thus, we create a new AR $r = r_1 \cup r_2$ (Line 5) with $\rho(r) = max(\rho(r_1), \rho(r_2))$ (Line 6).

We determine that the two regions r_1 and r_2 are mutually exclusive using a disjunction of sufficient (but not necessary), static checks on $\mathcal{A}$, encapsulated here by the function *areExclusive*. One such sufficient condition is to require that every path from r_1 to r_2 (and vice versa) must include some globally-synchronous transition (e.g. consensus) which is only enabled when all processes are outside of r_1 (resp. r_2).

Example 5.5. For the Consortium system (Fig. 1), Algo. 4 is called with the set $regions = \{\{node(\texttt{Decided}), node(\texttt{Announce}), node(\texttt{LeaderDone}), node(\texttt{ReplicaDone})\}, \{node(\texttt{Engage})\}\}$. First, the set *mergedRegions* is initialized with the contents of *regions* (Line 2). Then, beginning with $r_1 = \{node(\texttt{Decided}), node(\texttt{Announce}), node(\texttt{LeaderDone}), node(\texttt{ReplicaDone})\}$ and $r_2 = \{node(\texttt{Engage})\}$ (Line 3), we find that (i) for any processes to enter the locations of r_1, all processes must have left the `Engage` location, since all processes must engage in the `elect` transition together from the `Election` location (Fig. 1, Line 17), and (ii) for any processes to enter r_2, they must all leave r_1 at once via a reset transition (Fig. 1, Line 43 and Line 48). Thus, $areExclusive(r_1, r_2)$ returns *true* (Algo. 4, Line 4) and the merged region $r_{mrg} = r_1 \cup r_2 = \{node(\texttt{Decided}), node(\texttt{Announce}), node(\texttt{LeaderDone}), node(\texttt{ReplicaDone}), node(\texttt{Engage})\}$ is created (Line 5). The bounds $\rho(r_1)$ and $\rho(r_2)$ are both 1, so $\rho(r_{mrg}) = max(\rho(r_1), \rho(r_2)) = max(1, 1) = 1$ (Line 6). On Line 7, r_{mrg} is added to the set of merged regions. The execution is the same for when the contents of r_1 and r_2 are swapped, so the set of merged regions remains the same. Finally, the set $mergedRegions = \{\{node(\texttt{Decided}), node(\texttt{Announce}), node(\texttt{LeaderDone}), node(\texttt{ReplicaDone})\}, \{node(\texttt{Engage})\}, \{node(\texttt{Decided}), node(\texttt{Announce}), node(\texttt{LeaderDone}), node(\texttt{ReplicaDone}), node(\texttt{Engage})\}\}$ is returned, with all regions having a bound of 1 (Line 8).

A Minimal Bounded Region. After constructing a set of ARs with associated bounds, the final step in Algo. 1 (Line 5) is to select one which meets the conditions in Def. 4.8 and so represents a bounded region. That is, we select a region r which includes (i) the node representing the initial location,

Algorithm 4: Merging regions.

```
1 procedure mergeRegions(regions, 𝒜)
      Inputs : The LTS 𝒜 and an initial set regions of regions
      Output : a set of value-stable ARs
2     mergedRegions = regions
3     foreach (r1, r2) ∈ regions × regions | r1 ≠ r2 do
4         if areExclusive(r1, r2) then
5             r_mrg = r1 ∪ r2
6             ρ(r_mrg) = max(ρ(r1), ρ(r2))        ▷ larger bound applies to new merged region
7             mergedRegions = mergedRegions ∪ r_mrg
8     return mergedRegions
```

(ii) all nodes representing destination locations of consensus handlers, (iii) all nodes representing locations with handlers for sending actions with payloads, and (iv) any nodes corresponding to locations referenced by the global specification Φ. If no such region exists, `null` is returned.

We select the final *bounded abstract region* (BAR) r that satisfies these criteria and has the minimal bound $\rho(r)$ of all such candidate BARs. The bounded region Ψ is then $[r]$ with bound $\rho(r)$. Hence, the concrete region Ψ is bounded, because it is value-stable, and it satisfies the criteria above. We consider the soundness of Algo. 1 for computing bounded abstract regions in App. C.1.

Example 5.6. Consider the Consortium system (Fig. 1). According to Algo. 2, the initial regions include $r_1 = \{node(\texttt{Decided})\}$ with $\rho(r_1) = 1$ which captures the results of the `vc` consensus event, and $r_2 = \{node(\texttt{Engage})\}$ with $\rho(r_2) = 1$ which captures the initial default values. Algo. 3 expands the initial regions. One such expansion step of region r_1 through edge e from Example 5.1 adds $node(\texttt{Announce})$ to r_1 since the value in the `data` variable is preserved along edge e. A minimal valid BAR that meets conditions (i) through (iv) is $r = \{node(\texttt{Engage}), node(\texttt{Decided}), node(\texttt{Announce}), node(\texttt{LeaderDone}), node(\texttt{ReplicaDone})\}$. This r is the result of merging the expansions of r_1 and r_2 and hence has $\rho(r) = max(\rho(r_1), \rho(r_2)) = 1$.

Example 5.7. Further, imagine a modified version of the Consortium system where every location in Fig. 1 has an extra consensus handler allowing all processes to decide on a set of 10 values. Each process non-deterministically stores one of the agreed-upon values, and then goes to a location `Extra`. In the `Extra` location, let there be a reset handler sending all processes back to `Engage`. In addition to $r = \{node(\texttt{Decided}), node(\texttt{Announce}), node(\texttt{LeaderDone}), node(\texttt{ReplicaDone}), node(\texttt{Engage})\}$ with $\rho(r) = 1$ described in Example 5.6, Algo. 4 will produce *mergedRegions* which also contains a region $r' = r \cup \{node(\texttt{Extra})\}$ with bound $\rho(r') = 10$. While r and r' are both bounded regions for the Consortium system, our algorithm would prefer r due to the lower bound. This ultimately results in a more efficient verification target.

5.3 Discussion: Other Scenarios

Domain Reduction Without Consensus. Consider a system which begins by partitioning a finite number, k, of processes. Let each of those processes receive an arbitrary value from the environment and go to some location ℓ_1. Assuming ℓ_1 can only be reached by these k processes, there can be at most k distinct values held in ℓ_1. If there is another location ℓ_2 which can only be reached by the $n - k$ other processes by receiving one of those k values from a process in ℓ_1, then there can be at most k distinct values held in ℓ_2. Thus, the set $\{\ell_1, \ell_2\}$ is a bounded region with $\rho = k$. Using various values of k, bounded regions with various bounds may arise, even within a single system.

Here, the bounded region created by the k processes represents locations where some k values are distinguished amongst all values held in the system. Therefore, at least k values would be

needed to represent the set of such distinguished values at any one time. Values which are held by the other $n - k$ processes and are only known to the process that holds them do not need to be distinct, so very few values (enough to fill the variables of a single process) are needed to stand in for these remaining values in the system.

Domain Reduction Failure Despite Scalarset Data Use. Consider a system where processes leave the initial location, collectively perform some coordination task, and later receive an arbitrary value from the environment before returning to the initial location.

While the behavior of the system may be correct for the intended coordination task, the fact that processes bring arbitrary values back to the initial location means that the number of values needed to represent the values held in the initial location cannot be bounded. As a result, no suitable bounded region will exist, and domain reduction will not succeed, as the initial location will not be included in any bounded region constructed by our algorithms. Sec. 6 illustrates how the inclusion of the initial location in the bounded region is critical to the validity of our technique.

6 DOMAIN REDUCTION FOR MERCURY SYSTEMS

Having defined domain-reducible verification problems, and how we can recognize them automatically, we now show that when a verification problem $\langle \mathcal{M}_\Delta(n), \Phi \rangle$ is domain-reducible, the unbounded-domain system $\mathcal{M}_\Delta(n)$ satisfies Φ iff the finite-domain system $\mathcal{M}_{\overline{\Delta}}(n)$ satisfies Φ.

Theorem 6.1. *For domain-reducible* $\langle \mathcal{M}_\Delta(n), \Phi \rangle$, $\mathcal{M}_\Delta(n) \models \Phi \Leftrightarrow \mathcal{M}_{\overline{\Delta}}(n) \models \Phi$.

In what follows, we present a proof of Theorem 6.1. Our proof relies on establishing a *backward simulation* relation $\approx$ over pairs of global states in $\mathcal{M}_\Delta(n)$ and $\mathcal{M}_{\overline{\Delta}}(n)$: For all $r \in Q, \overline{r} \in \overline{Q}$, if $r \approx \overline{r}$ and $(q, e, r) \in R$, then there exists $(\overline{q}, \overline{e}, \overline{r}) \in \overline{R}$ such that $q \approx \overline{q}$. This relation enables a proof of Theorem 6.1 by induction on an arbitrary error path in $\mathcal{M}_\Delta(n)$ to show that a related error path exists in $\mathcal{M}_{\overline{\Delta}}(n)$. First, we define a new type of global state transformation. Then, we use this state transformation to define the appropriate relation $\approx$ and prove the theorem.

To simplify presentation, we continue, for now, with our assumption that $\mathcal{M}_\Delta(n)$ contains exactly one variable (and thus, a single data domain δ). Further, we elide the environment process in our examples. The environment is assumed to be a stateless over-approximation of many possible environment processes, and is thus unaffected by our reachability-preserving state transformation. More precisely, the environment process is modeled as a transition system with only a single state, so that all transitions are self-loops on that state. As a result, the state of the environment process is invariant to permutation, and all coordination with the environment is effectively non-blocking.

6.1 Component-Wise Permutations

To relate a global state in Q of $\mathcal{M}_\Delta(n)$ to a global state in $\overline{Q}$ of $\mathcal{M}_{\overline{\Delta}}(n)$, we require a state transformation which can eliminate unnecessary distinct values outside of the bounded region Ψ. Since standard permutations are bijections, which always preserve the number of distinct elements in their domain, they are not sufficient to relate states in Q to states in $\overline{Q}$. Thus, we define a new type of transformation over *global* system states which generalizes permutations for such states so that values in different *local* states may be treated independently.

Definition 6.2 (Component-Wise Permutation). A *component-wise permutation* (CWP) is a sequence of separate δ-permutations which are applied, component-wise, to each local state in a global state.

For a global state $q = (s_1, s_2, \ldots, s_n)$ let $\gamma = (\pi_1, \pi_2, \ldots, \pi_n)$ be a CWP over the values of δ and $\gamma(q)$ be the component-wise application of permutations in γ to local states in q. That is, $\gamma(q) = (\pi_1(s_1), \pi_2(s_2), \ldots, \pi_n(s_n))$. Intuitively, one can think of CWPs as a generalization of permutations

with the expressive power to alter the result of comparison relations between values across processes. In the case where that comparison across processes is equality, a CWP γ may cause values which began as unequal in q to be transformed in $\gamma(q)$ so that they are equal, potentially causing $\gamma(q)$ to include fewer unique values than q, as shown in the following example.

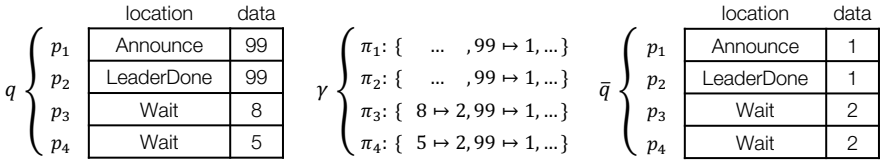


Fig. 3. A CWP.

Example 6.3. An example CWP $\gamma = (\pi_1, \pi_2, \pi_3, \pi_4)$ is shown in Fig. 3. Applying γ to the global state q on the left yields the global state $\overline{q}$ on the right. The local state $q[i]$ of each process p_i is permuted according to π_i. Observe that the larger values in state q are permuted to smaller values in state $\overline{q}$. Further, notice that while there are three distinct values in q, there are only two in $\overline{q}$! This is due to π_3 and π_4 permuting the distinct values 8 and 5 (respectively) to the same value: 2.

6.2 Scalarset Domain Reduction

While, in general, a CWP γ can permute the same value in different components to different permuted values, at certain global states in an execution, equality between some values in different processes is significant (e.g., when the results of consensus are finalized). Hence, component permutations of γ must permute such values *consistently* to preserve their equality (i.e., if such values are equal before applying γ, they should be equal after γ is applied). Further, since our goal is to transform a state q with possibly many values to a state $\overline{q}$ with fewer values, γ should eliminate unnecessary distinct values whenever possible.

Consistent and Minimal CWPs. As stated in Sec. 4.2, a bounded region captures all values whose distinctness across processes affects system transitions. In order to preserve (dis)equality across these values, we first characterize CWPs which permute values in the bounded region consistently.

Definition 6.4 (Consistent CWP). A CWP γ is *consistent* w.r.t. a bounded region Ψ and global state q, denoted $cons(\gamma, q, \Psi)$, iff all component permutations of γ map each value in the bounded region to the *same* permuted value. That is, $cons(\gamma, q, \Psi) \Leftrightarrow \forall i, j, \forall val \in \texttt{proj}_\delta^\Psi(q) : \gamma[i](val) = \gamma[j](val)$.

Then, in order to reduce unnecessary distinct values outside the bounded region, we characterize CWPs γ which bias their mapping of these values toward the smallest values in the domain δ, according to an arbitrary total order (denoted Υ_δ) over δ.

Definition 6.5 (Minimal CWP). A CWP γ is *minimal* w.r.t. a bounded region Ψ and global state q, denoted $min(\gamma, q, \Psi)$, iff each component permutation π_i of γ permutes all values outside of the bounded region to the Υ_δ-smallest *available* δ-values. That is, π_i permutes the j-th Υ_δ-smallest value in $\texttt{proj}_\delta(q[i]) \setminus \texttt{proj}_\delta^\Psi(q)$ to the j-th Υ_δ-smallest value that is not in the image of $\texttt{proj}_\delta^\Psi(q)$ under any component permutation of γ.

Example 6.6. For Fig. 3, consider a bounded region Ψ that contains the location `LeaderDone`. Since local state q[2] has v_{loc} set to `LeaderDone` and holds the value 99 in variable `data`, a CWP γ is consistent w.r.t. q iff all the component permutations of γ permute the value 99 consistently to the same permuted value (e.g., to 1 as done by $\gamma = (\pi_1, \pi_2, \pi_3, \pi_4)$). On the other hand, if π_1 is

changed to permute 99 to, say 2 instead of 1, then the resulting γ is not consistent. Further, assume that Ψ *does not contain* the location `Wait`. The CWP γ in Fig. 3 is minimal since it permutes the values outside of Ψ (8 and 5 in `Wait`) to the smallest available value, 2. If π_3 is changed to permute 8 to, say 3 instead of 2, then the resulting γ would not be minimal, because π_3 could have permuted 8 to a smaller value, 2.

Consistency Sets. We denote by Γ_δ the set of all CWPs over the domain δ. For the remainder of this section, we will focus on CWPs which are both consistent and minimal.

Definition 6.7 (Consistency Set). For a given region Ψ and global state q, the consistency set $\Gamma_\delta^\Psi(q)$ is the set of CWPs which are consistent and minimal w.r.t. Ψ and q. That is, $\Gamma_\delta^\Psi(q) = \{\gamma \in \Gamma_\delta \mid cons(\gamma, q, \Psi) \wedge min(\gamma, q, \Psi)\}$.

Using this definition of the consistency set, we can define a relation between the states of $\mathcal{M}_\Delta(n)$ and $\mathcal{M}_{\overline{\Delta}}(n)$ which will enable the backward simulation we need to prove our domain reduction.

Definition 6.8 (The $\approx_\Psi$ Relation). For a domain δ and bounded region Ψ, let the equivalence relation $\approx_\Psi \subseteq Q \times \overline{Q}$ relate pairs of states $(q, \overline{q})$ such that $\overline{q}$ is equal to the permutation of q according to some CWP γ in the consistency set of q. That is, $\approx_\Psi = \{(q, \overline{q}) \in Q \times \overline{Q} \mid \exists \gamma \in \Gamma_\delta^\Psi(q) : \gamma(q) = \overline{q}\}$.

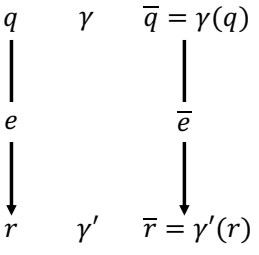
Informally, when two states q and $\overline{q}$ are related via $\approx_\Psi$ (denoted $q \approx_\Psi \overline{q}$), one can think of the CWP γ which relates them as a mechanism to "collapse" superfluous value disqualities across processes in q outside of the bounded region Ψ.

Backward Simulation. Consider the domain-reducible verification problem $\langle \mathcal{M}_\Delta(n), \Phi \rangle$ with $\mathcal{M}_\Delta(n)$ composed of n instances of P_δ and safety specification Φ. Let Ψ be a bounded region with bound ρ as defined earlier. Let $\overline{\delta}$ be a scalarset domain of size $\rho + 1$. We denote by $P_{\overline{\delta}}$, the process obtained from P_δ by replacing all variable and event declarations of type δ with declarations of type $\overline{\delta}$. Let $\mathcal{M}_{\overline{\Delta}}(n)$ be the system composed of n instances of $P_{\overline{\delta}}$. We then show that for each transition in $\mathcal{M}_\Delta(n)$, there is a corresponding transition in $\mathcal{M}_{\overline{\Delta}}(n)$ such that the source and destination states of these two transitions are related by $\approx_\Psi$.

Lemma 6.9. $\forall r \in Q, \overline{r} \in \overline{Q}, (q, e, r) \in R : r \approx_\Psi \overline{r} \implies (\exists (\overline{q}, \overline{e}, \overline{r}) \in \overline{R} : q \approx_\Psi \overline{q})$.

Proof sketch. Since $r \approx_\Psi \overline{r}$, we know there must exist a CWP γ' such that $\gamma'(r) = \overline{r}$. Then, we must identify an appropriate global state $\overline{q}$ and event $\overline{e}$ such that $(\overline{q}, \overline{e}, \overline{r}) \in \overline{R}$ and $q \approx_\Psi \overline{q}$. Since $\overline{q}$ must be related to q by $\approx_\Psi$, we need to show that there exists another CWP $\gamma \in \Gamma_\delta^\Psi(q)$ such that $\gamma(q) = \overline{q}$.

Our overall proof strategy is illustrated by the figure on the right. By identifying a γ and $\overline{e}$ such that $(\gamma(q), \overline{e}, \overline{r}) \in \overline{R}$, the lemma is proven by letting $\overline{q} = \gamma(q)$. We carefully define γ to (i) agree with γ' on values preserved in the transition from q to r so that $\gamma(q)$ may transition to $\overline{r}$, and (ii) permute values in $\mathtt{proj}_\delta^\Psi(q)$ consistently and values outside of $\mathtt{proj}_\delta^\Psi(q)$ to the smallest possible values so that $\gamma \in \Gamma_\delta^\Psi(q)$. More precisely, we detail a procedure $build_\gamma$ below which, given γ', q, and r, constructs the CWP $\gamma = build_\gamma(\gamma', q, r)$ as follows:



(1) Values in the bounded region both in q and r are permuted consistently as in γ': for any two process indices $i, j \in [1, n]$ and value $val \in \delta$, if val is in both $\mathtt{proj}_\delta^\Psi(q[i])$ and $\mathtt{proj}_\delta^\Psi(r[j])$, then for all k let $\gamma[k]$ permute val to the same value as the permuted value of val in $\gamma'[j]$.

(2) Values preserved in the transition from q to r are permuted as in γ': for any two process indices $i, j \in [1, n]$ and variables v, v', if $gets(r[j], v', q[i], v)$, where $gets$ in this case is used

to denote the flow of values between variables in particular *states* rather than locations, then let $\gamma[i]$ permute $q[i](v)$ to the same value as the permuted value of $r[j](v')$ in $\gamma'[j]$.

(3) Values in the bounded region in q but *not* in r are permuted consistently: for any value $val \in \delta$ for which there exists some process index $i \in [1,n]$ such that $val \in \mathtt{proj}_\delta^\Psi(q[i])$ but there is no process index $j \in [1,n]$ such that $val \in \mathtt{proj}_\delta(r[j])$, then for each process index $i \in [1,n]$ let $\gamma[i]$ permute val to the smallest value val' (according to the arbitrary total order Υ_δ) that was not used in any of the previous steps (for any process).

(4) Values not retained in the transition from q to r are permuted minimally: for each process $i \in [1,n]$ and variable $v \in \mathcal{V}$ of type δ, if there does not exist a process $j \in [1,n]$ and variable v' such that $gets(r[j], v', q[i], v)$ let $\gamma[i]$ permute $q[i](v)$ to the smallest value val' (according to Υ_δ) such that no value from steps 1 through 3 was permuted to val' in $q[i]$.

(5) Values not appearing in q are permuted so that each permutation $\gamma[i]$ is a bijection: for each $i \in [1,n]$, permute any remaining values to the next value val' (according to Υ_δ) that was not used in any of the previous steps.

The above construction gives $\gamma = build_\gamma(\gamma', q, r)$ several important properties. These are that (i) γ is a CWP, (ii) γ is consistent and minimal w.r.t. q, (iii) γ maps all values in $\mathtt{proj}_\delta(q)$ into the reduced domain $\overline{\delta}$, and (iv) γ preserves data-flow between q and r so that it is straightforward to determine an event $\overline{e}$ such that $(\gamma(q), \overline{e}, \overline{r})$ is a valid transition. Below, we briefly explain how these properties are obtained.

First, we note that γ is a CWP, because each component $\gamma[i]$ is a permutation. Step (1) fixes only the permutation in $\gamma[i]$ of values which are permuted consistently by all components $\gamma'[j]$. Step (2) fixes the permutation of some values in $\gamma[i]$ which are transmitted in the transition from q to r. The properties of Ψ as a bounded region ensure that these transmitted values are in $\mathtt{proj}_\delta^\Psi(r)$, so all component permutations $\gamma'[j]$ permute them consistently, ensuring no collision is possible with values from step (1). Steps (3) and (4) explicitly avoid permuting to values used in steps which precede them. Step (5) permutes all remaining values to ensure that each $\gamma[i]$ is a bijection.

Additionally, steps (1) and (3) ensure that any values in $\mathtt{proj}_\delta^\Psi(q)$ are permuted consistently, so it holds that $cons(\gamma, q, \Psi)$, and step (4) ensures that each component permutation $\gamma[i]$ permutes all values in $\mathtt{proj}_\delta(q[i]) \setminus \mathtt{proj}_\delta^\Psi(q)$ to the smallest available values, so it holds that $min(\gamma, q, \Psi)$. It then follows immediately that $\gamma \in \Gamma_\delta^\Psi(q)$, and thus $q \approx_\Psi \gamma(q)$.

Further, steps (1) through (4) ensure that all values in $q[i]$ are permuted by $\gamma[i]$ to be within the domain $\overline{\delta}$. In particular, steps (1) and (2) permute values in $q[i]$ to values which appear in some state $\gamma'(r[j]) = \overline{r}[j]$, and steps (3) and (4) permute the remaining values of $q[i]$ to the smallest available values to ensure that only values from $\overline{\delta}$ are used. As such, $\gamma(q) \in \overline{Q}$.

Finally, with the knowledge that $q \approx_\Psi \gamma(q)$ and $\gamma(q) \in \overline{Q}$, it only remains to show that there exists an event $\overline{e}$ such that $(\gamma(q), \overline{e}, \overline{r})$ is a valid transition in $\overline{R}$. This can be achieved by permuting the value carried by e to preserve data flow according to the transition from q to r. If e carries no payload value (e.g. for an internal transition), $\overline{e}$ is simply e. If e carries a payload value, $\overline{e}$ should carry that payload value, permuted to match the permutation (in γ) of a process that receives it.

The construction of γ ensures that all necessary equalities between values both within q and r as well as across q and r are preserved for $\gamma(q)$ and $\overline{r}$, respectively. This ensures that $(\gamma(q), \overline{e}, \overline{r})$ is a valid transition in $\overline{R}$. For the full proof of Lemma 6.9, we refer the reader to App. C.3, which includes a comprehensive case analysis of the various transition types within the Mercury semantics.

Proving Theorem 6.1 Using Backward Simulation. Recall that Theorem 6.1 states that for domain-reducible $\langle \mathcal{M}_\Delta(n), \Phi \rangle$, the system $\mathcal{M}_\Delta(n)$ is safe iff $\mathcal{M}_{\overline{\Delta}}(n)$ is safe. Equivalently, there

exists an execution η of $\mathcal{M}_\Delta(n)$ that ends in an error state iff there exists an execution $\overline{\eta}$ of $\mathcal{M}_{\overline{\Delta}}(n)$ that ends in an error state.

Having proved Lemma 6.9, we can prove the theorem by considering an arbitrary execution η of $\mathcal{M}_\Delta(n)$ from some initial state $q_0 \in Q_0$ to an (erroneous) final state $q_x \in Q$. Observe that a corresponding (erroneous) state $\overline{q_x} = \gamma'(q_x) \in \overline{Q}$ can be identified by constructing γ' according to the $build_\gamma$ procedure with q_x in place of q and assuming that r contains no δ-values (because no state follows q_x in η). By construction, $q_x \approx_\Psi \overline{q_x}$. Then, for each transition in the execution, beginning with (q_{x-1}, e, q_x), by Lemma 6.9, there must exist a transition $(\overline{q_{x-1}}, \overline{e}, \overline{q_x}) \in \overline{R}$ with source state $\overline{q_{x-1}}$ such that $q_{x-1} \approx_\Psi \overline{q_{x-1}}$. Repeating this process for each subsequent q_{x-k} and $\overline{q_{x-k}}$ constructs an execution $\overline{\eta}$ from initial state $\overline{q_0}$ to a final state $\overline{q_x}$ where all states appearing in $\overline{\eta}$ are in $\overline{Q}$ and $q_i \approx_\Psi \overline{q_i}$ for each $i \in [0, x]$. For an example of this process, see App. B.

Finally, as any bounded region Ψ such that $q_i \approx_\Psi \overline{q_i}$ for specification Φ must include the initial location and any error locations identified by Φ, we draw two conclusions. First, since the value held by all processes in the initial location must be permuted consistently for all processes, and $\mathcal{M}_\Delta(n)$ exhibits value symmetry, $\overline{q_0}$ must be in $\overline{Q_0}$. Second, because all values held in error locations are permuted consistently, $\overline{q_x}$ is an error state iff q_x is an error state. Therefore, there is a execution $\overline{\eta}$ of $\mathcal{M}_{\overline{\Delta}}(n)$ ending in an error state iff there is a execution η of $\mathcal{M}_\Delta(n)$ ending in an error state. □

Example 6.10. Fig. 4 illustrates the above process on a segment of an execution of the Consortium system. Here, the bounded region $\Psi = \{$`Engage`, `Decided`, `Announce`, `LeaderDone`, `ReplicaDone`$\}$, as determined in Example 5.6, and all locations in Ψ are shaded in gray. The left side of the figure shows the execution $q_1, \ldots, q_5$ in the unbounded system $\mathcal{M}_\Delta(n)$, while the right side of the figure shows the corresponding execution $\overline{q_1}, \ldots, \overline{q_5}$ in the bounded system $\mathcal{M}_{\overline{\Delta}}(n)$. Between each pair of corresponding states q_i and $\overline{q_i}$ is the CWP γ_i which relates them such that $\overline{q_i} = \gamma_i(q_i)$.

Beginning with q_5, we explain how the CWP γ_5 (which relates q_5 with the corresponding state $\overline{q_5}$) is constructed according to the procedure(s) above. Since the value 99 is in Ψ in state q_5, but no state follows q_5, 99 is permuted consistently to the smallest available value, 1, (as in step (3) of $build_\gamma$) by all component permutations of γ_5. As in step (5), all other values are permuted so each component permutation $\gamma_5[i]$ is a bijection. Similarly, the payload of event `inform(99)` is permuted to `inform(1)`. Then, each CWP γ_i for $i \in [1, 4]$ is constructed according to $build_\gamma$.

First, to construct γ_4, step (1) permutes 99 to 1 in all components $\gamma_4[i]$; steps (2) and (3) have no effect; step (4) permutes 8 to 2 in $\gamma_4[3]$ and 5 to 2 in $\gamma_4[4]$; and step (5) completes each permutation $\gamma_4[i]$. The `share` event has no payload, so it is used as-is in a valid transition from $\overline{q_3}$ to $\overline{q_4}$.

Second, to construct γ_3, step (1) permutes 99 to 1 in all components $\gamma_3[i]$; step (2) permutes 8 to 2 in $\gamma_3[3]$ and 5 to 2 in $\gamma_3[4]$; steps (3) and (4) have no effect; and step (5) completes each permutation $\gamma_3[i]$. The payload value of the `vc(99)` event is received by processes p_1 and p_2 in q_3, so the event payload, 99, is permuted to 1 according to $\gamma_3[1]$, yielding the permuted event $\overline{e} =$ `vc(1)`.

Third, to construct γ_2, step (1) has no effect; step (2) permutes 99 to 1 in $\gamma_2[1]$, 8 to 2 in $\gamma_2[3]$, and 5 to 2 in $\gamma_2[4]$; step (3) has no effect; step (4) permutes 17 to the smallest available value, 1, in $\gamma_2[2]$; and step (5) completes each permutation $\gamma_2[i]$. The payload value of the `influence(17)` event is received by process p_2 in q_2, so the event payload, 17, is permuted to 1 according to $\gamma_3[2]$, yielding the permuted event $\overline{e} =$ `influence(1)`.

Finally, to construct γ_1, step (1) has no effect; step (2) permutes 99 to 1 in $\gamma_1[1]$, 8 to 2 in $\gamma_1[3]$, and 5 to 2 in $\gamma_1[4]$; step (3) has no effect; step (4) permutes 1 to the smallest available value, 1, in $\gamma_1[2]$; and step (5) completes each permutation $\gamma_1[i]$.

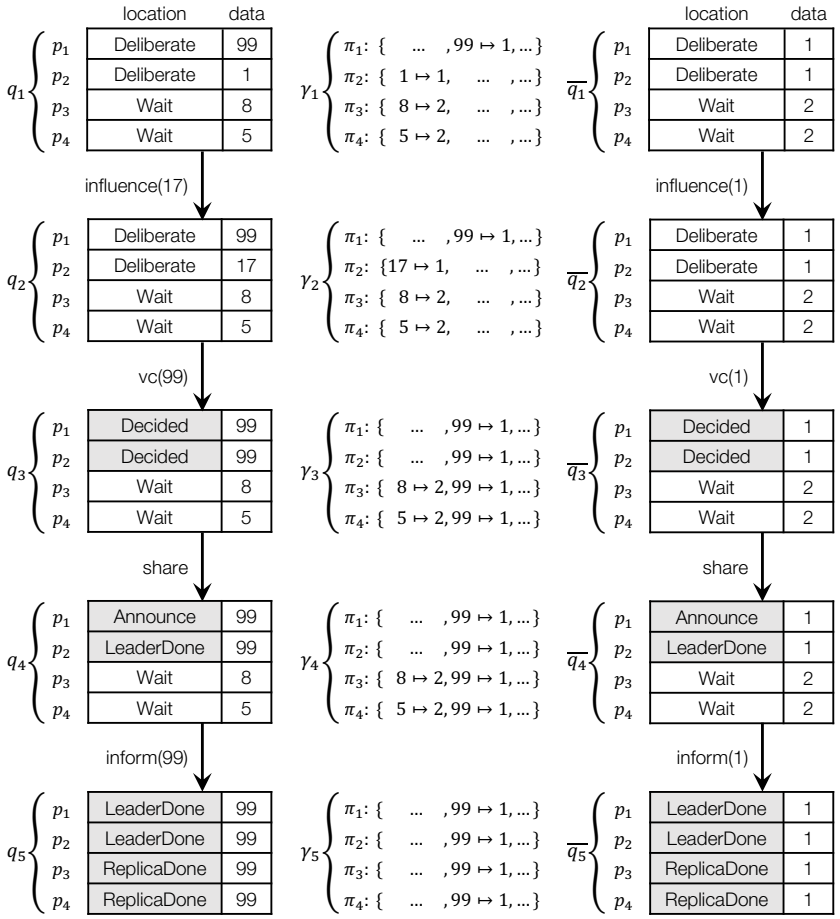


Fig. 4. Inductive application of $build_\gamma$ to an execution of $\mathcal{M}_\Delta(n)$.

7 DOMAIN REDUCTION FOR MULTI-VARIABLE AND MULTI-DOMAIN SYSTEMS

To simplify presentation, we assumed throughout Sections 4, 5, and 6 that $\mathcal{M}_\Delta(n)$ contained only a single variable, and thus, a single scalarset domain δ. In this section, we discuss how the above results extend to systems involving multiple variables. In Sec. 7.1, we address systems with multiple variables of the same unbounded scalarset domain. In Sec. 7.2, we discuss systems involving multiple unbounded scalarset domains.

7.1 Multi-Variable Systems

Regions. First, we redefine a *region* of a process P (Sec. 4.2) to be a set $\Psi \subseteq S \times \mathcal{V}$ of pairs of process-local states and variables, rather than a set of locations. Thus, we generalize our notion of a *process* being in a bounded region Ψ to a notion of a process holding *variables* in Ψ, while the process holding those variables may also hold some variables which are not in Ψ.

The definition of a value-stable region is augmented to include that there must exist a finite upper bound, λ, over all reachable global states q, on the number of distinct δ-values that are held by any *individual* process and are *not* in $\texttt{proj}_\delta^\Psi(q)$. The bound λ may be determined by various means, but one simple method is simply to count the number of variables in the process definition (as no

process can hold at any time more values than it has variables). For example, in the Consortium system (Fig. 1), λ is 1 since the `data` variable may appear in locations outside of its bounded region (e.g., in the `Wait` location).

The size of the domain cutoff, then is $\rho + \lambda$, rather than $\rho + 1$. Note that in the case of the single-variable assumption in the previous sections, these two formulas are equivalent.

Similarly, the definition of a bounded region Ψ is adjusted to ensure, as in the single-variable case, that system transitions only depend only the (dis)equality of δ-values across processes when those values are in the bounded region in the state where said (dis)equality is enforced by the transition relation. More precisely, in addition to value-stability, a bounded region Ψ must include every state-variable pair $(s, v) \in S \times \mathcal{V}$ such that (i) s is an initial local state, (ii) s is a destination local state of a consensus action, (iii) s is a source state of a broadcast action which takes its payload value from v, or (iv) s and v both appear together in the global specification Φ.

Determining Bounded Regions. When determining bounded regions, the construction of the LTS described in Sec. 5 changes slightly. Rather than each node x representing a location ℓ in program P, each node represents a set $[x]$ of local states with $v_{loc} = \ell$. Each of these nodes may be augmented with a Boolean predicate b to refine the set of states represented by x to be $[x] = \{s \mid s(v_{loc}) = loc(x) \wedge s \models b\}$. Then, an abstract region r is set of node-variable pairs (i.e., $r \subseteq L \times \mathcal{V}$) such that each node-variable pair (x, v) represents the set $[(x, v)] = \{(s, v) \mid s \in [x]\}$ of state-variable pairs. Let $[r]$ be the concrete region $\{(s, v) \in [(x, v)] \mid (x, v) \in r\}$ that the abstract region r represents. As before, we say an abstract region r is value-stable (resp. bounded) iff the corresponding region $[r]$ is, and the bounds $\rho(r)$ and $\lambda(r)$ are equal to the bounds ρ and λ of $[r]$.

Determining a suitable abstract bounded region proceeds as before, with a few alterations to account for refining bounded regions from sets of program locations to sets of state-variable pairs.

The three types of initial regions are modified as follows. For the first type, composed of all the nodes corresponding to locations where some **`consensus`** action terminates, each such node is paired with the variable holding the values agreed upon by consensus. For the second type, composed of a single node representing a location which can only be occupied by a single process at a time, we create an ABR consisting of this node paired with every variable in the process definition. The bound ρ of each of these ABRs is the number of variables of type δ, as one process can only hold as many unique δ-values as it has δ-variables. Finally, for the ABR encoding the initial location, pair each node with every variable of type δ, as all variables are assumed to begin with the same default value; the bound ρ for this region remains 1.

As before, initial regions are expanded using the *gets* relation. When the variable v may *only* get its value in a location $loc(x)$ from a variable v' in location $loc(x')$, where (x', v') is in some ABR r, the pair (x, v) is added to r (instead of just the node x).

When merging expanded regions, the definition of when two regions are mutually exclusive is relaxed along with the definition of regions. Two ABRs r_1 and r_2 are considered mutually exclusive if it can be shown that processes may hold data which is in r_1 only when no process may hold data in r_2 (and vice versa). Mutually exclusive regions are merged by taking the union of their state-variable pairs and the maximum of their associated ρ bounds.

Then, after constructing the set of merged ABRs, we select the one with the smallest bound ρ which satisfies the necessary conditions to represent a bounded region.

Scalarset Domain Reduction. While in Sec. 6, we assumed that $\mathcal{M}_\Delta(n)$ contained only one variable of type δ to simply presentation, we emphasize that the proofs of Lemma 6.9 and Theorem 6.1 are written in a way that is agnostic to this assumption. Thus, our results hold under the assumption of a process definition P_δ with an arbitrary number of variables of the unbounded domain δ.

7.2 Multi-Domain Systems

We focused on the single reduction $\mathcal{M}_{\Delta}(n) \xrightarrow{\delta} \mathcal{M}_{\bar{\Delta}}(n)$ with respect to unbounded scalarset domain $\delta \in \Delta$. Our results extend immediately to the case with multiple unbounded data domains via a sequence of reductions w.r.t. each $\delta \in \Delta$: $\mathcal{M}_{\Delta}(n) \xrightarrow{\delta_1} \mathcal{M}_{\Delta'}(n) \xrightarrow{\delta_2} \ldots \xrightarrow{\delta_m} \mathcal{M}_{\bar{\Delta}}(n)$.

8 EVALUATION

In this section, we present the implementation of our technique and evaluate it on a set of doubly-unbounded DAB systems.

Venus. We develop a tool, Venus, for verification of doubly-unbounded DAB systems that combines our domain reduction with a recent tool, QuickSilver, for parameterized verification of DAB systems with finite-state processes. As explained in Sec. 4, Venus yields a program with finite data domains, paving the way for QuickSilver to perform its reasoning.

We note that while each expansion step in Algo. 3 can only expand a region with a single node-variable pair, in Venus, the implementation of the region expansion procedure is more involved. We generalize an expansion step to expand a region with a strongly-connected set of node-variable pairs, even when its pairs cannot be used individually. Further, we emphasize that Venus automatically identifies data domains which can be treated as scalarsets.

Case Studies. Here, we demonstrate the efficacy of our technique on three case studies which represent commonly used DAB systems: Consortium, Distributed Register, and a Key-Value Store.

Consortium was introduced in Sec. 2 along with its safety property. We now introduce three variants of this system. The first variant, *Consortium-Three*, elects a consortium of size three instead of two to showcase that the number of processes in the bounded region is independent from the bound on the number of unique values there. The second variant, *Consortium-BCast*, does not elect a particular trusted actor to announce the decision, but rather allows either elected actor to perform a broadcast of the decided value to the rest of the system to illustrate that the domain cutoff computation is not sensitive to coordination within the bounded region. The third variant, *Consortium-Check*, forces both of the trusted actors to share the decided value with the rest of the system, and allows the rest of the actors to check, using a local assertion, if the shared values are identical, and if not, move to an error state. This variant illustrates that domain reduction can be applied to programs with local assertions.

Distributed Register is based on Atomix's AtomicValue [Atomix 2021] which gives a consistent view of some stored value under concurrent updates. In Distributed Register, clients in the environment submit requests to read from and update a register. Processes service read requests from their local copy of the register. For update requests, processes use consensus to determine a consistent value to be stored in the shared register. The safety property is that any two processes that are in a location where they serve client read requests always have equal values in their local copy of the register. We define a variant of this system, called *DistReg-Two*, that allows two registers to be manipulated simultaneously; we construct this variant to help demonstrate that multiple variable domains can be reduced in sequence.

Our Key-Value Store is a simple, dictionary-based, distributed key-value store inspired by [Redis 2021]. Clients in the environment submit requests to store a value, or read data stored, under a given key. When handling update requests for a given key, processes use consensus to determine a consistent value to be stored under that key. The safety property is that any two processes in a location where they serve a client read request for some key must have equal values stored under that key. To enable verification with variables with dictionary datatypes, we employ a lightweight *shape-reduction* to first reduce verification to that of a system where each process has a finite

number of integer domains. Intuitively, this is possible when the system's transitions on events indexed by some key are effectively independent of transitions and data indexed by other keys. We then apply our domain cutoff reduction which is the primary focus of this paper. This layered application of shape reduction, domain reduction, and finally process cutoff reduction demonstrates how domain reduction can be composed with other reduction techniques to verify systems which handle more complex datatypes than simple scalarset variables.

For each case study *CS*, we also define a version *CS-32Bit*, with 32-bit integer domains, to highlight that domain cutoffs can be applied to large, finite domains as well as unbounded domains.

Evaluation. We now discuss the result of evaluating our case studies in Venus. All experiments are performed on a Dell Latitude 7400 with Intel Core i7 CPU at 1.9 GHz and 16 GB of RAM.

Table 1. Performance of Venus.

Benchmark	LoC	Domain Size	Domain Cutoff	Process Cutoff	Time(s)
Consortium	62	∞	3	3	0.249 ± 0.003
Consortium-Three	62	∞	3	4	0.261 ± 0.012
Consortium-BCast	58	∞	3	3	0.218 ± 0.012
Consortium-Check	68	∞	3	5	0.312 ± 0.007
Consortium-32Bit	62	2^{32}	3	3	0.254 ± 0.010
DistReg	34	∞	2	2	0.220 ± 0.025
DistReg-Two	79	∞	2	2	1.620 ± 0.056
DistReg-32Bit	34	2^{32}	2	2	0.202 ± 0.003
KeyValStore	50	∞	2	2	0.205 ± 0.003
KeyValStore-32Bit	50	2^{32}	2	2	0.213 ± 0.014

The performance of Venus is shown in Table 1. For each benchmark, we provide the number of lines of code needed to model the benchmark in Mercury, the initial size of the data domain (marked ∞ if unbounded), the domain cutoff computed by Venus, the process cutoff used for verification[4], and the mean run time for 10 verification runs with the 95% confidence intervals.

Venus successfully reduces both large and unbounded data domains to relatively small, bounded domains (2 and 3). Note that the domain cutoff remains constant across all variations of a given benchmark. This illustrates that our domain cutoff computation is robust to control flow changes. Also, we point out that for the DistReg-Two benchmark, Venus identifies two unbounded data domains (corresponding to the two registers) and computes domain cutoffs for each independently, even though the variables storing the register values are both of type `int`. Finally, Venus is able to verify each of our benchmarks rather efficiently in under 2 seconds. While it is still true that the complexity of model checking finite systems is exponential in both the number of processes in the system and the size of each process's state space, Venus is able to find bounds for these systems which are small enough to enable *efficient* automated verification.

Intuition for Designing Domain-Reducible Systems. In practice, we find that localized data saturation often arises when the local/global knowledge of values in each variable is reflected in the processes' control locations. That is, when processes reach a location with a value in some variable

[4]While Venus is able to compute a domain cutoff for DistReg-Two, QuickSilver is unable to compute a process cutoff. Because Venus can potentially be combined with tools beyond QuickSilver for computing process cutoffs, we make an exception for this case by manually computing a process cutoff of 2.

x which has been communicated with all other processes, the process definition should prevent them from reaching that location again with a value in x which has not been shared globally.

9 RELATED WORK

The formal methods community has developed a variety of techniques for verifying infinite-state concurrent/distributed systems. At the coarsest level, these approaches can be divided into semi-automated [Krogh-Jespersen et al. 2020; Ma et al. 2019; Padon et al. 2017, 2016; Taube et al. 2018; v. Gleissenthall et al. 2019] and fully-automated [Abdulla et al. 2016, 1996, 2007; Alur et al. 2015; Aminof et al. 2018; Außerlechner et al. 2016; Bloem et al. 2015; Burch et al. 1992; Clarke et al. 1989, 1998, 2006; Emerson et al. 2000; Emerson and Kahlon 2000, 2003; Emerson and Sistla 1997; Emerson and Wahl 2003; Finkel and Schnoebelen 2001; Ip and Dill 1996; Jaber et al. 2020, 2021; Jacobs and Sakr 2018; Kaiser et al. 2010; Lubachevsky 1984; Marić et al. 2017; Schmitz and Schnoebelen 2013] techniques. Semi-automated verification typically involves manually discovering inductive invariants [Padon et al. 2016; Wilcox et al. 2015]. Fully automated verification (the focus of this paper) requires no user intervention and is challenging when applied to unbounded systems.

Symbolic representations and reduction strategies are typically employed to scale or enable automated verification for systems with large or infinite state spaces. For instance, symmetry reduction [Alur et al. 2015; Clarke et al. 1998; Emerson et al. 2000; Emerson and Sistla 1997; Emerson and Wahl 2003; Ip and Dill 1996] is used to explore the state space of a system via a reduced quotient structure. Automated verification of distributed systems is further often complicated by an unbounded number of processes, infinite data domains, or both. A popular approach to handling systems with an unbounded number of processes is to use (process) cutoffs [Abdulla et al. 2016; Aminof et al. 2018; Außerlechner et al. 2016; Bloem et al. 2015; Clarke et al. 2006; Emerson and Kahlon 2000, 2003; Jaber et al. 2020, 2021; Jacobs and Sakr 2018; Kaiser et al. 2010; Marić et al. 2017]. Another approach for dealing with an unbounded number of processes and/or unbounded variable domains is to impose a well-order [Finkel and Schnoebelen 2001; Schmitz and Schnoebelen 2013] over the state space of the system.

In what follows, we briefly explore prior work on verification of distributed systems with unbounded processes or infinite process state spaces.

Bounded Number of Processes, Unbounded Process States. One inspiration for this paper is the notion of *data saturation*, first broached by Lubachevsky [1984] and explored further by Ip and Dill [1996]. Infinite state spaces are also handled in well-ordering based frameworks [Abdulla et al. 1996] or using "temporal case splitting" wherein each case identifies a particular value and all other values are represented with a symbolic constant [Clarke et al. 1989]. None of these approaches target doubly-unbounded systems.

Unbounded Number of Processes, Bounded Process States. The two common approaches for parameterized verification of finite-state processes are based on well-ordering [Esparza et al. 1999; Finkel and Schnoebelen 2001; Schmitz and Schnoebelen 2013] and cutoff results computed either statically [Aminof et al. 2018; Außerlechner et al. 2016; Clarke et al. 2006; Emerson and Kahlon 2000, 2003; Jaber et al. 2020; Jacobs and Sakr 2018; Marić et al. 2017] or dynamically [Abdulla et al. 2016; Kaiser et al. 2010]. Notably, many of these cutoff results can potentially be combined with our domain reduction approach to enable verification of doubly-unbounded systems.

Doubly-Unbounded Systems. A related effort [Abdulla et al. 2007] tackles parameterized verification of infinite-state systems using a well-ordering over global states and performing a backward-reachability analysis. Our approach differs from this effort in the target distributed systems and their models, as well as the technical approach. The work of Abdulla et al. [2007] targets generic distributed systems that are modeled using the more-traditional system model where processes are

defined as extended finite-state automata with local variables and global conditions on transitions. On the other hand, we target modularly-designed DAB systems built with abstractions of agreement protocols and, in particular, modeled in an *easy-to-use* modeling language Mercury. Further, the reduction presented by Abdulla et al. [2007] does not require systems to be value-symmetric and tackles the unboundedness of *both* the number of processes and the process state space in one *consolidated* step. In contrast, while the reduction in our approach requires the system to be value-symmetric, it is *separable*, and, arguably, more *flexible*. Specifically, our approach relies on two separate reductions, one for the process state spaces and one for the number of processes, each of which can potentially be replaced with other reductions.

Clarke et al. [2006] combine predicate and counter abstractions into an "environment abstraction" to verify doubly-unbounded systems. The environment abstraction designates one process as the "reference" process and models all other processes in relation to it. Our work differs from their approach in two ways: (i) they do not verify systems which incorporate abstractions of agreement protocols, and (ii) they capture the relationships between values in the system using predicates describing the environment of the reference process, while we capture such relationships by determining bounded regions.

Marić et al. [2017] reduce parameterized verification of consensus algorithms over inputs from infinite domains to parameterized verification of these algorithms over binary inputs. They provide a process cutoff to reduce the problem to finite verification of consensus algorithms over binary inputs. Aside from focusing on consensus algorithms (as opposed to systems built on top of them), their work also differs from ours in that their reduction of infinite domains to finite ones relies on the zero-one principle for sorting networks rather than a static analysis of the system being verified, like our domain cutoff computation.

10 CONCLUSION

In this paper, we have characterized a class of domain-reducible safety verification problems for DAB systems with an unbounded number of infinite-state processes. We presented algorithms for identifying evidence that a given verification problem is domain-reducible and calculating a suitable reduced cutoff domain to yield a simplified verification problem. Further, we also provided proof that this evidence is sufficient to imply that successful verification of the simplified system implies verification of the original system. Our domain reduction can be *composed* with process cutoff reductions to enable fully-automated verification of doubly-unbounded DAB systems.

While our domain reduction cannot be applied to verification problems which are not domain-reducible (e.g. to systems which treat data asymmetrically), the composability of reduction techniques for different dimensions of unboundedness in a distributed system opens up interesting avenues for further research. As with our scalarset domain reduction, the addition of a reduction technique for one dimension of unboundedness advances the utility of reduction techniques for others. For example, in ongoing work, we are developing process cutoffs for client-server-based DAB systems. The domain reduction in this paper can be easily composed with this new process cutoff reduction to enable automated verification of a new class of doubly-unbounded DAB systems.

Composable reductions also pave the way for tackling other dimensions of unboundedness, *one dimension at a time*. For instance, we can envision that domain and process reductions can be composed with reductions for verification of systems with asynchronous, unbounded channels to verification of systems with bounded channels, thereby greatly expanding the classes of DAB systems and communication models that can be tackled using finite-state model checking.

ACKNOWLEDGMENTS

This work was partially funded by NSF grant 1846327 and Amazon Science.

REFERENCES

Parosh Abdulla, Frederic Haziza, and Lukavs Holik. 2016. Parameterized Verification Through View Abstraction. *International Journal on Software Tools for Technology Transfer* 18, 5 (2016), 495–516. https://doi.org/10.1007/s10009-015-0406-x

P. A. Abdulla, K. Cerans, B. Jonsson, and Yih-Kuen Tsay. 1996. General Decidability Theorems for Infinite-State Systems. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*. IEEE Computer Society, USA, 313. https://dl.acm.org/doi/10.5555/788018.788796

Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. 2007. Parameterized Verification of Infinite-State Processes with Global Conditions. https://doi.org/10.1007/978-3-540-73368-3_17

Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. 2015. Automatic Completion of Distributed Protocols with Symmetry. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 395–412. https://doi.org/10.1007/978-3-319-21668-3_23

Benjamin Aminof, Tomer Kotek, Sasha Rubin, Francesco Spegni, and Helmut Veith. 2018. Parameterized model checking of rendezvous systems. *Distributed Computing* 31, 3 (2018), 187–222. https://doi.org/10.1007/s00446-017-0302-6

Zachary Amsden, Ramnik Arora, Shehar Bano, Mathieu Baudet, Sam Blackshear, Abhay Bothra, George Cabrera andChristian Catalini, Konstantinos Chalkias, Evan Cheng, Avery Ching, Andrey Chursin, George Danezis andGerardo Di Giacomo, David L. Dill, Hui Ding, Nick Doudchenko, Victor Gao, Zhenhuan Gao, François Garillot, Michael Gorven, Philip Hayes, J. Mark Hou, Yuxuan Hu, Kevin Hurley, Kevin Lewi, Chunqi Li, Zekun Li, Dahlia Malkhi andSonia Margulis, Ben Maurer, Payman Mohassel, Ladi de Naurois, Valeria Nikolaenko, Todd Nowacki, Oleksandr Orlov andDmitri Perelman, Alistair Pott, Brett Proctor, Shaz Qadeer, Rain, Dario Russi, Bryan Schwab, Stephane Sezer, Alberto Sonnino, Herman Venter, Lei Wei, Nils Wernerfelt, Brandon Williams, Qinfan Wu, Xifan Yan, Tim Zakian, and Runtian Zhou. 2020. *The Libra Blockchain*. Technical Report. https://developers.libra.org/docs/assets/papers/the-libra-blockchain/2020-05-26.pdf

Atomix. 2021. Atomix. (2021). https://atomix.io/docs/latest/user-manual/primitives/AtomicValue/

Simon Außerlechner, Swen Jacobs, and Ayrat Khalimov. 2016. Tight Cutoffs for Guarded Protocols with Fairness. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.), Vol. 9583. Springer, 476–494. https://doi.org/10.1007/978-3-662-49122-5_23

Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. 2015. *Decidability of Parameterized Verification*. Morgan & Claypool Publishers. https://doi.org/10.1145/2951860.2951873

J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. 1992. Symbolic Model Checking: 10²⁰ States and Beyond. *Inf. Comput.* 98, 2 (jun 1992), 142–170. https://doi.org/10.1016/0890-5401(92)90017-A

E. Clarke, D. Long, and K. McMillan. 1989. Compositional Model Checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE Press, 353–362. https://dl.acm.org/doi/abs/10.5555/77350.77387

Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. 1998. Symmetry Reductions i n Model Checking. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98)*. Springer-Verlag, Berlin, Heidelberg, 147–158. https://doi.org/10.1007/BFb0028741

Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. 2006. Environment Abstraction for Parameterized Verification. In *VMCAI (Lecture Notes in Computer Science)*, Vol. 3855. Springer, 126–141. https://doi.org/10.1007/11609773_9

E. Allen Emerson, John W. Havlicek, and Richard J. Trefler. 2000. Virtual Symmetry Reduction. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS '00)*. IEEE Computer Society, USA, 121. https://dl.acm.org/doi/abs/10.5555/788022.788994

E. Allen Emerson and Vineet Kahlon. 2000. Reducing Model Checking of the Many to the Few. In *CADE (Lecture Notes in Computer Science)*, David A. McAllester (Ed.), Vol. 1831. Springer, 236–254. https://doi.org/10.1007/10721959_19

E. Allen Emerson and Vineet Kahlon. 2003. Exact and Efficient Verification of Parameterized Cache Coherence Protocols. In *CHARME (Lecture Notes in Computer Science)*, Vol. 2860. Springer, 247–262. https://doi.org/10.1007/978-3-540-39724-3_22

E. A. Emerson and A. P. Sistla. 1997. Utilizing Symmetry When Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. *ACM Trans. Program. Lang. Syst.* 19, 4 (jul 1997), 617–638. https://doi.org/10.1145/262004.262008

E. Allen Emerson and Thomas Wahl. 2003. On Combining Symmetry Reduction and Symbolic Representation for Efficient Model Checking. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 216–230. https://doi.org/10.1007/978-3-540-39724-3_20

Javier Esparza, Alain Finkel, and Richard Mayr. 1999. On the Verification of Broadcast Protocols. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 352–359. https://doi.org/10.1109/LICS.1999.782630

Alain Finkel and Philippe Schnoebelen. 2001. Well-structured Transition Systems Everywhere! *Theor. Comput. Sci.* 256, 1-2 (2001), 63–92. https://doi.org/10.1016/S0304-3975(00)00102-X

Jeremiah Griffin, Mohsen Lesani, Narges Shadab, and Xizhe Yin. 2020. TLC: Temporal Logic of Distributed Components. *Proc. ACM Program. Lang.* 4, ICFP, Article 123 (Aug. 2020), 30 pages. https://doi.org/10.1145/3409005

Hyperledger. 2021. The Hyperledger Project. (2021). https://www.hyperledger.org/

C Norris Ip and David L Dill. 1996. Better Verification Through Symmetry. *Formal methods in system design* 9, 1-2 (1996), 41–75. https://doi.org/10.1007/BF00625968

Nouraldin Jaber, Swen Jacobs, Christopher Wagner, Milind Kulkarni, and Roopsha Samanta. 2020. Parameterized Verification of Systems with Global Synchronization and Guards. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 299–323. https://doi.org/10.1007/978-3-030-53288-8_15

Nouraldin Jaber, Christopher Wagner, Swen Jacobs, Milind Kulkarni, and Roopsha Samanta. 2021. QuickSilver: Modeling and Parameterized Verification for Distributed Agreement-Based Systems. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 157 (oct 2021), 31 pages. https://doi.org/10.1145/3485534

Swen Jacobs and Mouhammad Sakr. 2018. Analyzing Guarded Protocols: Better Cutoffs, More Systems, More Expressivity. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science)*, Isil Dillig and Jens Palsberg (Eds.), Vol. 10747. Springer, 247–268. https://doi.org/10.1007/978-3-319-73721-8_12

Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2010. Dynamic Cutoff Detection in Parameterized Concurrent Programs. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.), Vol. 6174. Springer, 645–659. https://doi.org/10.1007/978-3-642-14295-6_55

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

Boris D. Lubachevsky. 1984. An Approach to Automating the Verification of Compact Parallel Coordination Programs. I. *Acta Inf.* 21, 2 (aug 1984), 125–169. https://doi.org/10.1007/BF00289237

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 370–384. https://doi.org/10.1145/3341301.3359651

Ognjen Marić, Christoph Sprenger, and David Basin. 2017. Cutoff Bounds for Consensus Algorithms. In *International Conference on Computer Aided Verification*. Springer, 217–237. https://doi.org/10.1007/978-3-319-63390-9_12

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos Made EPR: Decidable Reasoning about Distributed Protocols. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 108 (Oct. 2017), 31 pages. https://doi.org/10.1145/3140568

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 614–630. https://doi.org/10.1145/2908080.2908118

Redis. 2021. Redis. (2021). https://redis.io/

Sylvain Schmitz and Philippe Schnoebelen. 2013. The Power of Well-Structured Systems. In *CONCUR 2013 (Lecture Notes in Computer Science)*, Pedro R. D'Argenio and Hernán C. Melgratti (Eds.), Vol. 8052. Springer, 5–24. https://doi.org/10.1007/978-3-642-40184-8_2

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158116

Ichiro Suzuki. 1988. Proving Properties of a Ring of Finite-State Machines. *Inf. Process. Lett.* 28, 4 (July 1988), 213–214. https://doi.org/10.1016/0020-0190(88)90211-6

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 662–677. https://doi.org/10.1145/3192366.3192414

Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend Synchrony: Synchronous Verification of Asynchronous Distributed Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 59 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290372

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958

Received 2022-10-28; accepted 2023-02-25