Applying Formal Methods to Distributed Algorithms Using Local-Global Relations

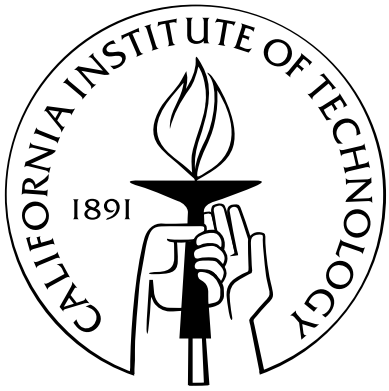
Thesis by

Jerome White

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

2011

(Defended December 6, 2010)

To Grandaddy.

Contents

Acknowledgements

I would first like to thank Mani, who inspired this work and allowed me the freedom to develop it. More significantly, however, there were times during the pursuit of this degree when I was unsure that I could obtain it. He always managed to convince me otherwise—something for which I am truly grateful.

This thesis would not have been possible without the help of my Infospheres labmates, especially Sayan and Concetta. Sayan introduced me to PVS, while Concetta helped to shape much of the work overall. Good scientific discoveries usually require the exploration of several bad ideas. This thesis was no exception. Concetta patiently sat through just about all of my bad ideas, time and again helping me see the error in my ways. Her insight amazes me just as much as her patience.

I would also like to thank and remember Brian, who had a significant impact on this project during its early stages. He was an incredible student and a wonderful person. I wish he could have seen the culmination of this work.

The department as a whole has been like an extended family; the faculty and staff have created a wonderful environment in which to learn as a student and to grow as a person. Maria and Mathieu, in particular, were both constant sources of encouragement, helping to make my life overall more enjoyable. Every graduate student should have people like them in their corner.

Finally, special thanks go to my parents. They instilled in me the importance of education, of attention to detail, and of giving my best. This degree is one of the many results of those values.

Abstract

This thesis deals with the design and analysis of distributed systems in which homogeneous, autonomous agents collaborate to achieve a common goal. The class of problems studied includes consensus algorithms in which all agents eventually come to an agreement about a specific action. The thesis proposes a framework, called local-global, for analyzing these systems. A local interaction is an interaction among subsets of agents, while a global interaction is one among all agents in the system. Global interactions, in practice, are rare, yet they are the basis by which correctness of a system is measured. For example, if the problem is to compute the average of a measurement made separately by each agent, and all the agents in the system could exchange values in a single action, then the solution is straightforward: each agent gets the values of all others and computes the average independently. However, if the system consists of a large number of agents with unreliable communication, this scenario is highly unlikely. Thus, the design challenge is to ensure that sequences of local interactions lead, or converge, to the same state as a global interaction.

The local-global framework addresses this challenge by describing each local interaction as if were a global one, encompassing all agents within the system. This thesis outlines the concept in detail, using it to design algorithms, prove their correctness, and ultimately develop executable implementations that are reliable. To this end, the tools of formal methods are employed: algorithms are modeled, and mechanically checked, within the PVS theorem prover; programs are also verified using the Spin model checker; and interface specification languages are used to ensure local-global properties are still maintained within Java and C# implementations. The thesis presents example applications of the framework and discusses a class of problems to which the framework can be applied.

Chapter 1

Introduction

1.1 Motivation

1.1.1 Multi-Agent Systems Operating in Unreliable Environments

Errors in systems can have disastrous consequences. This thesis deals with designs of reliable systems in which multiple agents, operating in unreliable or hostile environments, collaborate to achieve a common goal. The design of such reliable systems is challenging because designers have no control over the environments in which agents operate. A system in which mobile agents communicate with each other over unreliable, wireless channels is an example of such a system. Agents communicate with others when they are in communication range of each other and they cease communication when they move out of range. The external environment determines whether and when agents can communicate. For example, enemies may jam communication among agents or the wireless environment may be noisy.

This thesis studies multi-agent systems in which designers cannot schedule interactions among agents. Designers cannot assume that an agent X will be able to interact with an agent Y some time in the future. Agents cannot control the environment which determines when agents can communicate and interact. This thesis explores undependable environments to understand the conditions under which multi-agent algorithms can operate correctly.

Many systems described in the literature assume that the environment is reliable. Reliable distributed systems are often modeled as static graphs in which each vertex represents an

agent and each edge represents an interaction mechanism between a pair of agents; for example, neighboring agents in the graph can operate on shared variables in atomic actions. This thesis departs from this type of earlier work in two ways. First, agent interactions cannot be scheduled. In terms of the traditional graph representation, we consider systems in which graph edges are changed dynamically by an external mechanism over which agents have no control. Second, in cases of applications based on messages, communication may be unreliable: messages may be lost or delivered out of order.

The environment is modeled as a nondeterministic mechanism. In carrying out worst-case analysis it is helpful to treat the environment as an adversary whose intent is to thwart the agents from reaching their goals. The adversary determines which agents can interact and which messages are lost. An all-powerful adversary is uninteresting because it could prevent any agent from interacting with any other and thus stop all computations. Therefore, we consider powerful adversaries, but not all-powerful ones.

We constrain the adversarial mechanism to obey certain weak "fairness" criteria. Our goal is to understand the maximum power that the adversarial environment can have—or equivalently, the weakest fairness criteria—that still ensures that the multiagent system operates correctly. For example, if a system is partitioned into two nonempty non-communicating subsets, then an agent in one subset cannot compute functions over the states of agents in the other subset. Therefore we explore algorithms in which a fairness criterion is that the system is never permanently partitioned into non-communicating subsets of agents. This thesis explores the boundaries between environments in which multi-agent computations can work and environments in which they cannot work.

An agent cannot control when it will interact with other agents, nor can it determine the identities of the agents with which it will interact; therefore, when a set of agents interacts, it must carry out its computation opportunistically without knowing about what may happen in the future. Each set of interacting agents carries out a computation independent of the number and identities of agents in the set. Thus, in our algorithms, every subset of agents carries out the same computation as the global set of all agents in the system. Therefore, we study algorithms in which a *local* computation is the same as the *global* computation. We

illustrate the ideas by using a simple example.

Example 1 (System semantics) Each agent j has a local constant $y[j]$ and a local variable $x[j]$ of some type $\mathcal{T}$. Let H be the multiset, or bag, of values $y[j]$. The goal is for each agent j to set its local variable $x[j]$ to a common value $f(H)$ where f is a given function that maps multisets of elements of type $\mathcal{T}$ to type $\mathcal{T}$. For example, the local constant $y[j]$ could be a real number and each agent j is required to set its local value $x[j]$ to the average of the y values. We next look at algorithms for systems with different connectivity between agents.

First consider systems that can be modeled by static graphs in which vertices represent agents and directed edges represent communication channels along which messages can be passed. To begin with consider an ideal graph in which there is a directed edge (j, k) from every agent j To every agent k. A simple algorithm is one in which each agent j broadcasts its local constant $y[j]$ to every other agent. Thus each agent receives the local constant $y[j]$ of every agent j, then determines the multiset H, and finally sets its local variable $x[j]$ to $f(H)$. An alternative algorithm is one in which agents first elect a single agent as a "leader." All agents send their local constants to the leader. The leader determines H, computes $f(H)$, and broadcasts $f(H)$ to all agents. All agents then set their local variables to the result.

Next consider systems in which the environment determines which agents can interact and when. The environment picks a subset K of agents and allows communication among agents in K; agents outside K cannot communicate. Agents have no control on how K is chosen. We can model this system by a dynamic graph in which edges are created and deleted by the environment. The environment connects all agents in a subset of agents K. That subset remains connected until either all the agents complete a step of a computation or perform the empty step in which they do not carry out an operation. The environment may then delete these edges and create edges connecting another subset of agents.

Designers do not know whether a subset of interacting agents is the entire global set or is a proper subset. Therefore we consider algorithms in which every subset of interacting agents carries out the same computation independent of the size and constituents of the subset. In

particular, if the system consists of a single agent j, then the algorithm is required to set the agent's local variable $x[j]$ to the given constant $y[j]$. Therefore, we consider algorithms of the following form: Initially each agent j sets $x[j]$ to $y[j]$. When a set K of agents interacts, each agent j in K sets its local variable $x[j]$ to $f(H')$ where H' is the multiset of the local variables $x[j]$ of agents j in K. In the case of computing the average, each agent j in a set of interacting agents sets its local value $x[j]$ to the average of the x-values of agents in the set. □

This simple example illustrates some of the issues explored in the thesis:

- We study fairness constraints on agent interactions that enable algorithms to reach the desired result. We show that certain computations operate correctly even in undependable environments.

- In some cases, the computation may converge to the desired result in the limit as time becomes arbitrarily large. The computation may never reach the desired result though it may get arbitrarily close. Much of the earlier work deals with formal methods to prove termination of computation. We use formal methods to prove convergence as well as termination.

- A computation can terminate only after all agents participate in the computation; if an agent never participates in a computation then that agent's initial values cannot influence the final result. If agents do not have information about the total number of agents in the system, or some other global information, then agents cannot detect termination of a computation because the computation cannot determine whether all agents have participated in it. If agents do have information about the global state of the system, such as the numbers and id's of agents, then they can employ more efficient algorithms, and we present such algorithms.

- Many papers deal with distributed algorithms in which the input to the algorithm is fixed. Earlier we discussed the example: compute the average over all agents j of $y[j]$ where $y[j]$ is a given input that remains constant. In this thesis we also consider

systems in which the input changes with time. Since inputs may change with time, the result computed by the multi-agent algorithm at each instant may be different from the result for the case where inputs are constant; we define the instantaneous error as the difference between the result computed by the algorithm and the true result. Our goal is to determine a bound on the error if such a bound exists. Consider the problem of computing the averages where the values $y[j]$ change with time, and the agents estimate the instantaneous average of $y[j]$ over all j; we wish to determine if the error between the estimated average and the true average is bounded.

- We use temporal logic and mechanical theorem proving systems to prove the correctness of multi-agent systems. In some cases we prove termination, and in other cases we prove convergence, and in yet other cases we prove bounds on the error.

Proving the correctness of systems of agents operating in hostile or unreliable environments is difficult. This thesis explores repeated use of a small number of formally proved algorithms to develop a large number of programs. We show how reasoning about systems using abstract algebraic concepts, such as monoids, helps in developing algorithms that are correct and implementations that are reliable. The benefit of program reuse has been demonstrated by object-oriented and compositional programming systems in which abstract components are used to create concrete programs. In these cases, development time is greatly improved by leveraging pre-built objects that are known to be correct. We consider the same idea applied to formal proofs: the cost of proving the correctness of an algorithm is amortized over multiple instances of programs that re-use that proof.

1.2 A Detailed Example

The following example presents the idea of repeated use of an algorithm and its proof. It is meant to be illustrative and is therefore described informally. A more formal treatment can be found in Chapter 3.

Consider a distributed system consisting of a fixed set of agents. Each agent contains a

unique identifier that is static, and a value from a given type that is mutable. The problem is to develop a distributed algorithm by which eventually all agents reach a consensus state, where consensus is a function of the initial states of the agents. Examples of such a function include

min The state of an agent state is a number.[1] The consensus state is the minimum value in the initial states of the agents.

gcd The state of an agent is a positive integer. The consensus state is the greatest common divisor of the initial states of agents.

convex hull The state of an agent is a set of points in a two-dimensional Cartesian plane. The consensus state is the convex hull of the initial sets of points of all agents.

Agents exchange values by sending messages to a communication medium, which, in turn, determines the set of agents that will receive that message. The communication medium is faulty in the sense that messages destined for a particular set of agents can be lost, duplicated, or delivered out-of-order. Moreover, if a message is not lost, its delivery can be delayed for an arbitrary, but finite time. That is, if a message m sent at time t is not lost, it will be delivered at some time $t + \Delta$, where Δ is an unknown non-negative bound. That Δ is bounded is more important than its value: this ensures that the amount of message overtaking is finite. Initially the communication medium contains no messages and messages are not corrupted by the communication medium.[2] Therefore, every message delivered to an agent was sent by some agent.

Some papers represent distributed systems as directed graphs where the vertices are agents and the edges represent message-passing channels. In our case, the graph is dynamic in that edges are added and deleted by a mechanism outside of the programs control. Note that consensus across all the agents in the system cannot be reached if there exists a subset of agents that are permanently partitioned from the global set. Therefore, we assume that

[1] More generally an element of a total order.

[2] We consider a relaxation to this assumption in Chapter 7.

for any non-empty proper subset of agents, messages from agents outside the subset are received by agents within the subset infinitely often.

Our goal is to write a program in an executable programming language, such as C or Java, for each agent. Consider the problem of computing the minimum of the initial values of agents. Let $S_t(k)$ be the state of agent k at a point t in the computation; further, assume that the state of an agent is an integer. The desired consensus value is x where:

$$x = \min_k S_0(k).$$

Agents repeatedly send a message containing their current state to subsets of agents within the system. When a message m is received in state S, the new state of the system S' follows

$$S' = \min(S, m).$$

1.2.1 Correctness

In this thesis, program correctness is first demonstrated for algorithms written in a logic notation and then in a conventional programming notation, such as Java.

1.2.1.1 Algorithm Correctness

One can reason, somewhat informally, about the correctness of the program in the following way. Let M be the set of messages in the communication medium at a point in the computation. An invariant of the system is:

$$(x = \min_k S(k)) \ \wedge \ (x \leq \min(\{M\})) \tag{1.1}$$

where $\min(M)$ is the minimum of all values in M. The argument for proving the invariant is that it holds initially, and that every action—sending or receiving a message, and updating state—maintains Equation 1.1.

Likewise, an informal proof of progress uses a variant function: the number of agents

whose state is different from x. Let D be the set of agents whose value is different from x, and let $\bar{D}$ be its complement. We need to show that if D is not empty then its cardinality will decrease eventually. Our fairness assumption implies that eventually an agent in D will receive a message from an agent in $\bar{D}$. Once this occurs, the receiving agent will change its state to x and become a member of $\bar{D}$, thus reducing the variant function.

1.2.1.2 Program Correctness

To establish that a program implements a given specification, we employ a theorem prover, a model checker, and code specification languages. Algorithms are first encoded in the notation of the theorem proving system—PVS [1] in our case. Unlike "conventional" programming languages, such as C or Java, This notation uses higher-order logic, making it easier to reason about algorithm correctness. Our use of PVS can be broken into two parts: a representation phase, and a proof phase. In the representation phase, our challenge is to represent the distributed system in the logic of PVS. The proof phase consists of proving safety and progress properties of the system. It is carried out in the steps we start with, proving distributed systems that use abstract data types such as monoids. Later, we prove that concrete implementations, such as min, satisfy the axioms of the abstract types. Such proofs consist of showing algebraic properties over system operators, such as

$$\forall a \in A\colon\ \min(a, a) = a$$

where A is a totally ordered set. We use a theorem prover proofs that we develop by hand.

We also evaluated the use of a model checker [2] within the development cycle. Using SPIN involves encoding the algorithm in Promela, which is closer to a conventional programming language than PVS. Model checkers are easier to use because they offer more automation than mechanically verifying proofs using a theorem prover.

After the algorithm is determined to be correct, we perform manual translations from PVS to conventional languages such as C#. Mechanical translations are the subject of a wide body of work [3, 4, 5, 6], but are not covered in this thesis. Our abstractions and stepwise

refinement from PVS to C# are simpler than developing and proving a program in C# from scratch. For example, in this final translation step the verification requirement is to prove that an action $z = \max(x, y)$ in Java implements the action $z = \max(x, y)$ in PVS. Therefore, ensuring that the translation is correct is easier than deriving a correct concurrent program. For added program verification, we employ the Java Modeling Language (JML) [7], along with our own invariant assertion classes. We also provide an implementation in C#, which is similar to Java, and in Erlang [8], which demonstrates the applicability of our abstractions to other programming notations.

1.3 Local-Global Algorithms

The thesis focuses on algorithms in which the actions taken by agents in a distributed system are identical to the actions that would be taken by a single process in a non-distributed system. For example, an algorithm to compute the minimum of a set of numbers in a single process scans the numbers in the set and updates the minimum value seen so far to the smaller of its current value and the number scanned. The distributed algorithm is identical: when a subset of agents interact they compute the minimum of values in the subset.

We call these algorithms *local-global* because the local actions taken by subsets of agents are identical to the global actions of a single, central, process. The association between such local actions and the global state of the system is known as a local-global relation. Many distributed algorithms are not local-global: the distributed implementation is different from an implementation on a single central process. Our goal in studying local-global algorithms and relations is to understand the simplest class of distributed algorithms that are obtained by replicating a single-process centralized algorithm for subsets of agents. We identify local-global relations to generate the aforementioned algorithmic abstractions.

1.3.1 The Local-Global Concept in Algorithm Development

To help describe the idea of local-global algorithms, we present a few examples.

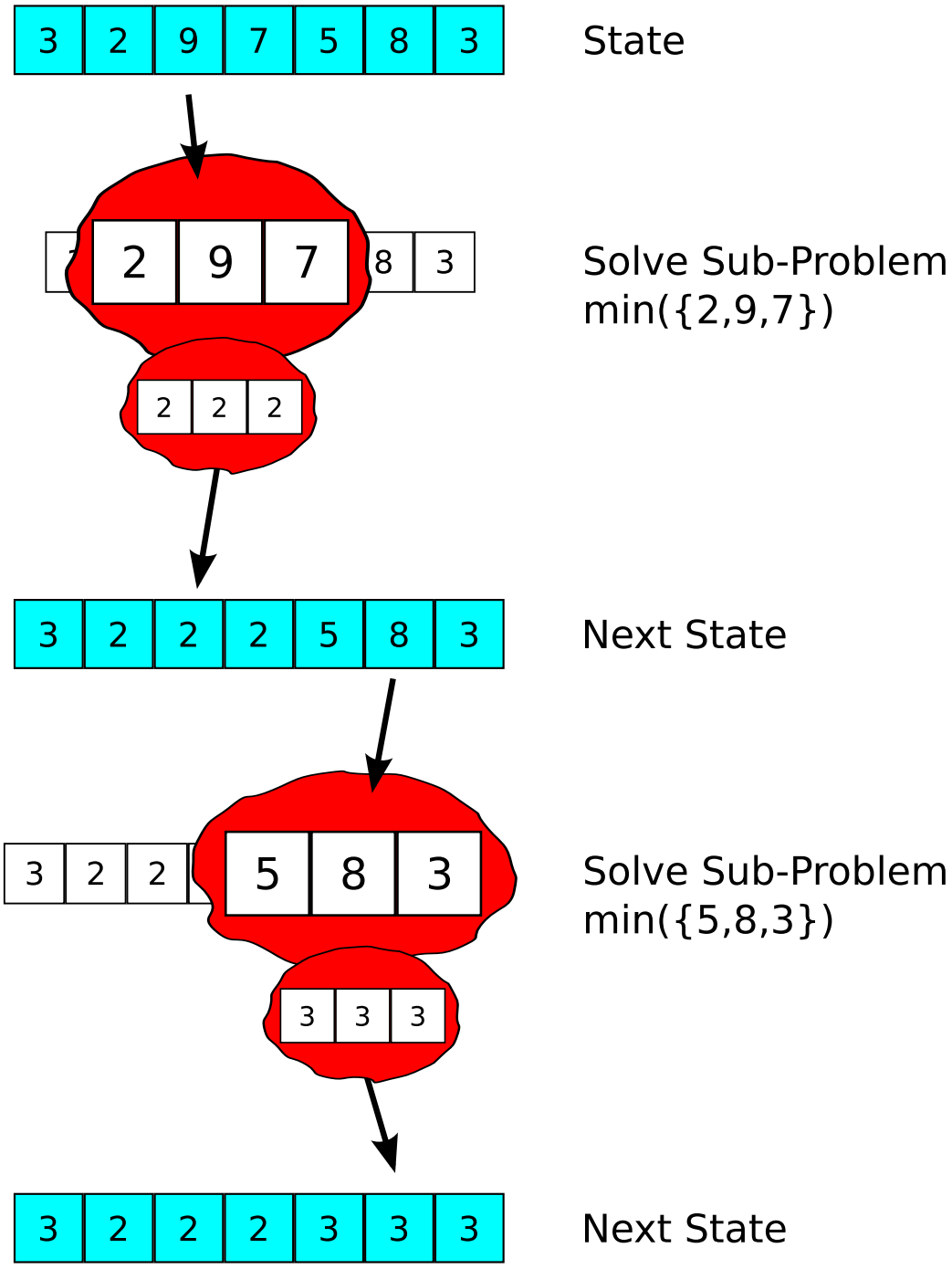


Figure 1.1: An example of distributed consensus, which fits a local-global relation. See Section 3.1 for details.

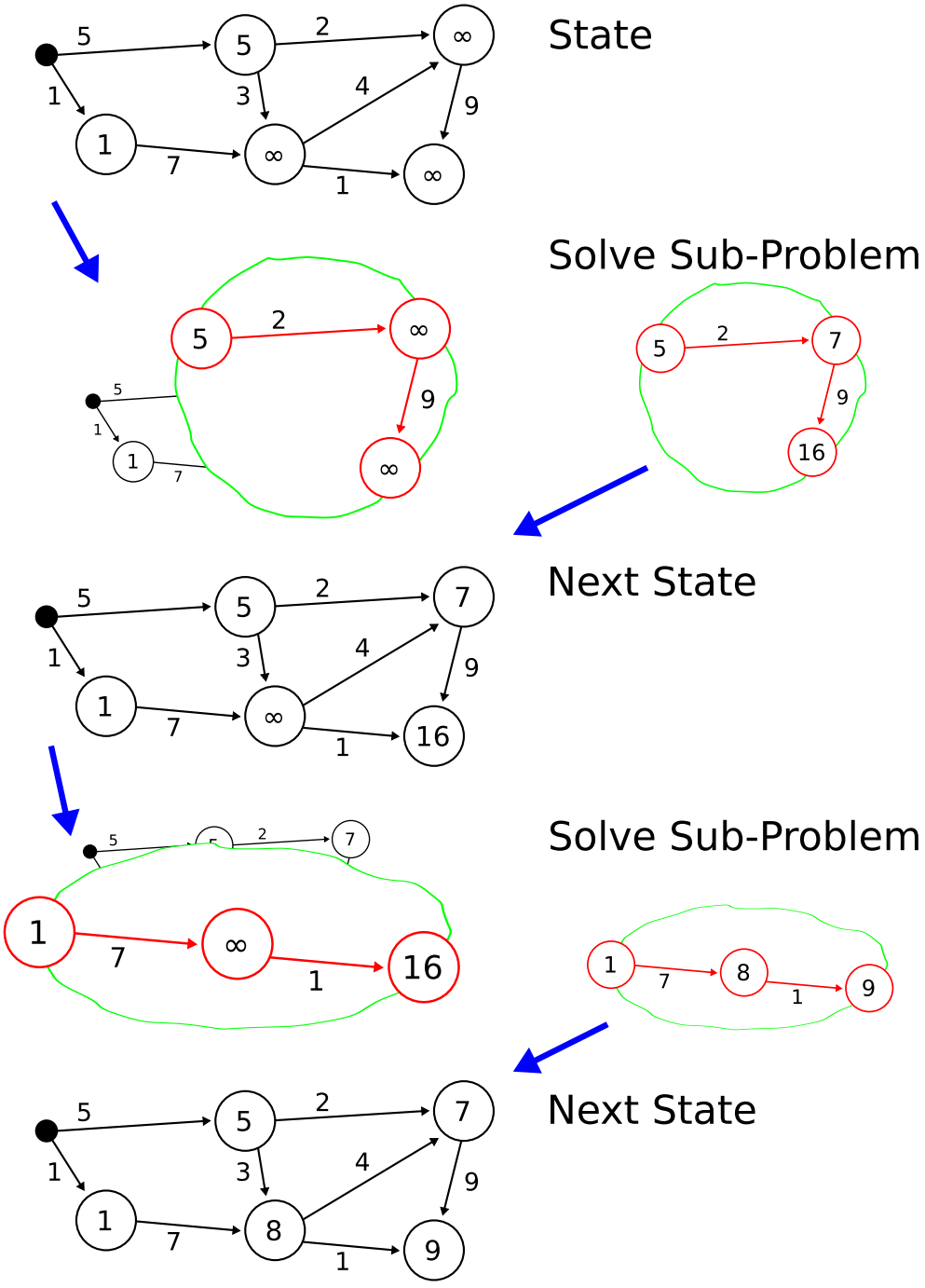


Figure 1.2: An example of distributed path analysis, which fits a local-global relation. See Chapter 4 for details.

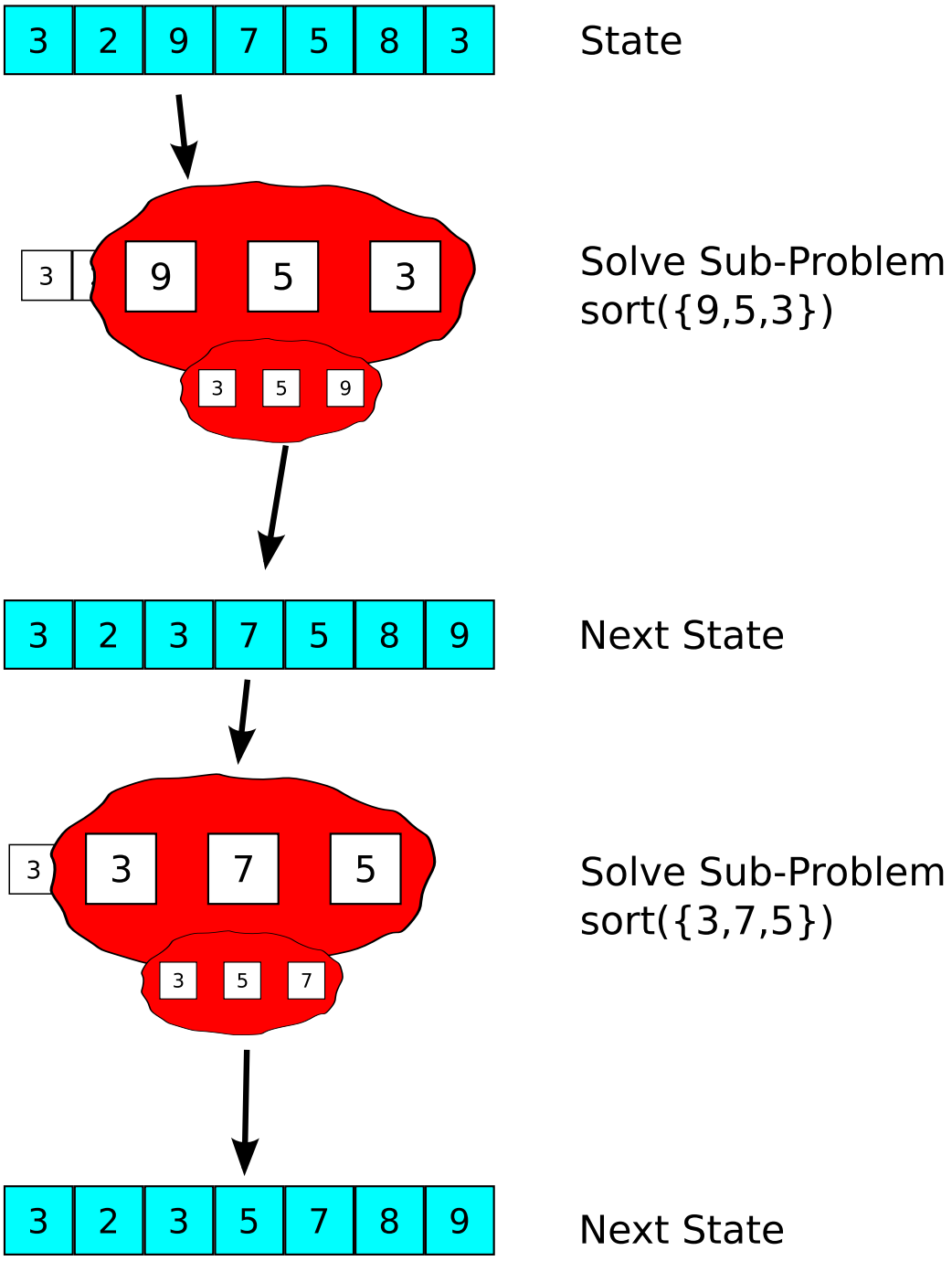


Figure 1.3: An example of distributed sorting, which fits a local-global relation. See Chapter 5 for details.

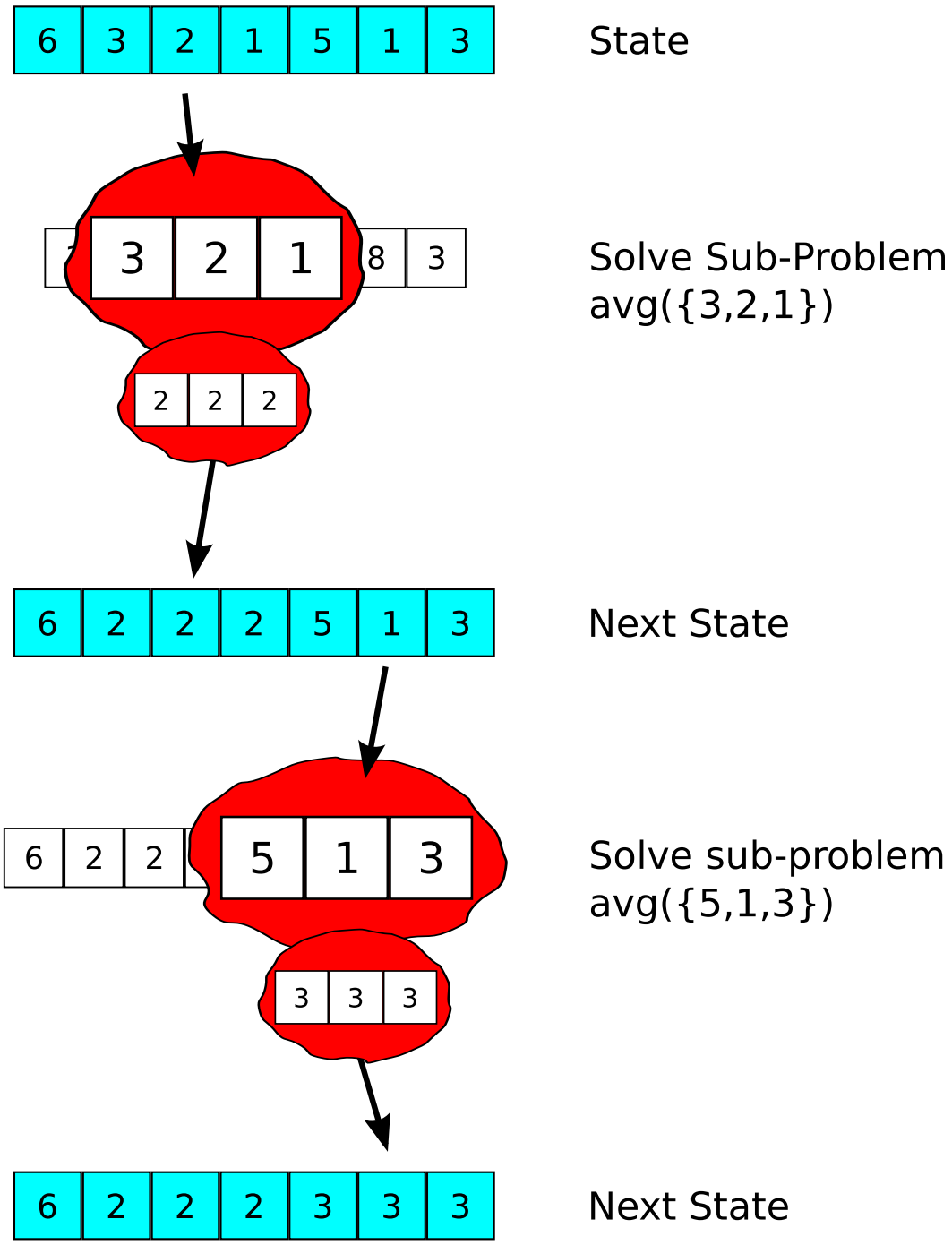


Figure 1.4: An example of distributed average consensus, which fits a local-global relation. See Chapter 6 for details.

Example 2 (Minimum array value) Consider a data structure that agents may only have partial access to. That is, they can read or modify subsections of the data, but not the entire structure. In the case of computing the minimum of an array, the problem is specified in terms of the array: an agent may only be able to access a subset of elements. If the agent could operate on the entire array, then the problem is quickly solved by setting all elements of the array to the minimum value. The action carried out by the agent on the subproblem is identical to the action that an agent would carry out if it could modify the entire array: set all elements of the subarray to the minimum value of the subarray.

This is shown in Figure 1.1 where the initial state is

$$[3, 2, 9, 7, 5, 8, 3]$$

indexed j where $0 \leq j < 0$. The first action is executed by an agent that can only read and modify elements 2 and 3 of the array. Therefore, it operates on the subarray to get a new state for the subarray as shown in the figure. The values of other parts of the data structure remain unchanged, giving the next global state shown in the diagram. □

Example 3 (Shortest paths) Consider the problem of computing the lengths of the shortest paths from a vertex root to all vertices in a directed graph. If an agent can read the entire graph it solves the entire problem; that is, it computes the shortest distances to all vertices. The algorithm by which the problem is solved is irrelevant to the local-global concept.

Now consider the case where an agent can see only a part of the graph. Assume that associated with each vertex is the length of a path to that vertex; this length may not necessarily be the shortest one. An agent that can see only a part of the graph solves the shortest path problem for the subgraph that it can see. This is shown in Figure 1.2. □

Example 4 (Sorting) Figure 1.3 shows the local-global idea applied to sorting. An agent sorts the part of the array that it can see. □

Example 5 (Average consensus) Figure 1.4 shows the idea applied to computing the average of a collection of values. Each element of an array is to be set to the average value

of the array. If an agent can only modify a part of the array, then it sets the values that it can modify to the average of their values. □

Some problems cannot be solved using the local-global approach. For example, consider the problem in which all elements of an array are to be set to its *second-smallest* value. Applying the local-global idea we would design an algorithm in which an agent that could operate on only a part of the array sets all the elements that it could modify to the second-smallest of their values. But this algorithm is incorrect. A local-global algorithm to compute the two smallest, or in general the k smallest, values works correctly. This thesis does not provide necessary and sufficient conditions on problem specification for the problem to be amenable to the local-global approach.

1.3.2 Local-Global Consistency in Proofs

We usually demonstrate the correctness of a centralized single-agent algorithm by providing a safety property—typically an invariant—and a progress property. We need to show that the proof for the single-agent algorithm, in which the agent has access to the entire data structure, is not violated by an algorithm in which an agent operates on only the part of the data structure that it sees.

Safety We need to prove that if a local operation on part of the data structure is safe, that it satisfies an invariant, then the local operation also keeps the global data structure safe as well.

Progress Likewise, we need to show that if a local operation on part of the data structure moves that part closer to the goal—the desired end state for that part—then that local operation also moves the global data structure closer to the global goal. Further, we need to show that a local operation that moves a local part of the data structure will be executed eventually; we do so using specialized fairness arguments.

We present a generic approach for proving that if local operations satisfy safety and progress properties for the parts of data structures then these operations also satisfy safety and

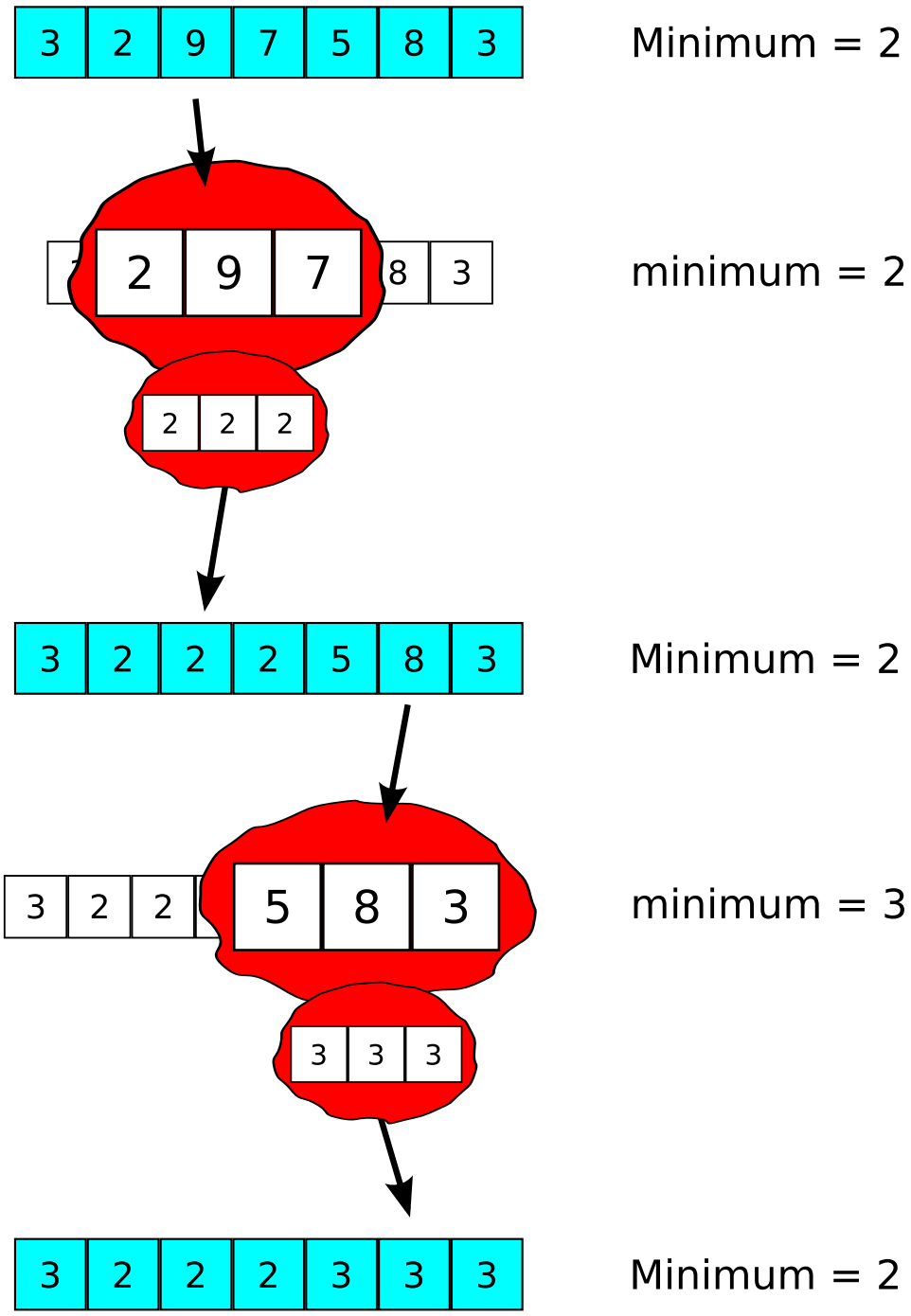


Figure 1.5: Consider again the example presented in Figure 1.1. The local-global relation implies that the global minimum is conserved.

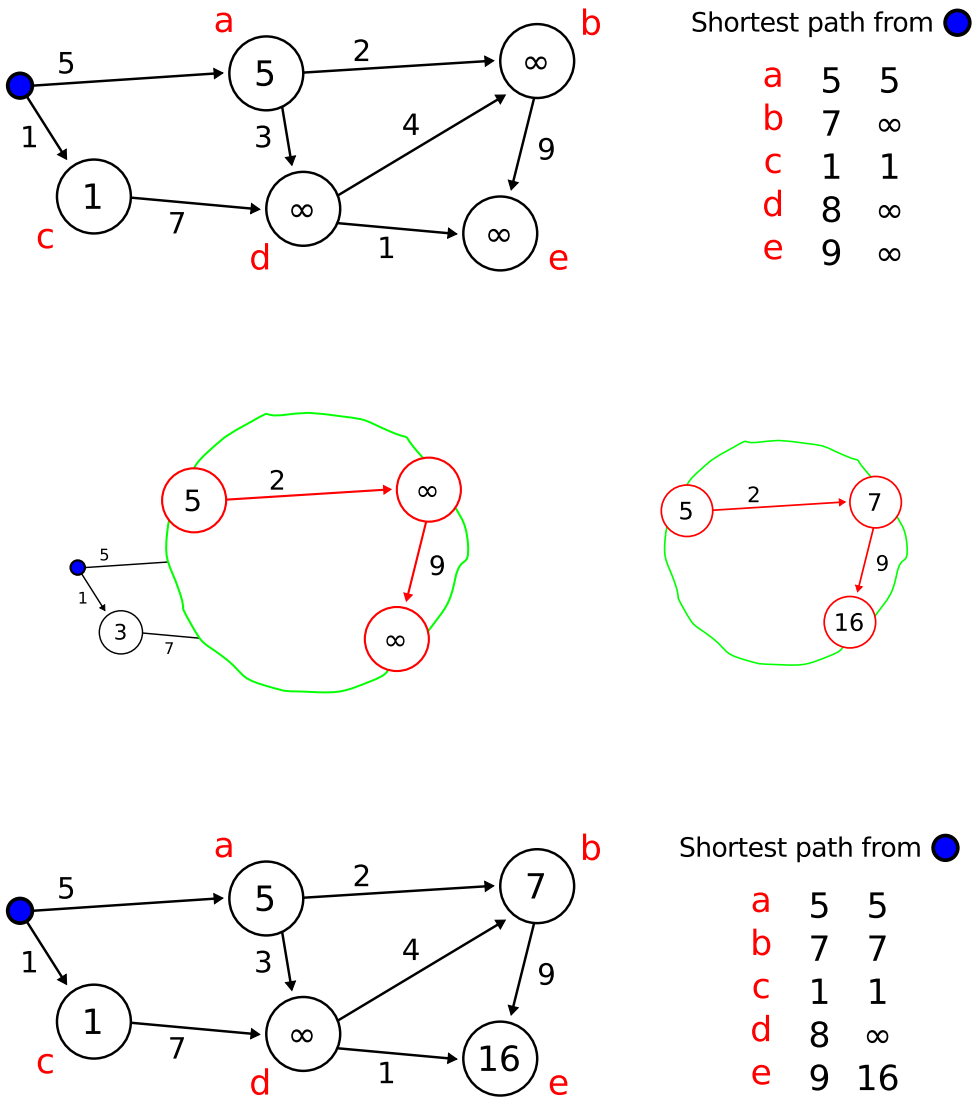


Figure 1.6: Consider again the example presented in Figure 1.2. The local-global relation implies that edge traversal improves global path estimation.

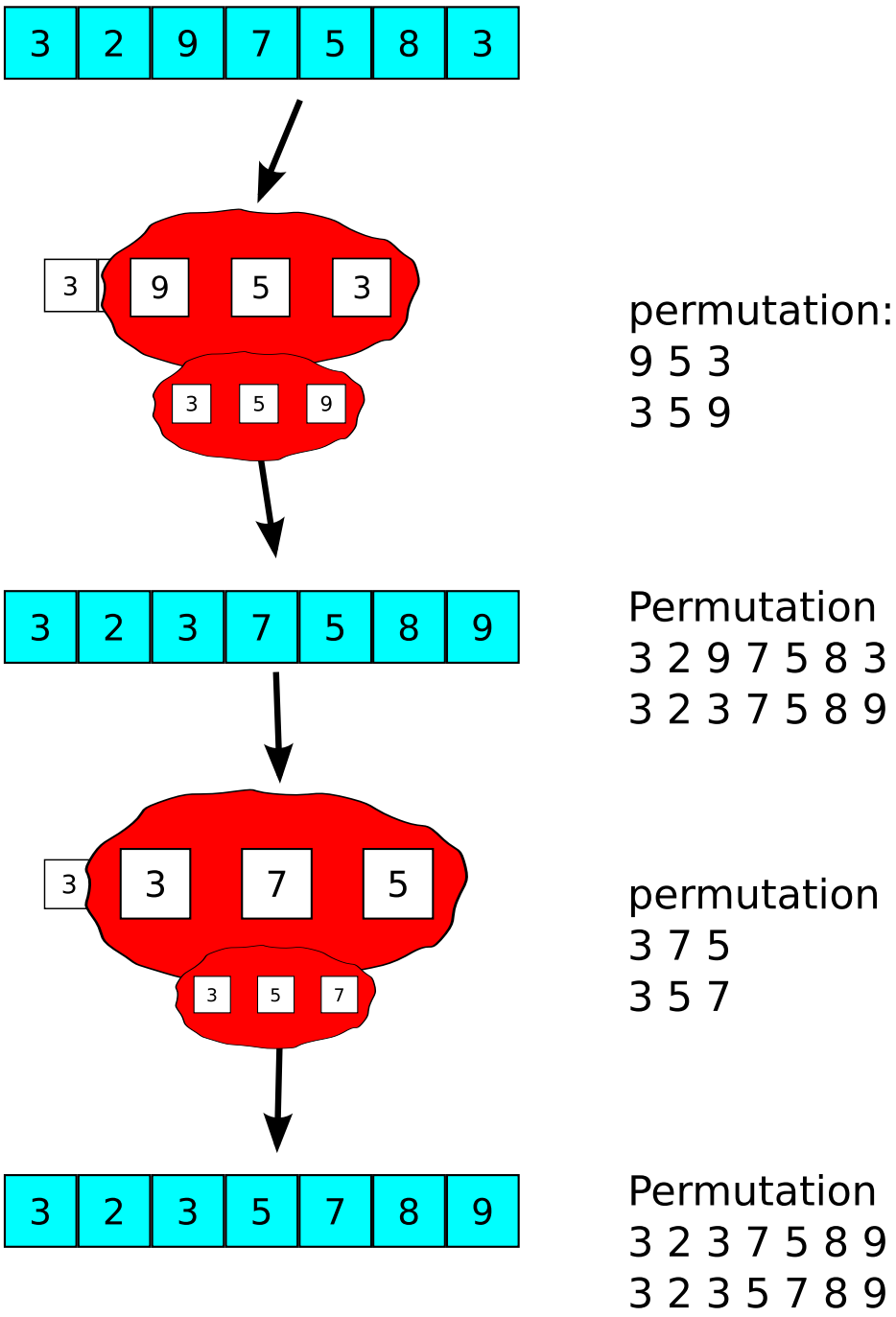


Figure 1.7: Consider again the example presented in Figure 1.3. The local-global relation implies that a permutation of the array is maintained.

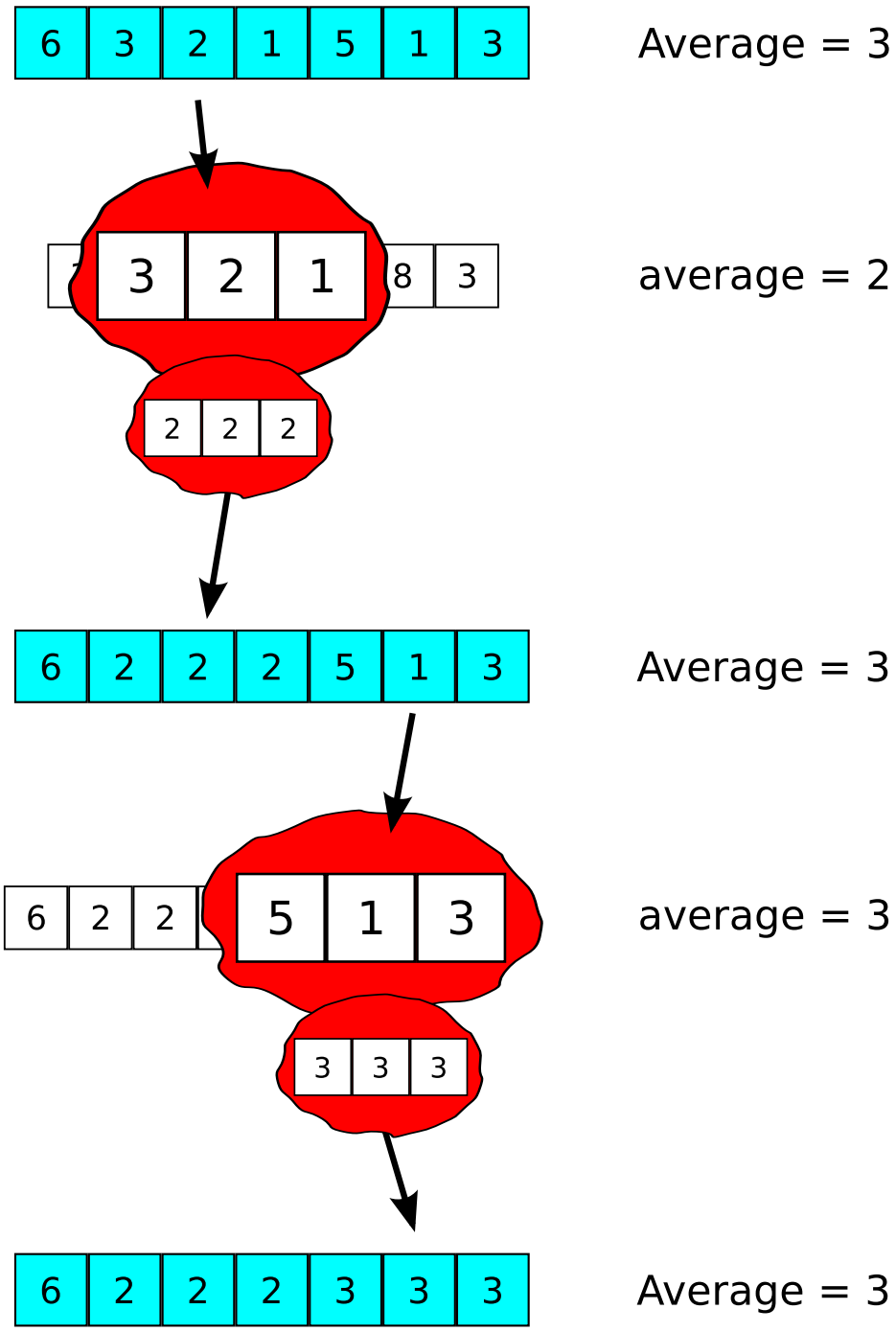


Figure 1.8: Consider again the example presented in Figure 1.4. The local-global relation implies that the average is maintained.

progress properties of the entire data structure. Then we use this generic approach repeatedly for all the problems considered in the thesis.

1.4 Contribution and Scope

1.4.1 Reducing the Cost of Verification

Although formal methods are recognized as being beneficial in preventing software errors, that recognition comes largely from within the research community. Part of this disparity stems from the fact that formal methods can be difficult for non-experts to use. Formal proofs that are mechanically verified provide high confidence in a programs correctness, but require an understanding of predicate calculus and an expertise in a problems domain. Model checking works by systematically checking program paths and analyzing program state in search of various error conditions. This approach can provide detailed insight into why a particular error might occur, but might not find such errors when a programs state space is large [9]. Using specification languages can be difficult for large programs. Targeting where specifications should be placed and the proper level of discourse they should display can be difficult for developers [10].

This thesis attempts to mitigate the challenges that these tools present by developing algorithm abstractions that are reusable, which can be done using local-global relations. Specifically, our specification of a distributed system is independent of the algorithm run on that system. Further, the algorithm specification is independent of the operation it is performing. For example, consensus to a minimum value can be seen as consensus using a monoid where the operator within the monoid is min. With respect to theorem proving, we develop a library of theorems based on distributed algorithms using this abstraction methodology. This can be thought of as the mathematical library that most theorem provers provide, but with an emphasis on distributed algorithms. We are also able to focus the efforts of model checking and code specification.

1.4.2 Limits to Local-Global Computations

Systems in which agents can orchestrate interactions perfectly will have better performance than systems in which agent interactions are determined by an external mechanism. Likewise, systems in which agents can send messages to any other agent without message loss will have better performance than systems in which messages get lost. An issue we explore is how much performance is lost by multi-agent systems operating in uncertain and hostile environments compared to performance in ideal environments. This analysis evaluates the total time, the total number of messages, and the total volume of information exchanged, in the ideal environment and a hostile environment. If we only make fairness assumptions about agent interactions, then the ratio between the best and worst cases can be unbounded. Therefore, we also carry out analysis with tighter constraints on mechanisms for agent interaction.

We study algorithms for termination detection for systems in which agent interactions are determined by external agencies. Many of the termination detection algorithms in the literature assume that agent interactions are specified by static graphs in which vertices represent agents and edges represent shared variables or message-passing channels. The termination-detection algorithms studied here have to operate in an environment in which algorithm designers do not know which interactions can occur and when.

1.5 Related Work

Frameworks for proof classification and reuse have been considered before. Lynch, for example, has built hierarchical correctness proofs using I/O automata [11], and has applied her idea to various distributed algorithms [12]; Jonsson champions a similar technique for I/O systems [13]. Both define the refinement of one automata to another based, amongst other things, on execution output traces. Unlike our work, many of the algorithms are concerned with communication protocols. Because they start with specifications of a given system, even their highest level of abstraction is more focused than what we consider. Moreover, their motivation for refinement is primarily to ease the requirements of correctness proofs.

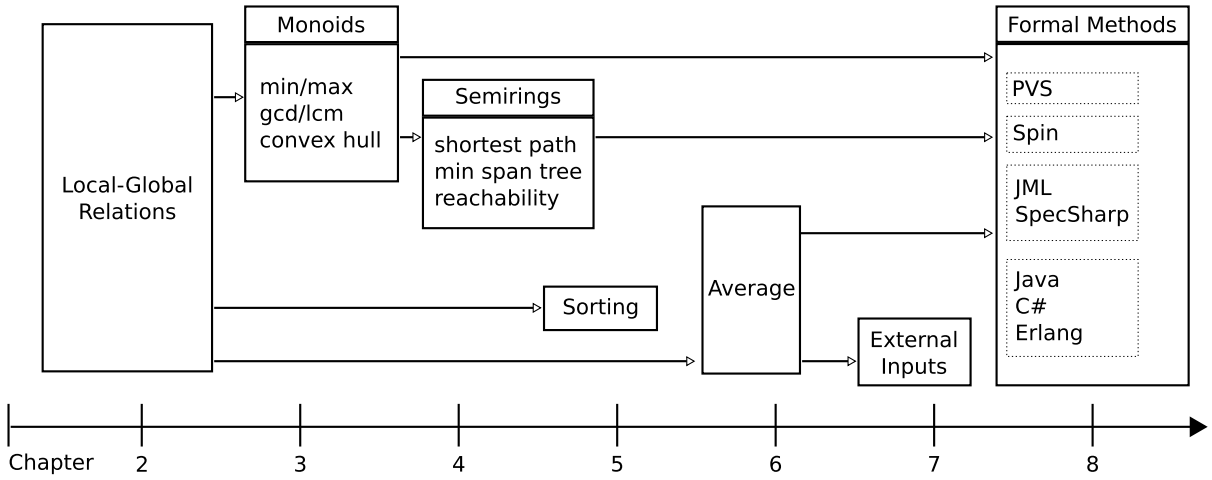


Figure 1.9: Thesis outline. Each box represents a chapter—its main points listed therein.

While we share in this goal, we are also trying to build a library of reusable proofs.

Simplifying formal methods for program development has been studied by Möller [14]. He approaches the problem from a very theoretical level, first defining, then applying, an algebra of formal languages that is reusable across problem domains. As is the case in our formalization, he identifies key properties necessary for concrete operators to posses—associativity and commutativity, for example. He applies his formalization to sorting and graph problems, which our formalization maps to as well (Sections 4 and 5, respectively).

1.6 Organization of the Thesis

Figure 1.9 presents a visual outline of this thesis: Chapter 2 details our system model, including system fairness and correctness. It also introduces *local-global relations*, a recurring concept throughout the thesis. Chapter 3 applies these relations to monoids within consensus problems; Chapter 4 continues this abstraction by applying semirings to graph problems. Chapter 5 looks at local-global relations with respect to sorting; Chapter 6 within averaging. Chapter 7 extends Chapter 6 to study systems with external inputs. Termination detection, error bounds for systems with changing inputs, and limits to the local-global approach are

discusses in Chapter 8. We conclude with Chapter 10.

Chapter 2

Model and Assumptions

The first section of this chapter reviews widely-used models for distributed systems. The material is presented here for completeness. Section 2.2 introduces the basic idea of local-global relations and algorithms that were introduced in Chapter 1.

2.1 System Model

Next, we present a brief review of labeled transition systems.

Definition 1 (Labeled Transition System) A labeled transition system is an ordered quadruple, $(\mathcal{S}, \Lambda, \mathcal{L}, \twoheadrightarrow)$, where: $\mathcal{S}$ is a set of states, $\Lambda \subseteq \mathcal{S}$ a set of start states, $\mathcal{L}$ a set of labels, and $\twoheadrightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ a ternary relation defining transitions between states. □

The set of labels are associated with "actions" of agents in a distributed system; for example, an action may be to send a message containing the current state of the agent. A state transition is represented by

$$S \overset{l}{\twoheadrightarrow} S'$$

where S is the state before the transition, S' is the state after the transition, and l is the label or the action that caused the transition. The execution of an action l when the system is in state S may result in one of many possible next states. For example, let S, S' and S'' be states of the system; then both $S \overset{l}{\twoheadrightarrow} S'$ and $S \overset{l}{\twoheadrightarrow} S''$ may be valid state transitions. An example of a nondeterministic action that will be used later in the thesis occurs in distributed

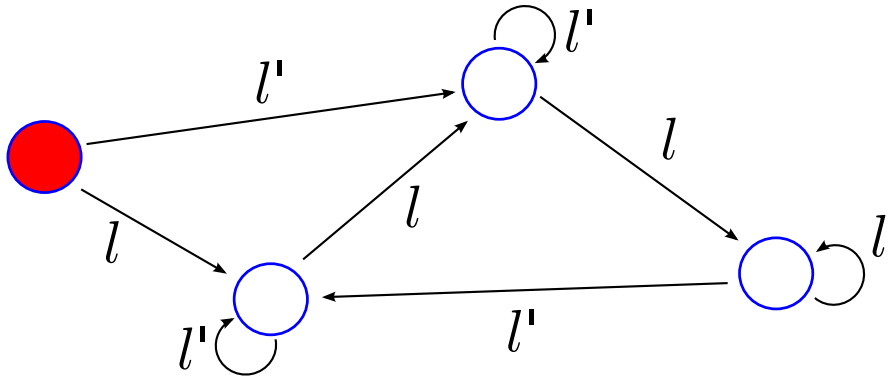


Figure 2.1: A graphical representation of a labeled transitions system. Darkened agents denote possible start states.

computation of averages: the action modifies the values of real variables in a set of agents so that the average of the values remains unchanged and the sum of squares is reduced by *at least* p percent. This action does not specify the precise amount by which the sum of squares should be reduced; this action can take the system from a given state to different next states depending on how much the sum of squares is reduced.

A labeled transition system with a finite number of states can be represented by a labeled directed graph in which the vertices represent states and the labeled directed edges represent transitions between states. There may be many outgoing edges with the same label from the same vertex. In some models, actions are deterministic—for each label and each vertex there is at most one outgoing edge with that label. In some models, such as UNITY [15], actions are deterministic and any action can be executed in any state (though the action may be a *skip* which does not change the state); in the corresponding graph model, for each label and each vertex, there is exactly one outgoing edge with that label.

An *execution* of a transition system is a sequence of state transitions, starting from an initial state, where the end-state of each transition is the start-state of the next transition.

$$\left\{ (S, l, S')_i \mid S \overset{l}{\rightarrow} S' \right\}$$

where $i \in \mathbb{N}_0$, $l \in \mathcal{L}$, and $S, S' \in \mathcal{S}$. In terms of the graph, an execution is a path in the graph starting at an initial state represented by a root vertex (Figure 2.1). It is sometimes convenient to denote the execution in a sequential, rather than set-builder, notation:

$$S_0 \overset{l_0}{\rightarrowtriangle} S_1 \overset{l_1}{\rightarrowtriangle} \ldots \overset{l_i}{\rightarrowtriangle} S_i \overset{l_{i+1}}{\rightarrowtriangle} \ldots,$$

where $S_0 \in \Lambda$ and $\forall i \in \mathbb{N}_0 \colon (S_i, l_i, S_{i+1}) \in \rightarrowtriangle$. We make the assumption made in UNITY that for each label l and each state S there is at least one transition from S, which may be a "skip" from S back to itself. This assumption simplifies the model when we discuss fairness. Because of this assumption, for each state S there exists an infinite execution from S, though that execution may remain in the same state forever.

2.1.1 Distributed Systems

A distributed system is a fixed finite set $\mathcal{A}$ of agents and a labeled transition system that has the following properties. Let N be the number of agents. Associated with each agent v is a set $\mathcal{T}_v$ of agent states and a subset of these states called the initial states of v. In most of the applications studied in this thesis all agents have the same sets of agent states; so, we drop the subscript v and refer to the set of agent states as $\mathcal{T}$. The properties we require of a distributed system are:

1. A state in the transition system is an N-tuple of agent states, where $N \in \mathbb{N}_0$. We refer to a state of the transition system as a *global* state or *system* state, and to the state of an agent as a *local* state or an *agent* state.

2. Each transition leaves the states of some subset of agents unchanged and may change the states of agents that are not in the set. We use the label $l \in \mathcal{L}$ for any transition that may change the states of a set l of agents while leaving the states of agents not in l unchanged.

3. The initial global state is a tuple of initial agent states.

Channels and message communication media are modeled as agents. For example, the message communication medium described in the previous chapter is modeled as a set of messages in transit. The state transition corresponding to delivering a message from this set to an agent v may change the state of the communication medium and agent v but leave the states of all other agents unchanged.

We use $S(k)$ to denote the state of an agent k when the global state is S. A global state is represented either as an N-tuple or as a set $\{(k, S(k)) \mid k \in \mathcal{A} \wedge S \in \mathcal{S}\}$ of pairs. We denote the restriction of S to a subset of agents $K \subseteq \mathcal{A}$ by $S|_K$.

$$S|_K \;=\; \{(k, S(k)) \mid k \in K\}\,.$$

2.1.2 Fair Executions

An execution in which agents in one subset K never interact with agents in the complementary set cannot reach a global consensus or reach other global goals because agents in K never have information about agents that are not in K. Therefore we restrict infinite executions to have certain fairness properties.

Many models have fairness criteria that all actions are executed infinitely often in an infinite execution. Thus, in these models, for each point t in an infinite fair computation, for each action l, there is a later point t' at which action l is executed. Most of the programs discussed in this thesis rely on a weaker model: We only require that the set of agents are not permanently partitioned into subsets K and $\bar{K}$ where no action is executed that includes agents in both K and $\bar{K}$ (see Figure 2.2). The fairness criterion in this case is that for every non-empty proper subset K of agents: actions that are executed jointly by agents in both K and $\bar{K}$ are executed infinitely often in an infinite fair execution.

Consider a system with agents indexed $k = 0, 1, 2, 3$. For any non-empty proper subset, such as $K = \{0, 1\}$, actions that span both K and its complement are, for example, actions that include agents in the sets $\{0, 2\}$, or $\{0, 3\}$, or $\{1, 2\}$, or $\{1, 3\}$, or $\{0, 2, 3\}$, and so forth, through the full set $\{0, 1, 2, 3\}$. The fairness criteria is that at each point t in the

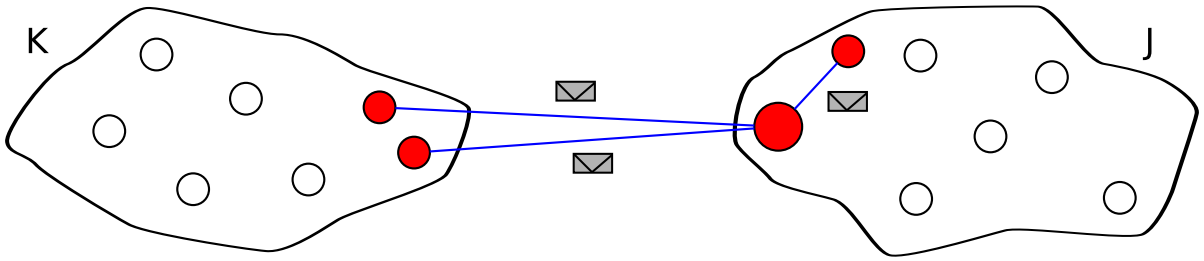


Figure 2.2: In a *fair* execution, agents will never be permanently partitioned. During an execution, agents communicate within groups; K and J above, for example. However, infinitely often, eventually agents will communicate across these partitions, such as the darkened agents above.

computation there is a later point t' at which one of these actions is executed in an infinite fair computation. Let F_K be the set of actions that include an agent from K and an agent from $\bar{K}$; then the fairness requirement is that each point t in an infinite fair computation there is a later point t' at which one of the actions in F_K is executed.

A different set of agents, say $J = \{0, 2\}$, also has a fairness requirement to ensure that agents in J can interact with agents outside J. This reasoning gives us the following fairness requirement:

Definition 2 (Fair Execution) A fairness condition $\mathcal{F}$ for a set of transitions is a finite collection $\{F_i\}_{i=1}^{n}$, where each F_i is a non-empty subset of $\mathcal{L}$. An infinite sequence of actions $l_1, l_2, \ldots$ is fair if and only if

$$\forall F \in \mathcal{F},\ \forall n \in \mathbb{N}_0,\ \exists m \in \mathbb{N}_0 :\quad m > n\ \wedge\ l_m \in F.$$

□

Definition 2 requires that infinitely often a spanning tree of the communication graph is formed, regardless of which spanning tree that is. This specification is a weaker model of fairness than traditional weak fairness assumptions [16]. By reducing all $F_K \in \mathcal{F}$ to particular singleton subsets, a model of traditional weak fairness can be formed.

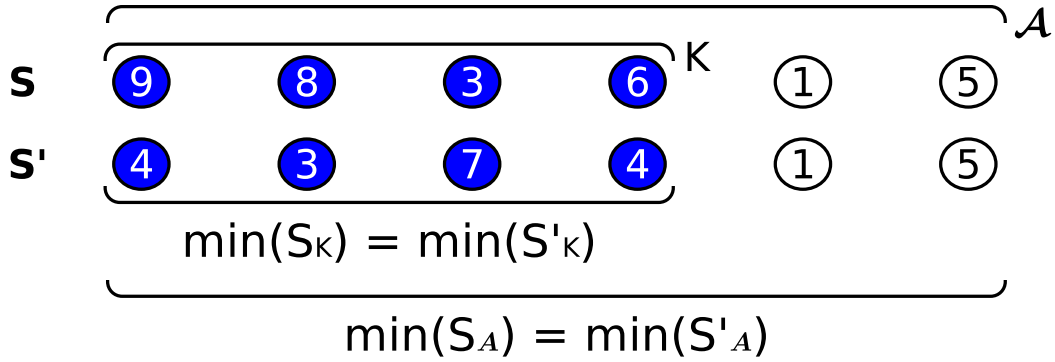


Figure 2.3: A local-global relation. Consider the transition $S \xrightarrow{K} S'$, denoted above. The "local" minimum value—the minimum value of the subset of agents K—in S is equal to the local minimum value in S'. Likewise, when the minimum value over the global state of the system is calculated, that value is also equal in both S and S' as well.

2.2 Local-Global Relations

As we discussed in the previous chapter, we study a class of algorithms in which the computational steps taken by any subset of agents that participate in an action are the same as the steps taken if *all the agents in the system* participate in an action. In other words, the steps taken by any subset of agents in an action are the same as those taken by a single, central process with access to all the data. We used the simple example of computing the minimum of a set of values: if all the agents in a system can participate in an atomic interaction then each agent sets its value to the minimum of the values of all the agents. Likewise, when all the agents in any subsystem participate in an interaction, each agent sets its value to the minimum of the values of all agents in the subset.

Many problems cannot be solved by algorithms in this manner; that is, in which local actions of agents in a distributed system are identical to the actions of a single central process. Some interesting problems, however, can be solved in this way. Here we explore some properties of problems that allow for the development of such algorithms.

We motivate the definition by continuing use of the simple example of computing the

minimum. Let the global state of the system, with agents indexed 0, 1, 2, be

$$\{(0,5),(1,7),(2,4)\},$$

where a pair $(k, S(k))$ identifies an agent k and its state $S(k)$. When any set K of agents participates in an interaction, each agent in K sets its value to the minimum of the values of all agents in K. For example, if agents 0 and 1 participate in an interaction then the global state after the interaction is

$$\{(0,5),(1,5),(2,4)\}.$$

The minimum of the values of the agents in K—the value 5—is not changed by the action. Moreover, the minimum of the values of all the agents in the system—the value 4—is not changed by the action either. This property is an example of a conservation law: if a property is conserved locally then it is also conserved globally. We capture this notion by the following definition of a *local-global* relation between states. The form of the conservation law for the minimum example is:

$$\begin{aligned}&(\min(\{S(k) \mid k \in K\}) = \min(\{S'(k) \mid k \in K\})) \bigwedge (\forall j \notin K\colon S(j) = S'(j))\\ &\implies\\ &\min(\{S(k) \mid k \in \mathcal{A}\}) = \min(\{S'(k) \mid k \in \mathcal{A}\}).\end{aligned}$$

We generalize this idea to local-global relations between states of sets of agents.

Definition 3 (Local-Global Relation) Let $\trianglerighteq$ be a transitive binary relation between system-states. A local-global relation follows

$$\forall K \subseteq \mathcal{A}, j \notin K\colon \left(S|_K \trianglerighteq S'|_K \bigwedge S(j) = S'(j)\right) \implies S|_{\mathcal{A}} \trianglerighteq S'|_{\mathcal{A}}.$$ □

An example of a relationship that is not local-global is for a collection of agents to compute their *second* minimum value, denoted $\min_2$. That is, rather than taking the smallest value of a set after an interaction, agents take the second smallest value. Again, let the global

state of the system be

$$\{(0,5),(1,7),(2,4)\}.$$

The second smallest value in the system is 4, which the agents should agree upon at some point during the execution. However, consider the case where agents 1 and 2 interact; the global state of the system is updated as follows

$$\{(0,5),(1,7),(2,7)\}.$$

In this post-interaction state, the correct consensus value is lost and the local-global relation is violated:

$$\min_2(\{7,4\}) = \min_2(\{7,7\}) \nRightarrow \min_2(\{5,7,4\}) = \min_2(\{5,7,7\}).$$

The equality relation holds in the antecedent, $7 = 7$, but not in the consequent, where $5 \neq 7$.

Theorem 1 (Reduced Local-Global Relation) *Let $\trianglerighteq$ be a transitive binary relation between system-states. If*

$$\forall K \subseteq \mathcal{A}, j \notin K\colon \Big(S|_K \trianglerighteq S'|_K \bigwedge S(j) = S'(j)\Big) \implies S|_{K\cup\{j\}} \trianglerighteq S'|_{K\cup\{j\}}$$

holds where K is not empty, then $\trianglerighteq$ is a local-global relation. □

Proof The proof is by reverse induction on K: We first prove the theorem for the full set of agents. We then assume its correctness for a general set of agents, and use this assumption to show correctness for a smaller set. This scheme is formally outlined in Section A.1.

Base Case Let K be the full set of agents $\mathcal{A}$:

$$\forall S, S' \in \mathcal{A}\colon S|_{\mathcal{A}} \trianglerighteq S'|_{\mathcal{A}} \wedge \forall j \notin \mathcal{A}\colon S(j) = S'(j) \implies S|_{\mathcal{A}} \trianglerighteq S'|_{\mathcal{A}}.$$

This holds trivially since there are no agents not in $\mathcal{A}$.

Inductive Step For all $S, S' \in \mathcal{A}$,

$$S|_{K\cup\{k\}} \trianglerighteq S'|_{K\cup\{k\}} \wedge \forall j \notin (K \cup \{k\}) : S(j) = S'(j) \implies S|_{\mathcal{A}} \trianglerighteq S'|_{\mathcal{A}} \quad \bigwedge \tag{2.1}$$

$$S|_K \trianglerighteq S'|_K \wedge \forall j \notin K \colon S(j) = S'(j) \tag{2.2}$$

$$\implies$$

$$S|_{\mathcal{A}} \trianglerighteq S'|_{\mathcal{A}}, \tag{2.3}$$

where $k \notin K$. Equation 2.3 follows directly from the consequent in Equation 2.1; however, to use that consequent we must discharge its antecedent—a two-step process because of the conjunction. Assuming Equation 2.2,

1. $S|_K \trianglerighteq S'|_K \implies S|_{K\cup\{k\}} \trianglerighteq S'|_{K\cup\{k\}}$, which follows by assumption.
2. $\forall j \notin K \colon S(j) = S'(j) \implies \forall j \notin (K \cup \{k\}) : S(j) = S'(j)$, which holds since $(\mathcal{A} \setminus (K \cup \{k\})) \subsetneq (\mathcal{A} \setminus K)$. ∎

The local-global theory is very general: it talks about neither the nature of the communication nor the computation, that takes place between agents. Thus, assuming that the computation is done locally, the local group size determines the amount of time required to solve a given problem. In the worst case, a set of agents performs only pairwise calculations; while the best case is one in which all agents form a single group after system initialization.

Another method for use the local-global framework is for agents to only exchange information about their initial state, and perform the computation "offline" once they have a complete understanding of the global state. In theory, such a methodology is acceptable as long as the implicit assumptions about our framework are obeyed:

1. commutativity of the operation is preserved. If agents are to perform the computation over the global state of the system, the order in which they process individual agent values should not matter;

2. agents must exchange the aggregate state of all agents they have encountered. If they only exchange their local-state, a stronger fairness assumption than they one relied on

in this thesis is required;

3. and, a protocol for determining when all states of all agents have been exchanged must be put in place. Without such knowledge, agents will never perform the underlying computation. Although the local-global framework as we have presented it does not mention termination, as we will see in Section 2.3.2, the system still makes incremental progress during execution.

In practice, performing delayed global computation is not always practical; for instance, when a large number of agents are in the system. In this case, an implementation may fail or perform poorly due to memory and computational constraints. By performing solving the problem locally, our framework deals with such issues in-place.

2.3 Correctness

2.3.1 Invariants

An action l is said to satisfy a local-global relation $\trianglerighteq$ if and only if any transition from state S to any state S' due to execution of action l satisfies $S|_K \trianglerighteq S'|_K$. Formally,

$$\forall l \in \mathcal{L}, K \subseteq \mathcal{A}: \quad K = l \ \wedge \ S \xrightarrow{l} S' \implies S|_K \trianglerighteq S'|_K. \tag{2.4}$$

Theorem 2 (Maintaining Local-Global Relations) *If all actions of a transition system satisfy a local-global relation, then the system has the following invariant:*

$$\textbf{\textit{Invariant}}: \qquad S_0|_{\mathcal{A}} \trianglerighteq S_t|_{\mathcal{A}}.$$

□

Proof of Theorem 2 follows from transitivity on $\trianglerighteq$.

Corollary 1 (Conservation) *Let f be a function from states of sets of agents to some*

type. Consider $\trianglerighteq$ *defined as*

$$\forall K \subseteq \mathcal{A}: \quad S|_K \trianglerighteq S'|_K \;\equiv\; f(S|_K) = f(S'|_K).$$

If all actions of a transition system satisfy $\trianglerighteq$*, then an invariant of the system is*

$$\textbf{\textit{Invariant}}: \qquad f(S_0) = f(S). \qquad \square$$

Corollary 2 (Non-increasing) *Let* f *be a monotone function from states of sets of agents to some totally ordered set. Consider* $\trianglerighteq$ *defined as*

$$\forall K \subseteq \mathcal{A}: \quad S|_K \trianglerighteq S'|_K \;\equiv\; f(S|_K) \geq f(S'|_K).$$

If all actions of a transition system satisfy $\trianglerighteq$*, then an invariant of the system is*

$$\textbf{\textit{Invariant}}: \qquad f(S_0) \geq f(S). \qquad \square$$

2.3.2 Progress

Let d be a function from states to a partially ordered set P that has a unique lower bound, G be an invariant of the system, and Q be a predicate on global states of the system. We are interested in sufficient conditions for proving that eventually Q holds. This section lays the ground work for that proof by showing that there are only two possible outcomes for a given fair execution: either Q holds, or d strictly decreases. The following theorem is taken from the literature [17]; we state it here without proof.

Theorem 3 (System Progress) *[17] If the following hold*

D1. $\forall k \in \mathcal{A}, K \subseteq \mathcal{A}$: $G(S) \wedge S \xrightarrow{K} S' \implies d(S) \geq d(S')$

D2. $\exists F_K \in \mathcal{F}, \forall K \in F_K$: $G(S) \wedge \neg Q(S) \wedge S \xrightarrow{K} S' \implies d(S) > d(S')$,

then, for all executions, either

E1. for all $p \in P$, if $d(S) = p$ at any point in an execution, then there is an infinite suffix of the execution where $d(S) < p$ for all states in the suffix:

$$\forall p \in P \colon \Box\,(d(S) = p \implies \Diamond\Box\; d(S) < p)\,,$$

or

E2. every execution has a suffix where Q holds at every point in the suffix:

$$\Diamond\Box\; Q(S').$$

□

Theorem 4 (Local-Global System Progress) *If the following hold*

H1. $\forall K \subseteq \mathcal{A} \colon G(S) \wedge S \xrightarrow{K} S' \implies d(S|_K) \geq d(S'|_K)$

H2. $\exists F \in \mathcal{F}, \forall K \in F \colon G(S) \wedge \neg Q(S) \wedge S \xrightarrow{K} S' \implies d(S|_K) > d(S'|_K)$*, and*

H3. d is a local-global relation with respect to $>$ and $\geq$,

then, for all complete executions, either E1 or E2 holds. □

Proof Since d is a local-global relation, the following implications hold by definition:

$$d(S|_K) \geq d(S'|_K) \implies d(S) \geq d(S') \bigwedge d(S|_K) > d(S'|_K) \implies d(S) > d(S').$$

■

2.4 Related Work

The impact that local interactions can have on the global state of a system has been studied before to different extents and within different contexts. At a very abstract level, biologists

have considered such dynamics when studying living systems, such as molecular development [18, 19] and swarm intelligence [20, 21]. Understanding the nature of such biological systems well enough to mimic their behavior in computing systems is a focus of the amorphous computing [22, 23] and self-assembly communities. Although they have different approaches, both areas study how autonomous processes can build structures or create patterns. Instead of having a blueprint for the final product, these processes possess only instructions on how to interact. The challenge for scientists is to formally describe these instructions. To this end, graph grammars have been considered [24, 25] and new languages developed—growing point language (GPL) [26], origami shape language (OSL) [27], and Proto [28], for example. These efforts take more of an engineering stance toward the process, concentrating on the primitives for solving specific problems, rather than on understanding the advantages and limitations of local interactions.

Recent work by Daniel Yamins has been an attempt to bridge this gap. He too is explicitly interested in formally explaining global structures built from local rules. To characterize local interactions, Yamins defines a function intended to be run over some set of agents, and a means of composing that function—similar to our transition semantics. In early work, his primary applications is to the one-dimensional equigrouping problem [29, 30]. Later work applies this model to other formation problems, and offers deeper insight into its implications [31, 32, 33]. While Yamins is also interested identifying local interactions in much the same way we do, he does not consider it for proof reuse. Interesting future work would be for us to apply our model to his examples.

While local-global relations are meant to ensure global system behavior from local properties, other work, such as Law Governed Interaction (LGI) [34, 35], looks to ensure it at run time. LGI is a framework for the specification and enforcement of local interaction policies; the idea being that global system behavior is a product of correct local interactions. When implemented, LGI is a middleware that mediates agent communication to ensure that a set of predefined laws are being obeyed. Such laws are defined by a system architect and, like our formalization, can inherit from one another [36, 37]. The framework has been applied to various problems, such as spam detection [38] and electronic commerce [39], making it a very

generic solution. Indeed, LGI could enforce that agent interactions fit our local-global model. Moreover, we make the assumption that all agents within the system are homogeneous with respect to their state update—LGI could be utilized to verify this assumption at run time.

Locally stable predicates are another means of identifying global system properties based on local, per-agent, information. A predicate is defined to be *locally stable* if it holds eventually-always for a subset of agents [40, 41]. Work in this area has focused mostly on termination and deadlock detection: algorithms analyze subsets of global snapshots to make a decision about the global state of the system. The analysis can be reduced to Boolean algebra, which is an example of a local-global relation as we have defined them. Thus, locally stable predicates can be described using our framework.

Local-global relations are an *implicit* part of distributed computing. As discussed, there is a wide body of research on understanding, and controlling, their role in problem development; especially given the complexity of contemporary computing systems [42]. Our contribution is to make them an *explicit* part of algorithm design and analysis.

Chapter 3

Consensus Using Monoids

Given a set of agents, the goal of a distributed consensus algorithm is for all agents to come to a consensus value. In this chapter we consider distributed consensus problems specified as follows. Let S_0 be the initial state of the system with $S_0(k)$ the initial value of the agent indexed k. Desired consensus states are specified as a function f from global states $\mathcal{S}$ to local states, $\mathcal{T}$:

$$f : \mathcal{S} \to \mathcal{T}.$$

In the case of the example of computing the minimum, the desired consensus agent state is the minimum of the values of the initial agent states.

A consensus global state $S^\star$ is one in which all agents $k_0, k_1, \ldots, k_{n-1}$ are in the consensus agent state $f(S_0)$,

$$S^\star = \{(k_0, f(S_0)), (k_1, f(S_0)), \ldots, (k_{n-1}, f(S_0))\}.$$

In this chapter we consider problems that require the system to enter, and remain thereafter, in the consensus state $S^\star$. That is, in any infinite fair computation, if S_0 is the initial state then eventually the computation enters a point after which the state is always $S^\star$; in the notation of temporal logic:

$$S_0 \implies \Diamond\Box\, S^\star.$$

In later chapters we will consider problems that require the system to converge to a consensus

state in the limit.

This chapter restricts attention to functions f which is a *folding* [43, 44] of the initial values of the agents with an operator $\oplus$; for example, if the agents are indexed $k = 0, \ldots, n-1$, then

$$f(S) = S(0) \oplus S(1) \oplus \ldots \oplus S(n-1). \tag{3.1}$$

In the case of computing the minimum, $\oplus$ is the min operator. We restrict attention to operators $\oplus$ that are associative and commutative and have an identity element. Thus, the agent states and the operator form a commutative monoid [45]. For completeness we review the definition of monoids.

Definition 4 (Monoid) A monoid consists of the pair, $\langle T, \oplus \rangle$, where T is a set of elements, and $\oplus$ is a binary operation on those elements. The operator $\oplus$ is associative and closed over T, and there exists an identity element in T:

$$\exists a \in T, \forall b \in T : a \oplus b = b \oplus a = b.$$

□

A monoid is said to be *commutative* if $\oplus$ is also commutative with respect to T. For the remainder of this chapter, we restrict attention to commutative monoids where the operator $\oplus$ is idempotent:

$$\forall a \in T : \ a \oplus a = a.$$

We first give lemmas without assuming idempotence, and later give lemmas that assume it.

3.1 Theorems about Monoids

The previous chapter introduced local-global relations and their applicability to distributed algorithms. This chapter illustrates the central goals outlined in the previous chapter; the illustration uses monoid structures. Later chapters use other algebraic structures. Next we give and prove theorems about monoids. The proofs that are mechanically verified in the theorem-proving system, PVS, are given in the appendix (Section A.2).

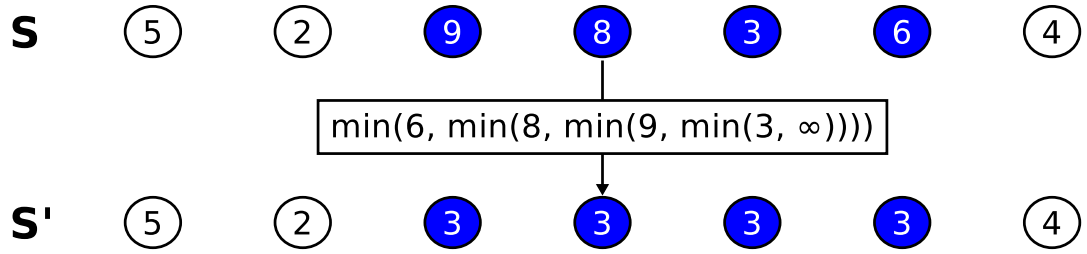


Figure 3.1: An example of monoid composition. Consider a system transition, $S \xrightarrow{K} S'$, where the K is the set of blue agents. The monoid applied during a transition is $\langle \mathbb{N}_0, \min \rangle$, with an identity element of ∞. The sequence of min applications (pictured between states) is an example of how an agent within K uses composition (Definition 6) to update its state.

Definition 5 (Monotonic) Let $\geq$ be an ordering relation over elements in $\mathcal{T}$, where $\forall a \in \mathcal{T} \colon a \geq a$. A binary operation, $\oplus \colon \mathcal{T} \to \mathcal{T}$, is monotonic when

$$\forall a, b, c \in \mathcal{T} \colon a \geq b \implies a \oplus c \geq b \oplus c.$$

□

Let $\oplus$ be the min operator. An example of Definition 5 follows

$$a \geq b \implies \min(a, c) \geq \min(b, c).$$

Definition 6 (Monoid Composition) Let $\langle \mathcal{T}, \oplus \rangle$ be a commutative monoid with identity element $\bar{0}$, where $\oplus$ is also monotonic. Recall that $\mathcal{T}$ is the type of the agent state. Let $K \subseteq \mathcal{A}$. The $\bigoplus$ operation is defined to be the following composition function over the monoid:

$$\bigoplus_{k \in K} S(k) = \begin{cases} \bar{0} & \text{if } K = \emptyset, \\ \forall j \in K \colon S(j) \oplus \bigoplus_{i \in K \setminus \{j\}} S(i) & \text{otherwise.} \end{cases} \tag{3.2}$$

□

Figure 3.1 provides a visual example of Definition 6 where the operation is min. From here, it is left to show that this interaction is local-global. We outline a series of lemmas building to this proof.

Lemma 1

$$\bigoplus_{k \in K \cup \{j\}} S(k) = \bigoplus_{k \in K} S(k) \oplus \begin{cases} \bar{0} & \text{if } j \in K, \\ S(j) & \text{otherwise.} \end{cases} \qquad \square$$

PROOF There are two cases to consider: where j is a member of K, and where it is not. In the former, the lemma holds since $\bar{0}$ is the identity element:

$$\begin{aligned} j \in K \implies & \bigoplus_{K \cup \{j\}} S = \bigoplus_{K} S \oplus \bar{0} \\ \implies & \bigoplus_{K} S = \bigoplus_{K} S \oplus \bar{0} \\ \implies & \bigoplus_{K} S = \bigoplus_{K} S. \end{aligned}$$

The second case holds by definition

$$k \in K \cup \{k\} \implies \bigoplus_{K \cup \{k\}} S = S(k) \oplus \bigoplus_{K \cup \{k\} \setminus \{k\}} S \qquad \bigwedge \qquad (3.3)$$

$$k \notin K \cup \{k\}$$

$$\implies$$

$$\bigoplus_{K \cup \{k\}} S = S(k) \oplus \bigoplus_{K} S, \qquad (3.4)$$

where 3.3 is the definition of fold with $K = K \cup \{k\}$. We can safely assume the antecedent in 3.3, since $k \in K \cup \{k\}$ is a tautology. Further, $K \cup \{k\} \setminus \{k\} = K$. Such rewrites render 3.3 equal to 3.4, and the theorem follows. ■

3.2 System Correctness

Theorem 5 (Monoid Composition is Local-Global) *Let $\geq$ be a transitive relation on elements in $\mathcal{T}$.*

$$\bigoplus_{k \in K} S(k) \geq \bigoplus_{k \in K} S'(k) \ \wedge \ S|_{\{j\}} = S'|_{\{j\}} \implies \bigoplus_{k \in K \cup \{j\}} S(k) \geq \bigoplus_{k \in K \cup \{j\}} S'(k),$$

where $j \notin K$. Note, that when the relation is equality:

$$\bigoplus_{k \in K} S(k) = \bigoplus_{k \in K} S'(k) \ \wedge \ S|_{\{j\}} = S'|_{\{j\}} \implies \bigoplus_{k \in K \cup \{j\}} S(k) = \bigoplus_{k \in K \cup \{j\}} S'(k).$$

□

Proof Follows from Lemma 1 and assumed monotonicity of $\oplus$ with respect to $\geq$. ■

Definition 7 (Consensus Transition) A transition, $S \overset{K}{\rightarrow} S'$ is $\geq$-preserving if and only if

$$S \overset{K}{\rightarrow} S' \implies \bigoplus_{k \in K} S(k) \geq \bigoplus_{k \in K} S'(k).$$

□

Corollary 3 (Local-Global Invariant Property) *If all state transitions preserve $\geq$, then*

$$\textbf{\textit{Invariant}}: \quad \bigoplus_{k \in \mathcal{A}} S_0(k) \geq \bigoplus_{k \in \mathcal{A}} S(k).$$

If all state transitions preserve $=$, then

$$\textbf{\textit{Invariant}}: \quad \bigoplus_{k \in \mathcal{A}} S_0(k) = \bigoplus_{k \in \mathcal{A}} S(k).$$

□

Proof Follows from Theorem 5 and Theorem 2. ■

We have shown that the general monoid structure fits our local-global framework. What remains to be shown are that concrete instantiations of the monoid fit the assumed algebraic

properties outlined in its definition (Definition 4). Others have used recursive operators to simplify inductive proofs [43, 44] for sequential algorithms but not for distributed systems.

Progress

We use Theorem 3 to prove progress. Recall that the theorem requires an invariant of the system G, along with a predicate Q and distance function d on the state space. We introduce a mapping $g\colon \mathcal{S} \times \mathcal{A} \to 2^{\mathcal{A}}$ to facilitate the definition of these elements. An invariant of the system is that for all agents k the state $S(k)$ is the $\oplus$ operator applied to all elements of some set K of agents; let $g(S,k)$ be the largest such set. Initially,

$$\forall k \in \mathcal{A}\colon g(S_0, k) = \{k\}\,.$$

When agents interact, the mapping is updated:

$$S \xrightarrow{K} S' \implies g(S',k) = \begin{cases} g(S,k) \cup \bigcup_{j \in K} g(S,j) & \text{if } k \in K, \\ g(S,k) & \text{otherwise.} \end{cases}$$

Definition 8 (Progress Variables) The predicates G and Q on the state space of the system are

$$G(S) \equiv \forall k \in \mathcal{A}\colon S(k) = \bigoplus_{j \in g(S,k)} S_0(j)$$
$$Q(S) \equiv \forall k \in \mathcal{A}\colon g(S,k) = \mathcal{A}.$$

The variant, or Lyapunov, function d is

$$d(S) = n - \sum_{k \in \mathcal{A}} |g(S,k)|$$

where n is the number of agents in the system, $n = |\mathcal{A}|$. □

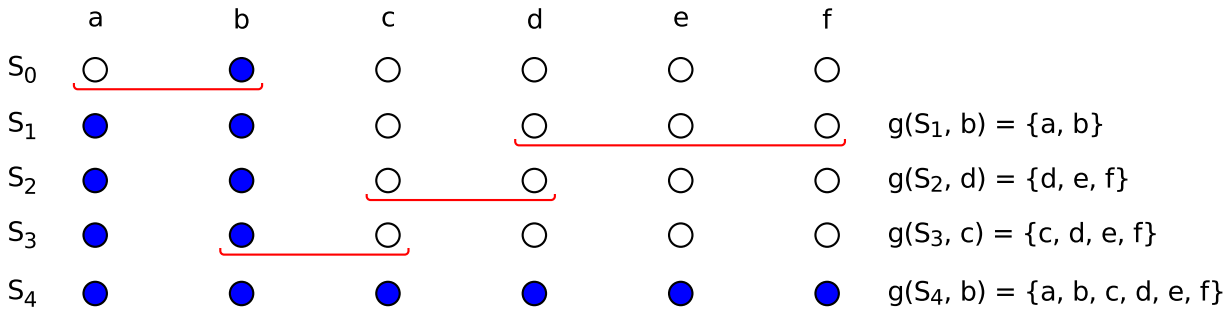


Figure 3.2: Evolution of the variant function g during an execution. Darkened nodes represent the range of g when applied to agent b in a given state; agent interactions are underscored. For example, in state S_2, agents c and d interact, while $g(S_2(b) = \{a, b\}$. Note that the transition from S_3 to S_4 completes the communication graph for agent b even though b has only participated in two transitions.

The predicate G holds in state S if the state of each agent is equal to $\oplus$-composition of the start state, restricted to agents in g; see Figure 3.2 for a visual interpretation. The predicate Q holds in S if the set defined by g holds all agents in the system. Finally, the distance function d is a measure of partitioned agents in a given state.

Lemma 2 *The predicate G is an invariant of the system.* □

Proof The proof is by induction on S:

Base Case Consider the start state, $G(S_0)$:

$$\begin{aligned} S_0(k) &= \bigoplus_{j \in g(S_0,k)} S_0(j) \\ &= \bigoplus_{j \in \{k\}} S_0(j) \\ &= S_0(k). \end{aligned}$$

Inductive Step Assume $G(S_t)$ and $S_t \overset{K}{\rightarrow} S_{t+1}$. Let $J = \bigcup_{j \in K} g(S, j)$. It follows that,

$$\begin{aligned} S_{t+1}(k) &= \bigoplus_{j \in g(S_{t+1}, k)} S_0(j) \\ &= \bigoplus_{j \in g(S_t, k) \cup J} S_0(j). \end{aligned}$$

Because $J \subseteq \mathcal{A}$ the equation holds from properties on the monoid. ■

Theorem 6

$$\forall k \in \mathcal{A}, K \subseteq \mathcal{A} \colon G(S) \wedge S \overset{K}{\rightarrow} S' \implies d(S) \geq d(S')$$ □

Proof From the definition of actions,

$$\begin{aligned} S \overset{K}{\rightarrow} S' &\implies g(S, k) \subseteq g(S', k) \\ &\implies |\mathcal{A} \setminus g(S, k)| \geq |\mathcal{A} \setminus g(S', k)| \\ &\implies d(S_k) \geq d(S'_k) \\ &\implies d(S) \geq d(S'). \end{aligned}$$ ■

Theorem 7

$$\exists F_K \in \mathcal{F}, \forall K \in F_K \colon G(S) \wedge \neg Q(S) \wedge S \overset{K}{\rightarrow} S' \implies d(S) > d(S')$$ □

Proof By definition,

$$\neg Q(S) \implies \exists k \in \mathcal{A} \colon g(S, k) \neq \mathcal{A}.$$

Using this to choose our instance of $F_K \in \mathcal{F}$:

$$F_K = \{(j, k) \mid k \in g(S, k) \wedge j \notin g(S, k)\}.$$

A similar argument to the proof of Theorem 6 follows[1]

$$\begin{aligned} S \overset{(j,k)}{\rightarrow} S' &\implies g(S,k) \subsetneq g(S',k) \\ &\implies d(S) > d(S'). \end{aligned}$$

■

3.3 Instantiations of Monoids

Thus far, our theory has been presented in terms of the generic operator $\oplus$. The advantage of our abstract theory is that when applying it to concrete examples, proofs of system correctness are reduced to algebraic proofs on monoid instances.

Example 6 (Min/Max) We briefly discussed a system built around the minimum operator earlier in this section. To review, the objective of the system is for each agent to contain the lowest value in the system. The dual of this algorithm is for agents to end up with the maximum value. We consider both cases.

Proof Obligation
For a total order $\mathbb{Z}$, $\langle \mathbb{Z}, \min \rangle$, with identity element $+\infty$, forms a commutative monoid that is idempotent. The local-global relation is equality. • $\forall a \in \mathbb{Z}\colon\ \min(a, +\infty) = a$ • $\forall a, b \in \mathbb{Z}\colon\ \min(a, b) \in \mathbb{Z}$ • $\forall a, b \in \mathbb{Z}\colon\ \min(a, b) = \min(b, a)$ • $\forall a, b, c \in \mathbb{Z}\colon\ \min(a, \min(b, c)) = \min(\min(a, b), c)$

A similar set of obligations exist for max.

[1] Where $\geq$ is replaced by $>$.

Definition 9 (Min and Max) The minimum and maximum values over a set of natural numbers S is defined:

$$\min\colon i, j \to \text{if } i < j \text{ then } i \text{ else } j$$
$$\max\colon i, j \to \text{if } i < j \text{ then } j \text{ else } i.$$

□

When calculating the minimum consensus, the monoid is $\langle \mathbb{N}_0, \min \rangle$; when finding the maximum consensus, the monoid is $\langle \mathbb{N}_0, \max \rangle$. Proofs that these structures fit a monoid can be found in Section A.2.2. In either case, the relations $\geq$, in the case of min, and $\leq$, in the case of max, could be substituted for equality. We consider this relation in Chapter 4. □

Example 7 (GCD/LCM) The objective of the system is to agree on the greatest common divisor (gcd) or least common multiple (lcm) of the agents. Before discussing their applicability to monoids and local-global relations, we provide a definition.

Proof Obligation
For a total order $\mathbb{N}_1$, $\langle \mathbb{N}_1, \text{lcm} \rangle$, with identity element 1, forms a commutative monoid that is idempotent.
• $\forall a \in \mathbb{N}_1\colon\ \text{lcm}(a, 1) = a$
• $\forall a, b \in \mathbb{N}_1\colon\ \text{lcm}(a, b) \in \mathbb{N}_1$
• $\forall a, b \in \mathbb{N}_1\colon\ \text{lcm}(a, b) = \text{lcm}(b, a)$
• $\forall a, b, c \in \mathbb{N}_1\colon\ \text{lcm}(a, \text{lcm}(b, c)) = \text{lcm}(\text{lcm}(a, b), c)$

A similar set of obligations exist for gcd.

Definition 10 (Divisibility)

$$\text{divides}\colon i, j \to \exists x\colon j = i \times x$$

□

Definition 11 (LCM and GCD)

$$\gcd\colon i, j \to \max\Big(\{k \mid \text{divides}(k, i) \wedge \text{divides}(k, j)\}\Big)$$
$$\text{lcm}\colon i, j \to \min\Big(\{k \mid \text{divides}(i, k) \wedge \text{divides}(j, k)\}\Big)$$

where i, j, and k are all positive natural numbers.[2] □

For lcm consensus, the proper monoid is $\langle \mathbb{N}_1, \text{lcm}\rangle$; for gcd the monoid is $\langle \mathbb{N}_0, \gcd\rangle$. In both cases, the local-global relation is equality. Formal proofs can be found in Section A.2.2. □

Example 8 (Convex Hull) The convex hull of a given set of points is the minimum set of points in which all other points are contained. Agent state consists of a set of coordinates on a plane. They maintain their current position, as well as set of points that represent the convex hull. Denote by $\mathcal{C}_h\colon P \to P$ the convex hull of a set of points that produces another set of points. When agents interact, they exchange coordinate information and apply $\mathcal{C}_h$ to update their current knowledge about the convex hull. The objective of the system is for all agents to agree on what the convex hull is. The monoid that describes this algorithm is $\langle \mathbb{R}, \mathcal{C}_h\rangle$, where equality is the local-global relation.

[2] In the case of lcm, it is imperative that neither i nor j be zero. For gcd, however, this condition can be relaxed: either variable can be zero, but not both. In the case of our proof sketch, we assume the lcm conditions for both operators; in our formal setting, however, gcd's possible zero value is taken into consideration.

Proof Obligation
For a total order $\mathcal{T}$, $\langle \mathcal{T}, \mathcal{C}_h \rangle$, with identity element $\emptyset$, forms a commutative monoid that is idempotent.
• $\forall a \in \mathcal{T} : \mathcal{C}_h(a,) = a$
• $\forall a, b \in \mathcal{T} : \mathcal{C}_h(a, b) \in \mathcal{T}$
• $\forall a, b \in \mathcal{T} : \mathcal{C}_h(a, b) = \mathcal{C}_h(b, a)$
• $\forall a, b, c \in \mathcal{T} : \mathcal{C}_h(a, \mathcal{C}_h(b, c)) = \mathcal{C}_h(\mathcal{C}_h(a, b), c)$

Proofs of such can be found in Section A.2.2. □

3.4 Message Passing

Consider a lossy message-passing medium discussed earlier where messages may be lost, duplicated an delivered out of order. The fairness requirement is that for any set K of agents, a message from some agents in $\bar{K}$, the complement of K, reaches some agent in K infinitely often.

We treat the communication medium as an agent with a different state space. Let $\mathcal{M}$ be the state of the communication medium; then $\mathcal{M}$ is a bag, or multiset, of messages in transit. Proof of the invariant

$$\textbf{Invariant}: \quad \left(\bigoplus_{k \in \mathcal{A}} S_0(k) = \bigoplus_{k \in \mathcal{A}} S(k) \right) \bigwedge \left(\bigoplus_{k \in \mathcal{A}} S_0(k) = \bigoplus_{k \in \mathcal{A}} S(k) \oplus \bigoplus_{m \in \mathcal{M}} S(m) \right)$$

follows from the properties of monoids. Proof of the progress property is identical to the case where agents interact directly without messages since the monoid is assumed to be idempotent.

Chapter 4

Distributed Path Computations using Semirings

Consider a distributed computation of shortest paths in a directed graph in which there is an agent at each vertex. An agent's state includes information about the weights of edges incident on that agent's vertex. We assume that the graph is strongly connected and that there are no negative-weight cycles in the graph. All agents compute the length of the shortest path to them from a special agent called the "source." The problem of computing the path from the source to all agents, is called the single-source problem; it can be extended in a straightforward way to the all-points shortest path problem in which all agents compute the lengths of the shortest paths to all other agents.

This problem occurs in Internet protocols in which each router determines the minimum-congestion paths to other routers. Distributed algebraic computations are abstractions of distributed shortest-path computations in which distances, possibly real numbers, and the operations of minimum and addition are replaced by operations in a semiring. The goal, as described in the first chapter, is to reuse abstract theorems that are verified by a mechanical theorem proving system for multiple concrete implementations. The same goal and method were used in the previous chapter; in this chapter we use semirings whereas we used monoids in the previous chapter.

4.1 Central Idea

We describe the idea starting with the shortest path algorithm. Assume that vertices are indexed $k = 0, 1, \ldots, N-1$, for $N > 0$. Let $W[j,k]$ be the weight of the edge from vertex j to vertex k if the directed edge (j,k) exists. The problem is to compute the shortest path from a vertex, called the "source," to all other vertices. Let us assume that the source is vertex 0. If there is no path from the source to a vertex k, then the length of the shortest path to vertex k is infinity. Let $D[k]$ be the length of the shortest path from vertex 0 to vertex k. Since no cycles of negative length exist, $D[0] = 0$.

Each vertex is associated with an agent. The state of agent k includes the values of the edge weights $W[j,k]$ for all j, and the set of vertices to which there there is an outgoing edge from k.

A distributed version of the well-known sequential algorithm is as follows. Associated with each agent k is a local variable $w[k]$ which eventually becomes $D[k]$, the shortest distance from the source to vertex k. Also, each agent k has a local variable *parent*$[k]$ which eventually becomes the prefinal vertex on the shortest path from the source to k; in other words a shortest path from the source to vertex k goes from the source to vertex *parent*$[k]$ and then traverses the edge from *parent*$[k]$ to k.

Initial Condition The initial condition of the algorithm is:

$$w[source] = 0$$
$$\forall k \neq source\colon w[k] = \infty$$
$$\forall k\colon parent[k] = source.$$

Rules of the Algorithm The algorithm is given by a set of rules, with one rule for each ordered pair, (j,k):

if $w[k] > w[j] + W[j,k]$ **then**
$w[k] \leftarrow w[j] + W[j,k]$
$parent[k] \leftarrow j$

end if.

We refer to the inequality condition as the *triangle property*.

Rules are selected non-deterministically. The fairness criterion is that every rule is executed infinitely often.

Invariant An invariant of the algorithm is that for all k, $w[k]$ is either infinity or it is the length of a path from the source to k that goes from the source to $parent[k]$ and then along the edge from $parent[k]$ to k. We give a proof of this invariant for the general case of semirings later.

Progress We prove progress using a variant function in the usual way: we show that (i) for all actions, execution of the action does not increase the value of the variant function, and (ii) if the desired predicate is not reached then there exists an action that is executed infinitely often that decreases the variant function.

A variant function for this problem is as follows. For each vertex k, rank order all the cycle-free paths from the source to a vertex k in increasing order of distance and index the paths with $0, 1, 2, \ldots$ in the sequence, with the shortest path having index 0 and a fictitious path of infinite length having the largest index. There are a bounded number of such cycle-free paths in a finite graph. In each state, the value of $w[k]$ corresponds to a path from the source to vertex k and therefore corresponds to an index in this sequence; let us call this index $r[k]$. The variant function f is the sum of the indexes of all agents.

$$f = \sum_{k \in \mathcal{A}} r[k]$$

We first show that the variant function does not increase in value as computation proceeds. For any agent k, the execution of any action decreases $w[k]$ or leaves it unchanged; therefore the execution of any action does not increase $r[k]$.

We next show that if the desired predicate (for all k, $w[k] = D[k]$) does not hold then there exists some action, which is executed infinitely often, and which decreases the

variant function. To do so we show that if the desired predicate does not hold then there exists some edge (j, k) such that the triangle property does not hold:

$$w[k] \not> w[j] + W[j, k].$$

Discussion Consider another path problem, such as computing reachability of vertexes from a source vertex. We could carry out a similar argument as for the shortest path problem. The use of abstract algebra reduces the amount of work required to prove the correctness of distributed algorithms for similar sorts of problems.

The arguments for correctness of the algorithm given above cannot be verified by a mechanical proof checker. This is because the arguments use natural, English-like, language to talk about concepts in graphs. A great deal of effort is required to develop proofs that can be verified by a program; this effort is amortized over several problems by presenting and proving an algorithm using data structures from abstract algebra.

4.2 Semirings

We extend the monoid introduced in Section 3.1 and applied in Chapter 3, to semirings.

Definition 12 (Semiring) A semiring consists of the quintuple, $\langle \mathcal{T}, \oplus, \otimes, \bar{0}, \bar{1} \rangle$, such that

- $\langle \mathcal{T}, \oplus \rangle$ is a commutative monoid with identity element $\bar{0}$,
- $\langle \mathcal{T}, \otimes \rangle$ is a monoid with identity element $\bar{1}$,
- $\otimes$ distributes over $\oplus$, and
- $\bar{0}$ is an annihilator[1] when used with $\otimes$. □

Definition 13 (Idempotent Semiring) An idempotent semiring is a semiring where

[1] $\forall a \in \mathcal{T} \colon a \otimes \bar{0} = \bar{0}$

- $\oplus$ is idempotent, and
- $\bar{1}$ is an annihilator for $\oplus$. □

Theorem 8 (Semiring Partial Order) *Given an idempotent semiring, $\succeq$ defines a partial order over $\mathcal{T}$ such that $\forall a, b \in \mathcal{T} : b \succeq a \iff a \oplus b = a$.* □

Proof Reflexivity follows directly from the definition and idempotence. Anti-symmetry: $a = a \oplus b = (a \oplus b) \oplus b = (b \oplus a) \oplus b = b \oplus b = b$. Transitivity: $a = a \oplus b = a \oplus (b \oplus c) = (a \oplus b) \oplus c = a \oplus c$. ■

Theorem 9 (Bounded Semiring) *Given an idempotent semiring,*

$$\forall t \in \mathcal{T} : \bar{0} \succeq t \succeq \bar{1}.$$

□

Proof From Definition 12, $t \oplus \bar{0} = t$; by Theorem 8, $t \oplus \bar{0} = t \implies \bar{0} \succeq t$. Likewise, from Definition 13, $t \oplus \bar{1} = \bar{1}$; by Theorem 8, $t \oplus \bar{1} = \bar{1} \implies t \succeq \bar{1}$.[2] ■

As an addendum to Definition 13, we assume that $\oplus$ is monotonic with respect to $\succeq$.

4.3 System Specification

System state is a mapping from agents to pairs:

$$S \colon \mathcal{A} \to \left(o \colon \mathcal{A} \to \mathcal{T}, i \colon (\mathcal{A}, \mathcal{T})\right)$$

where

o is an agent's set of neighbors and the cost associated with contacting them. Unless an agent is fully connected, this function is partial. Moreover, the range defines the set

[2]Note that in some texts, Theorem 8 is presented as $a \succeq b \iff a \oplus b = b$. In this case the bounded ordering is reversed: $\bar{1} \succeq t \succeq \bar{0}$. The proof is symmetric.

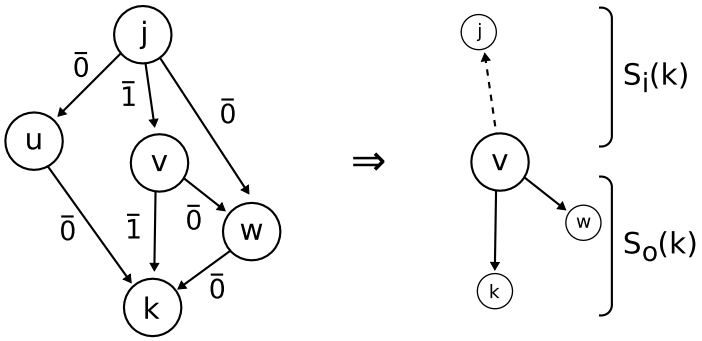


Figure 4.1: Components of agent state as it relates to an arbitrary graph.

of neighbors an agent is able to communicate with. In problems we consider, both the range and domain of o are static throughout an execution.

i is the parent of a given agent and value of the edge connecting them. Unlike o, this value is not a set and is mutable over an execution. We use subscript notation to denote the extraction of an element from i: $S_{i,1} \colon \mathcal{A} \to \mathcal{A}$, and $S_{i,2} \colon \mathcal{A} \to \mathcal{T}$.

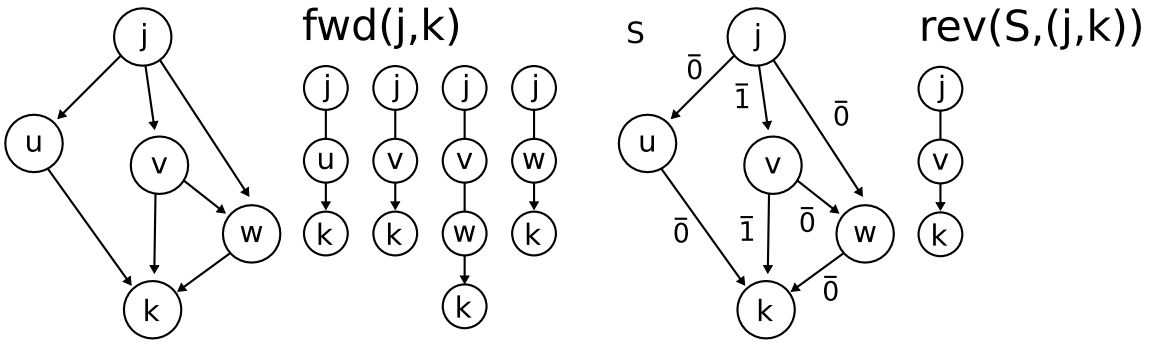
In this way, the state of the system is a distributed representation of a directed, weighted, graph: the function o is an agent's set of outgoing vertices, and i its incoming edges (see Figure 4.1). Given an agent j containing a directed edge to agent k, the weight of the edge from j to k is denoted $S_o(j)(k)$. For convenience, we express elements of the agent state with $W \colon \mathcal{A} \times \mathcal{A} \to \mathcal{T}$ and $w \colon \mathcal{S} \times \mathcal{A} \to \mathcal{T}$. For all agents $j, k \in \mathcal{A}$:

$$W(j,k) \equiv S_o(j)(k)$$
$$w_S(k) \equiv S_{i,2}(k).$$

Finally, since o is a partial, we use the predicate E to denote the domain of definition: $E(j,k)$ holds if $W(j,k)$ exists.

Definition 14 (Path) A *path* is a set of agent pairs, $\mathcal{P} = \{(i,j) \mid E(i,j)\}$. □

There are two ways of specifying paths amongst a set of agents.



(a) A forward path, defined by fwd, is the set of all paths from j to k.

(b) A reverse path, defined by rev, is a single path from j to k. This path is the optimal path between the two agents, denoted here with the edges of value $\bar{1}$. Recall that $\forall a \in \mathcal{T}\colon a \in [\bar{1}, \bar{0}]$.

Figure 4.2: Defined incoming and outgoing path functions.

Definition 15 (Forward Path) A forward path is the set of paths from one agent to another (Figure 4.2(a)). It is specified using $\text{fwd}\colon \mathcal{A} \times \mathcal{A} \to 2^{\mathcal{P}}$. Let $P = \text{fwd}(j, k)$,

$$\big(\forall\,(u,v) \in P\colon u \neq v\big) \bigwedge \big(\exists\,(u,v) \in P\colon u = j\big) \bigwedge \big(\exists\,(u,v) \in P\colon v = k\big).$$

□

Definition 16 (Reverse Path) A reverse path is a single path between agents (Figure 4.2(b)), $\text{rev}\colon \mathcal{S} \times \mathcal{A} \times \mathcal{A} \to \mathcal{P}$, where

$$\forall\,(u,v) \in \mathcal{P}\colon\ (u,v) \in \text{rev}(S,(j,k)) \implies S_{i,1}(v) = u$$

for all $P \in \mathcal{P}$ and $(j,k) \in P$. □

Definition 17 (Path Traversal) Path traversal is a function, $\delta\colon \mathcal{P} \to \mathcal{T}$ such that

$$\delta(P) = \bigotimes_{(j,k)\in P} W(j,k).$$

□

We consider algebraic path problems that are *single-source*, meaning that we find the optimal path from a single agent, the *root*, to all other agents within the system; the root agent is denoted $\hat{r}$. In the initial state, $S_0 \in \Lambda$,

$$\forall k \in \mathcal{A}\colon w_{S_0}(k) = \bar{0} \ \wedge \ S_{0_{i,1}}(k) = k. \tag{4.1}$$

The goal of the system is reach a state in which

$$\forall k \in \mathcal{A}\colon w_{S^\star}(k) = \bigoplus_{P \in \mathrm{fwd}(\hat{r},k)} \delta(P).$$

From the root the cost of the paths to each agent using incoming path information is the same as the lowest cost route using the outgoing path information. Recall that, for each agent, the incoming weight is a single value that is dynamic over an execution while outgoing information is static—optimal routes using this information are invariant.

Definition 18 (Algebraic Path Transition) State transitions are the result of pairwise interactions between neighboring agents. Assuming $W(j,k)$ is defined, an interaction between j and k follows

$$S \overset{\{j,k\}}{\rightarrowtriangle} S' \implies S'_i(k) = \begin{cases} (j, w_S(j) \otimes W(j,k)) & \text{if } w_S(k) \succeq w_S(j) \otimes W(j,k), \\ (k, w_S(k)) & \text{otherwise.} \end{cases}$$

Agents other than j and k remain unchanged. □

Recall that in the description of i, the second element of an agent state, the value was mutable—the state transition is where this mutation occurs. Moreover, this mutation is based on the "optimal" incoming edge. Thus, not only is $S_i(k)$ the best parent of agent k in state S, but, by construction, any incoming path to k is also optimal.

Typically in the literature, local state transitions are expressed using both operations of the semiring. The following lemma shows that our transition semantics are general enough to encompass this case.

Lemma 3 $S \overset{\{j,k\}}{\rightarrow} S' \implies w_{S'}(k) = w(k) \oplus \big(w_S(j) \otimes W(j,k)\big)$ □

Proof Start with the case in which $w_S(k) \succeq w_S(j) \otimes W(j,k)$: From Theorem 8,

$$w_S(k) \succeq w_S(j) \otimes W(j,k) \implies w(k) \oplus \big(w_S(j) \otimes W(j,k)\big) = w_S(j) \otimes W(j,k)$$
$$= w_{S'}(k).$$

The proof for the second case, in which $w_S(k) \not\succeq w_S(j) \otimes W(j,k)$, is symmetric. ■

The advantage of expressing the system transition as we have in Definition 18, as opposed to the traditional way (Lemma 3), is that there is a cleaner separation of the updated optimal node and the path value to that node.

System transitions maintain a local-global relation with respect to $\succeq$. Within the domain of algebraic path problems, local-global relations revolve around paths. Thus, to fit Definition 3, consider j to be a single path, and K a collection of paths.

Lemma 4 (Algebraic Path Local-Global Relation) *Let (j,k) be a valid reverse path in S and S', and u be an agent in the system.*

$$\delta(\text{rev}(S,(j,k))) \succeq \delta(\text{rev}(S',(j,k))) \implies \delta(\text{rev}(S,(j,k)) \cup \{u\}) \succeq \delta(\text{rev}(S',(j,k)) \cup \{u\})$$

where the union operation maintains the path. □

Proof The codomain of rev is defined by the transition, which, from Lemma 3, is derived using a commutative monoid. Since, $\succeq$ is transitive by definition, the lemma follows from Theorem 5. ■

4.4 System Correctness

Recall that the theorem on progress requires an invariant of the system G, along with a predicate Q, and distance function d, on the state space. We introduce external functions that aid in their definition and subsequent correctness proofs.

Let $p\colon \mathbb{N}_0 \times \mathcal{A} \to \mathcal{T}$ be an indexed path from the root to k; denoted $p_i(k)$ for some agent k. The index, known as the p-index, denotes a path's ordering with respect the optimal path. That is,

$$p_0(k) = \bigoplus_{P \in \text{fwd}(\hat{r},k)} \delta(P).$$

As i increases, so to does the corresponding path weight:

$$\forall i, j \in \mathbb{N}_0 \colon j > i \implies p_j(k) \succeq p_i(k).$$

We restrict attention to non-cyclic paths and the initial path; thus, p's index ranges from 0 to $|\text{fwd}(\hat{r}, k)| + 1$.

Let $g\colon \mathcal{S} \times \mathcal{A} \to \mathbb{N}_0$ return the p-index within a given state:

$$g(S, k) = i \text{ where } p_i(k) = \text{rev}(S, (\hat{r}, k)).$$

Definition 19 (Progress Variables) The predicates G and Q on the state space of the system are

$$G(S) \equiv \forall k \in \mathcal{A} \colon w_{S_0}(k) \succeq w_S(k)$$
$$Q(S) \equiv w_S(k) = \bigoplus_{P \in \text{fwd}(\hat{r},k)} \delta(P).$$

The variant function d is

$$d(S) = \sum_{k \in \mathcal{A}} g(S, k).$$

□

The predicate G holds if the path maintained by all agents to the root in state S is "better-than" the optimal path in the start state. The predicate Q holds if the path maintained by all agents to the root in state S is globally optimal. The variant function d is a measure of the number of optimal paths remaining to be discovered.

Theorem 10 *The predicate G is an invariant of the system.* □

PROOF Follows since $\succeq$ is a local-global relation with respect to the monoid $\oplus$ (Lemma 4 and Corollary 3). ■

Theorem 11

$$\forall k \in \mathcal{A}, K \subseteq \mathcal{A}\colon G(S) \wedge S \overset{K}{\rightarrow\!\!\!\!\triangleright} S' \implies d(S) \geq d(S')$$

□

PROOF Follows directory from the predicate G: if the optimal path during an execution does not increase, neither will its p-index. ■

Theorem 12

$$\exists F_K \in \mathcal{F}, \forall K \in F_K\colon G(S) \wedge \neg Q(S) \wedge S \overset{K}{\rightarrow\!\!\!\!\triangleright} S' \implies d(S) > d(S')$$

□

PROOF By definition,

$$\begin{aligned}
\neg Q(S) &\implies \exists k \in \mathcal{A}\colon w_S(k) \neq \bigoplus_{P \in \mathrm{fwd}(\hat{r},k)} \delta(P) \\
&\implies \exists k, j \in \mathcal{A}\colon \big(w_S(k) \succeq w_S(j) \otimes W(j,k)\big) \bigwedge \big(w_S(k) \neq w_S(j) \otimes W(j,k)\big). \quad (4.2)
\end{aligned}$$

Let F_K be the family of edge sets such that the agents in each edge are valid instantiations of k and j in Equation 4.2. Let u and v be such agents, respectively; by construction,

$$\begin{aligned}
S \overset{\{u,v\}}{\rightarrow\!\!\!\!\triangleright} S' &\implies w_{S'}(v) \not\succeq w_S(v) \\
&\implies g(S,k) > g(S',k) \\
&\implies d(S) > d(S')
\end{aligned}$$

■

4.5 Examples

We restrict attention to the application of semirings to vertex reachability and shortest path calculations. Because shortest path calculations require the notion of a graph path, they

are a natural refinement to the reachability problem. We define the semiring used in each algorithm, and prove that the elements of the semiring meet the requirements assumed in the Section 4.2.

Example 9 (Reachability) The objective of the graph reachability problem is to determine what vertices have a path from a given root vertex. It can be solved using the *Boolean* semiring.

Proof Obligation
$\langle \{0,1\}, \vee, \wedge, 0, 1 \rangle$ is an idempotent semiring.

Graph transitions take the form

$$w_{S'}(k) = w(k) \vee \big(w_S(j) \wedge W(j,k)\big).$$

□

Example 10 (Shortest Path) We have used the shortest path as an ongoing example throughout the chapter. We state it again here, formally.

Proof Obligation
$\langle \Re_+ \cup \{+\infty\}, \min, +, +\infty, 0 \rangle$ is an idempotent semiring.

The range of W and f, $\mathcal{T}$, are known as *weights*. Because of our range-subset restriction, $(0, +\infty)$, we only consider graphs with positive weights. Shortest path transitions follow

$$w_{S'}(k) = \min\big(w_S(k), w_S(j) + W(j,k)\big).$$

□

Example 11 (Minimum Spanning Tree) As was the case in the previous example, we only consider graphs with positive weights.

Proof Obligation
$\langle \Re_+ \cup \{+\infty\}, \min, \max, +\infty, 0 \rangle$ is an idempotent semiring.

Transitions consist of

$$w_{S'}(k) = \min\big(w_S(k), \max\left(w_S(j), W(j,k)\right)\big).$$

□

Example 12 (Viterbi Algorithm) The Viterbi Algorithm [46] can determine the likelihood of unknown events by analyzing those that are known. Within a graph, events are represented as vertices. The edges between them not only describe their occurrence in time, but their ordering; thus, edge weights are probabilities of event sequence. In this way, the problem is an algebraic path one, and a semiring can be used to find a solution [47].

Proof Obligation
$\langle [0,1], \max, \times, 0, 1 \rangle$ is an idempotent semiring.

The corresponding transition is

$$w_{S'}(k) = \max\big(w_S(k), w_S(j) \times W(j,k)\big).$$

□

4.6 Related Work

A large body of work has been dedicated to applying semirings to this formalization, where solutions to path problems are repeated applications of operators within the structure [48, 49, 50, 51]. Such theoretic work has found several practical applications, especially with respect to systolic arrays. Systolic arrays are multi-processor networks notable for their efficiency at performing matrix multiplication. Several methodologies for implementing semiring-based

solutions to path problems have been offered [52, 53, 54, 55, 56]. A comprehensive survey of this area was reported by Fink [57].

Several more path-based contributions have been made, each with slightly different formalisms and varying degrees of generality. Lehmann, for example, showed that many previously considered problems, such as matrix inversion and proofs about regular languages, were just instances of transitive closure. This was known prior to this work, but not correctly formalized. He went on to show how they could be solved using using semirings [58]. Lehmann's foundation was later extended and applied to new domains [59, 50, 47]. Of particular note is the especially generic and encompassing framework presented by Mohri [60]. The solution he offers relaxes both semiring and underlying system assumptions used by researchers prior.

The derivations presented in this chapter are similar to the formalizations presented by previous authors. Our model most closely resembles that of Mohri, however our system setup and semiring assumptions are more appropriate for a distributed setting. Where our work differs most is in our use of local-global relations as a vehicle for correctness. This not only simplifies correctness proofs, but absolves us from a rigid path traversal order for showing convergence. Such an order was implicit in some previous models, but it is something that we are able to make explicit thanks to our framework.

Chapter 5

Sorting

This chapter studies a problem that exhibits local-global in which the abstract data structure is a total order. Modeling the system and proving correctness, is similar to the process followed in Chapters 3 and 4. Actions are executed by subgroups of agents; an action by a subgroup may change the states of agents in the subgroup, leaving the states of agents outside the subgroup unchanged.

Sorting a group of elements is a well-studied problem within computer science, even within the distributed system community [61]. At an abstract level, sorting is an ordering of elements within a permutation [62]. This section models the abstraction as a local-global relation within a distributed system.

5.1 System Specification

As is case throughout this thesis, the system consists of a non-empty finite set of agents. The state of an agent k in S consists of an arbitrary value from the set $\mathcal{T}$. The concrete family of $\mathcal{T}$ is not important, only that the set is ordered.

We assume that there exists a strict ordering over agents and agent values: for all $j, k \in \mathcal{A}$, either $j < k$ or $k < j$; likewise for members of $\mathcal{T}$. We also assume that within a state, agent values are distinct:

$$\forall j, k \in \mathcal{A}, S \in \mathcal{S} \colon j \neq k \implies S(j) \neq S(k).$$

In this way, states are bijective mappings and can be inverted to return corresponding agents:

$$t = S(k) \implies S_{-1}(t) = k.$$

Because values in $\mathcal{A}$ and $\mathcal{T}$ are unique and ordered, we consider collections of each to be finite sequences, rather than sets. Let $K \subseteq \mathcal{A}$, elements within K are referred to using their relative index value, and obey

$$\forall i, j \in \mathbb{N}_0 \colon (i < |K| \implies k_i \in K) \land (i < j \implies k_i < k_j).$$

That is, based on the global ordering of agents, we assume that when the state is restricted to a subset, an ordering can be created over that subset as well. Moreover, that subset ordering is continuous in $\{K\}$.

The specification of the system is that the system eventually enters a state at and after which the following predicate holds:

$$\forall j, k \in \mathcal{A} \colon j < k \implies S(j) < S(k).$$

Definition 20 (Permutation) Let ρ be a binary predicate between states, $\rho \colon \mathcal{S} \times \mathcal{S} \to \mathbb{B}$ that holds if the states are permutations of one another. That is, there exists a bijective function that is also a mapping between agents in distinct states;

$$\rho(S, S') \implies \exists (h \colon \mathcal{A} \to \mathcal{A}) \colon \forall j, k \in \mathcal{A} \colon h(j) = h(k) \implies j = k \qquad \bigwedge \qquad (5.1)$$

$$\forall j \in \mathcal{A} \colon \exists k \in \mathcal{A} : h(k) = j \qquad \bigwedge \qquad (5.2)$$

$$\forall k \in \mathcal{A} \colon S(k) = S'(h(k)). \qquad (5.3)$$

□

Equation 5.1 and 5.2 make up the bijection, defining h to be an injection and surjection, respectively. Equation 5.3 uses h to map agents between permuted states.

Lemma 5 (Permutations Are Transitive)

$$\rho\left(S, S'\right) \wedge \rho\left(S', S''\right) \implies \rho\left(S, S''\right) \qquad \square$$

Proof From the antecedent, we can assume there exist two functions, $(f_1, f_2) : \mathcal{A} \to \mathcal{A}$, such that both are bijections and

$$\forall k \in \mathcal{A}\colon S(k) = S'(f_1(k)) \wedge S'(k) = S''(f_2(k)).$$

We use the composition of these functions, $f_2 \circ f_1$, to instantiate h in the consequent. See Section A.4 for details that this composition is a permutation. ■

Definition 21 (Transposition) Let (k, j) be a pair of agents in $\mathcal{A}$, $\otimes$ exchanges the values of k and j depending on their order with respect to $\mathcal{A}$ and on the total order of their values. Formally, $\otimes\colon \mathcal{S} \times (\mathcal{A}, \mathcal{A}) \to 2^{\mathcal{A}}$, where

$$\otimes(S, (k, j)) = \begin{cases} \{(k, S(j)), (j, S(k))\} & \text{if } \min(k, j) = k \bigwedge S(\max(k, j)) < S(\min(k, j)), \\ \{(k, S(k)), (j, S(j))\} & \text{otherwise.} \end{cases} \qquad \square$$

We use the following notation to represent the extraction of elements from the the range of $\otimes$ in particular, and a pair of agents in general. Let $S' = \otimes(S, (k, j))$; the element k in S' can be referenced as

$$\{(j, S'(j))\} = \left[\otimes\left(S, (k, j)\right)\right]_k$$
$$S'(j) = \otimes(S, (k, j))(j).$$

Lemma 6 (Transposition Permutes) *Transposition produces a permutation of its input:*

$$\forall i, j \in \mathcal{A}\colon \ \rho\left(S|_{\{i,j\}}, \otimes(S, (i, j))\right),$$

where $S \in \mathcal{S}$ and $S|_{\{i,j\}}$ is the restriction of S to elements i and j. □

PROOF We must show that there exists a mapping from agents onto themselves that is bijective. Formally,

$$\exists(h\colon \mathcal{A} \to \mathcal{A})\colon B(h) \wedge \forall k \in \mathcal{A}\colon S(k) = \otimes(S,(i,j))(h(k)),$$

where B is a predicate over $\mathcal{A} \to \mathcal{A}$ that holds if the mapping is bijective. Let $\hat{S} = \otimes(S,(i,j))$. We define h to be the composition of $\hat{S}$'s inverse and S:

$$B(\hat{S}^{-1} \circ S) \qquad \bigwedge \tag{5.4}$$

$$\forall k \in \mathcal{A}\colon S(k) = \hat{S}(\hat{S}^{-1}(S(k))). \tag{5.5}$$

In 5.4, S is bijective by definition and $\hat{S}$ by construction—composition of two bijective functions is also bijective. Equation 5.5 holds trivially. ■

As previously mentioned, composition transposes groups of agents. It does so by recursively operating on "neighboring" agents in a given set.

Definition 22 (Composition) Composition applies a transposition to a sequence of agents. Formally, $\bigoplus\colon \mathcal{S} \times 2^{\mathcal{A}} \to 2^{\mathcal{A}}$, such that

$$\bigoplus_{(k_0,k_1)\in K} S = \begin{cases} \{(k_0, S(k_0))\} & \text{if } |K| = 1, \\ \left[\otimes(S,(k_0,k_1))\right]_{k_0} \cup \bigoplus_{K'} S & \text{otherwise,} \end{cases}$$

where K is a non-empty sequence of agents, and $K' = \left[\otimes(S,(k_0,k_1))\right]_{k_1} \cup (K \setminus \{k_0,k_1\})$.□

System transitions are the result of compositions amongst groups of agents.

Definition 23 (Sorting Transition) Let $K \subseteq \mathcal{A}$. Sorting transitions follow

$$S \overset{K}{\rightarrowtriangle} S' \implies \forall k \in K\colon S'(k) = \left(\bigoplus_K S\right)(k)$$

where agents not in K remain unchanged. □

Because the result of a composition is a permutation, system transitions are local-global relations.

Theorem 13 (Permutations Are Local-Global)

$$S \xrightarrow{K} S' \implies \left(\rho\left(S|_K, S'|_K\right) \implies \rho\left(S|_{K\cup\{j\}}, S'|_{K\cup\{j\}}\right)\right),$$

where $S(j) = S'(j)$. □

Proof The proof is by induction on K:

Base Case The cardinality of K is 1. In this case, h in Definition 20 is the identity function.

Inductive Step We induct again on K.

Base Case The cardinality of K is 2. Proof follows from Lemma 6.

Inductive Step By definition, the additional element to the computation is unique. Thus, since the union of an element to a permutation is still a permutation, the global permutation remains. ■

5.2 System Correctness

Conservation

Lemma 7 (Sorting Conservation) *If all state transitions preserve a global permutation, then*

$$\textbf{\textit{Invariant}}: \qquad \rho\left(S_0, S\right).$$

□

Proof Follows from Lemmas 5 and 13. ■

Progress

Before going through the typical steps for showing system liveness, we define the concept of an *inversion*.

Definition 24 (Inversion) An *inversion* is a set of agent pairs from a given state whose values are out of order with respect to the pairs' agent ordering;

$$\operatorname{inv}(S) = \{(j,k) \mid j < k \wedge S(k) < S(j)\}\,.$$

□

The number of inversions in a finite set both finite and bounded below. As such, its codomain forms a well-founded set that is appropriate for reasoning about an execution. We continue with the standard definitions of the progress variables.

Definition 25 (Progress Variables) The predicates G and Q on the state space of the system are

$$G(S) \equiv \rho\,(S_0, S)$$
$$Q(S) \equiv \forall j,k \in \mathcal{A}\colon j < k \implies S(j) < S(k).$$

The distance function d counts the number of inversions in a given set.

$$d(S) = |\operatorname{inv}(S)|\,.$$

□

Lemma 8 *The predicate G is an invariant of the system.* □

Proof Follows from Lemma 7. ■

Theorem 14

$$\forall k \in \mathcal{A}, K \subseteq \mathcal{A}\colon G(S) \wedge S \xrightarrow{K} S' \implies d(S) \geq d(S')$$

□

PROOF Follows from the definition of transposition (Definition 21) and that transitions are repeated applications of transpositions. ■

Theorem 15

$$\exists F_K \in \mathcal{F}, \forall K \in F_K \colon G(S) \wedge \neg Q(S) \wedge S \xrightarrow{K} S' \implies d(S) > d(S') \qquad \square$$

PROOF The negation of Q provides a set of out-of-order agents:

$$\neg Q(S) \implies \exists j, k \in \mathcal{A} \colon j < k \wedge S(k) < S(j).$$

We use this statement to define the fair set of transitions

$$F_K = \{K \mid (\exists j, k \in K \colon j < k \wedge S(k) < S(j))\}.$$

From Theorem 14 we know that transitions will either reduce the number of inversions or keep them the same. Since, by definition, agent values are distinct and the set K has at least one inversion, we can rule out the latter. ■

Chapter 6

Average Consensus

This chapter, like the last, examines a well known problem in the distributed systems community: distributed averaging. The results in this chapter extend the large amount of work in the literature by reposing the problem with a focus on its inherent local-global relation. As has been the case throughout this thesis, we are able to show algorithm correctness based in part on this relation.

6.1 Background and Motivation

Distributed averaging calculates the average of a collection of data spread across several autonomous processes. We define it in this setting as the *algebraic mean*:

Definition 26 (Algebraic Mean) The algebraic mean is a function, $\mathrm{AM}\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, such that

$$\mathrm{AM}(S, K) = \frac{1}{|K|} \sum_{k \in K} S(k),$$

where $S \in \mathcal{S}$ and $K \subseteq \mathcal{A}$. □

Performing this computation in a distributed setting lies at the heart of several practical applications. Many peer-to-peer networks, for example, use distributed averaging to estimate active network size [63]. Coordination problems, such as vehicle or pattern formation, rely on the distributed averaging to determine the center of mass [64]. Sensor networks that make physical measurements for things like data fusion or inference often compute a

distributed average [65, 66]. Networks of servers calculate distributed averages for proper load balancing [67]. Given this wealth of applications, distributed averaging has received quite a bit of attention within the research community, where several protocols, and improvements to protocols, have been offered. Schemes have become increasingly distributed, asynchronous [68], and more robust—both with respect to network dynamics [69] and communication variability [70, 71, 72]. Much of this work is focused on conditions, and rates, of convergence.

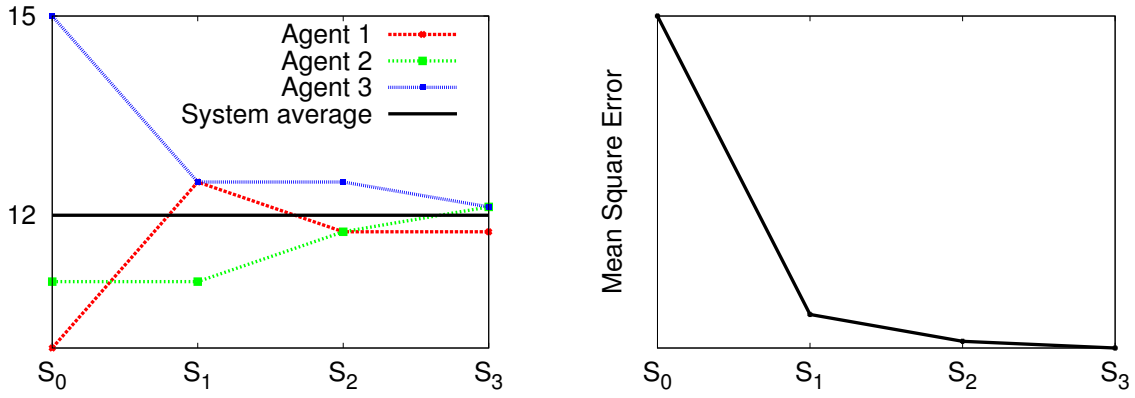
We offer a method of describing and reasoning about the distributed average problem that fits the local-global framework outlined in Section 2.2. We express the solution algebraically, which has benefits when reasoning about the problem using a theorem prover. Our scheme does not expect a static network topology, only that the communication graph is always eventually connected.

Distributed averaging is in fact a consensus problem—the objective of each process is to agree on the average of agents' values within the system. However, unlike other consensus problems considered in this thesis, there is not a generic algebraic framework to describe it. In this way, we say that distributed average uses the local-global relations directly, rather than using, for instance, a monoid proxy.

Before detailing the system specification and proofs of correctness, we provide an example. This example not only outlines the problem solution, but offers intuition as why our approach works.

Example 13 (Distributed Average with 3 Agents) Consider a system with 3 agents. Suppose the initial state of the system is

$$S_0 = \{(k_0, 10), (k_1, 11), (k_2, 15)\}.$$



(a) Agent states converge toward the average as the execution proceeds.

(b) The average per-agent distance from the average decreases.

Figure 6.1: An example execution fragment of the distributed average consensus algorithm. As the execution proceeds, agents move closer to the global average, as seen both in their state space (left) and in the overall system error measure.

The objective of the system is for each agent to contain $\mathrm{AM}(S_0, \mathcal{A})$; in this context,

$$\begin{aligned} S^\star &= \{(k_0, \mathrm{AM}(S_0, \{k_0, k_1, k_2\}))\,, (k_1, \mathrm{AM}(S_0, \{k_0, k_1, k_2\}))\,, (k_2, \mathrm{AM}(S_0, \{k_0, k_1, k_2\}))\} \\ &= \{(k_0, 12)\,, (k_1, 12)\,, (k_2, 12)\}\,. \end{aligned}$$

A simple specification might be that when agents interact, they update their state with the average of the group. This not only maintains the global average, but decreases, on average, the distance each agent is from the goal. That distance, or error measure, could be something like a mean square error; we denote such an error function—specifically the mean square error—with the function d.

An example execution fragment that exhibits these properties is outlined in Figure 6.1, where agent states are converging toward $S^\star$. A detailed account of the fragment is

State 0 As previously stated, in the initial state

$$\begin{aligned} S_0 &= \{(k_0, 10), (k_1, 11), (k_2, 15)\}, \\ \mathrm{AM}(S_0, \mathcal{A}) &= 12, \\ d &\approx 4.67. \end{aligned}$$

Agents k_0 and k_2 interact: $\{(k_0, 10), (k_2, 15)\} \longrightarrow \{(k_0, 12.5), (k_2, 12.5)\}$.

State 1 The interaction of results in

$$\begin{aligned} S_1 &= \{(k_0, 12.5), (k_1, 11), (k_2, 12.5)\}, \\ \mathrm{AM}(S_0, \mathcal{A}) &= 12, \\ d &= 0.50. \end{aligned}$$

Agents k_0 and k_1 interact: $\{(k_0, 12.5), (k_1, 11)\} \longrightarrow \{(k_0, 11.75), (k_1, 11.75)\}$.

State 2 The interaction of results in

$$\begin{aligned} S_2 &= \{(k_0, 11.75), (k_1, 11.75), (k_2, 12.5)\}, \\ \mathrm{AM}(S_0, \mathcal{A}) &= 12, \\ d &\approx 0.13. \end{aligned}$$

Agents k_1 and k_2 interact: $\{(k_0, 11.75), (k_1, 12.5)\} \longrightarrow \{(k_0, 12.125), (k_1, 12.125)\}$.

State 3 The interaction of results in

$$\begin{aligned} S_3 &= \{(k_0, 11.75), (k_1, 12.125), (k_2, 12.125)\}, \\ \mathrm{AM}(S_0, \mathcal{A}) &= 12, \\ d &\approx 0.03, \end{aligned}$$

... and so on. □

6.2 System Specification

The system consists of a non-empty finite set of agents. The state of an agent k in S consists of a value from the set $\mathbb{R}$. From Section 2.1, $S(k)$ expresses the real value of agent k in state S. The goal of the system is to reach a state in which all agents, individually, contain the average value of the initial global state:

$$\forall k \in \mathcal{A} \colon S^{\star}(k) = \mathrm{AM}(S_0, \mathcal{A}).$$

The transition function maintains the sum of the set, and does not increase the sum of its squares.

Definition 27 (Average Transition) A system transition involves an agent a group $K \subseteq \mathcal{A}$. It maintains the following relation

$$S \xrightarrow{K} S' \implies \left(\sum_{k \in K} S(k) = \sum_{k \in K} S'(k)\right) \bigwedge \left(\sum_{k \in K} S(k)^2 \geq \sum_{k \in K} S'(k)^2\right).$$

The state of agents not in K are unchanged. □

In this way, system transitions maintain a local-global relation.

Theorem 16 (Average Transitions Are Local-Global) *For any $K \subseteq \mathcal{A}$, the action K is local-global:*

$$S \xrightarrow{K} S' \implies \left(\sum_{k \in K} S(k) = \sum_{k \in K} S'(k) \implies \sum_{k \in K \cup \{j\}} S(k) = \sum_{k \in K \cup \{j\}} S'(k)\right) \bigwedge \tag{6.1}$$

$$\left(\sum_{k \in K} S(k)^2 \geq \sum_{k \in K} S'(k)^2 \implies \sum_{k \in K \cup \{j\}} S(k)^2 \geq \sum_{k \in K \cup \{j\}} S'(k)^2\right) \tag{6.2}$$

where $j \notin K$. □

Proof Equation 6.1 follows from Theorem 5, since $\langle \mathbb{R}, + \rangle$ is a commutative monoid. Equation 6.2 follows for the same reason, and because multiplication is monotonic with respect

to $\geq$. ■

Example 14 (Interact) Consider a set of agents on a line, in one dimension, that start in arbitrary positions, but eventually converge to a single point [17]. Interactions occur between pairwise agents and "move" the agents towards one another, reducing their distance by some amount r, where $r \in [L, 1-L]$ and L is a constant such that $L \in (0, 0.5]$. The constant L is a lower bound on the overall improvement on the two agents. An interaction, denoted $\otimes(j, k, r)$, between agents j and k, follows

$$S'(j) = S(j) + r(S(k) - S(j))$$
$$S'(k) = S(k) - r(S(k) - S(j)).$$

It can be shown, by algebraic manipulation, that this action maintains the transition predicate (Definition 27). □

Fairness

We assume that the system cannot be permanently partitioned into non-communicating subsets. This ensures that eventually an action between agents in any non-empty subsets K and its complement $\bar{K}$, is executed (see Figure 2.2). Formally, for all $J \in 2^{\mathcal{A}}$,

$$F_K = \left\{ J \mid J \cap K \neq \emptyset \wedge J \cap \bar{K} \neq \emptyset \right\}. \tag{6.3}$$

For the remainder of the chapter, we restrict attention to interactions that follow the semantics of $\otimes$ (Example 14). The constant L is a unique solution to the equation $x(1-x) = C$, where $0 < C \leq 0.5$. The fairness condition $\mathcal{F}$ is

$$\mathcal{F} = \left\{ F_J \mid J \in 2^{\mathcal{A}} \setminus \emptyset \right\}.$$

6.3 System Correctness

Conservation

Lemma 9 (AM Conservation) *If all actions of a transition system are average transitions, then the system has the following invariant:*

$$\textbf{\textit{Invariant}}: \qquad \sum_{k\in\mathcal{A}} S_0(k) = \sum_{k\in\mathcal{A}} S(k) \bigwedge \sum_{k\in\mathcal{A}} S_0(k)^2 \geq \sum_{k\in\mathcal{A}} S(k)^2. \qquad \square$$

Proof Follows from Theorem 16 and Corollary 1. ■

Progress

Consider a new metric over system state: *mean square error*.

Definition 28 (Mean Square Error) The mean square error is a function, $\text{MSE}\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, such that

$$\text{MSE}(S, K) = \frac{1}{|K|} \sum_{k\in K} \left(S(k) - \text{AM}(S, K)\right)^2,$$

where $S \in \mathcal{S}$ and $K \subseteq \mathcal{A}$. $\square$

The mean square error plays an important role in our liveness analysis. It provides an approximation on the distance an agent value is from the global average, a good metric for system evolution.

Definition 29 (Progress Variables) The predicates G and Q on the state space of the system are

$$G(S) \equiv \text{AM}(S, \mathcal{A}) = \text{AM}(S_0, \mathcal{A})$$
$$Q(S) \equiv \forall k \in \mathcal{A}\colon S(k) = \text{AM}(S_0, \mathcal{A}).$$

The distance function d is the mean square error of S:

$$d(S) = \mathrm{MSE}(S, \mathcal{A}).$$

□

Theorem 17 *The predicate G is an invariant of the system.* □

Proof From the transition, we can imply that the average is maintained:

$$\begin{aligned} S \xrightarrow{K} S' &\implies \sum_{k \in K} S(k) = \sum_{k \in K} S'(k) \\ &\implies \frac{1}{|K|} \sum_{k \in K} S(k) = \frac{1}{|K|} \sum_{k \in K} S'(k) \\ &\implies \mathrm{AM}(S, K) = \mathrm{AM}(S', K). \end{aligned}$$

Moreover, because $\langle \mathbb{R}, + \rangle$ forms a commutative monoid, average is a local-global relation with respect to equality. Thus, from Corollary 1 and Corollary 3 the theorem holds. ■

Theorem 18

$$\forall k \in \mathcal{A}, K \subseteq \mathcal{A} \colon G(S) \wedge S \xrightarrow{K} S' \implies d(S) \geq d(S')$$

□

Proof From the transition, we can imply that the mean square error does not increase: By assumption, both the sum and the average of the elements in S and S' are equal; thus

$$2 \cdot \mathrm{AM}(S, K) + \mathrm{AM}(S, K)^2 = 2 \cdot \mathrm{AM}(S', K) + \mathrm{AM}(S', K)^2.$$

By algebraic manipulation:

$$\begin{aligned}
\sum_{k\in K} S(k)^2 &\geq \sum_{k\in K} S'(k)^2 \\
\sum_{k\in K} \left(S(k)^2 - 2\cdot \mathrm{AM}(S,K) + \mathrm{AM}(S,K)^2\right) &\geq \sum_{k\in K} \left(S'(k)^2 - 2\cdot \mathrm{AM}(S',K) + \mathrm{AM}(S',K)^2\right) \\
\sum_{k\in K} \left(S(k) - \mathrm{AM}(S,K)\right)^2 &\geq \sum_{k\in K} \left(S'(k) - \mathrm{AM}(S',K)\right)^2 \\
\frac{1}{|K|}\sum_{k\in K} \left(S(k) - \mathrm{AM}(S,K)\right)^2 &\geq \frac{1}{|K|}\sum_{k\in K} \left(S'(k) - \mathrm{AM}(S',K)\right)^2 \\
\mathrm{MSE}(S,K) &\geq \mathrm{MSE}(S',K).
\end{aligned}$$

Since the mean square error is a local-global relation with respect to $\geq$ (See Section A.3), the the theorem holds by Corollary 2:

$$\begin{aligned}
\mathrm{MSE}(S,K) \geq \mathrm{MSE}(S',K) &\implies \mathrm{MSE}(S,\mathcal{A}) \geq \mathrm{MSE}(S',\mathcal{A}) \\
&\implies d(S) \geq d(S').
\end{aligned}$$

■

Theorem 19

$$\exists F_K \in \mathcal{F}, \forall K \in F_K \colon G(S) \wedge \neg Q(S) \wedge S \overset{K}{\rightarrowtail} S' \implies d(S) > d(S')$$

□

Proof From the definition of $Q(S)$ and MSE,

$$\begin{aligned}
\neg Q(S) &\implies \exists k \in \mathcal{A} \colon S(k) \neq \mathrm{AM}(S_0,\mathcal{A}) \\
&\implies \exists k \in \mathcal{A} \colon (S(k) - \mathrm{AM}(S,\mathcal{A}))^2 \geq \mathrm{MSE}(S,\mathcal{A}). \qquad (6.4)
\end{aligned}$$

Without loss of generality, we restrict attention to an instance of k where

$$S(k) - \mathrm{AM}(S_0,\mathcal{A}) < 0.$$

Proof of the other case is symmetric. From this assumption, Equation 6.4 becomes

$$-S(k) + \mathrm{AM}(S, \mathcal{A}) \geq \mathrm{RMSE}(S, \mathcal{A})$$

where $\mathrm{RMSE}(S, \mathcal{A}) = \sqrt{\mathrm{MSE}(S, \mathcal{A})}$.

Let $a_0, a_1, \ldots, a_{n-1}$ denote an ordering over the agents such that $S(a_i) \leq S(a_{i+1})$ and $n = |\mathcal{A}|$. Thus

$$S(a_0) \leq S(k) \ \wedge \ \mathrm{AM}(S, \mathcal{A}) \leq S(a_{n-1}).$$

Further, define Δ such that $\Delta(a_i) = S(a_i) - S(a_{i-1})$, for all $i \in (0, n)$. We have

$$\begin{aligned} \mathrm{RMSE}(S, \mathcal{A}) &\leq S(a_{n-1}) - S(a_0) \\ &\leq \sum_{i=1}^{n-1} \Delta a_i. \end{aligned}$$

Since, for all i, $\Delta(a_i) \geq 0$,

$$\exists i \colon \Delta(a_i) \geq \frac{\mathrm{RMSE}(S, \mathcal{A})}{n-1}. \tag{6.5}$$

We use this i to choose our instance of F_K.

$$F_K = \{(j, k) \ \mid S(j) \geq S(a_i) \wedge S(a_i) > S(k)\}.$$

It is left to show that interactions in F_K strictly decrease the variant function. Recall that we are restricting attention to pairwise agent interactions that follow interaction semantics as defined in Example 14—when this action is executed,

$$\begin{aligned} S'(j) &= S(j) + r(S(k) - S(j)) \\ S'(k) &= S(k) - r(S(k) - S(j)). \end{aligned}$$

From Equation 6.5 we can conclude

$$S(j) - S(k) \geq \frac{\text{RMSE}(S, \mathcal{A})}{n-1}.$$

Hence,

$$\begin{aligned} S'(j)^2 + S'(k)^2 &= S(j)^2 + S(k)^2 - 2r(1-r)(S(j) - S(k))^2 \\ &\leq S(j)^2 + S(k)^2 - 2r(1-r) \cdot \frac{\text{MSE}(S, \mathcal{A})}{(n-1)^2}. \end{aligned}$$

The mean square error in the new state follows

$$\begin{aligned} \text{MSE}(S', \mathcal{A}) &\leq \text{MSE}(S, \mathcal{A}) - 2r(1-r) \cdot \frac{\text{MSE}(S, \mathcal{A})}{n(n-1)^2} \\ &= \text{MSE}(S, \mathcal{A}) \cdot \left(1 - \frac{2r(1-r)}{n(n-1)^2}\right). \end{aligned}$$

Since $L \leq r \leq 1 - L$, and $r(1-r) \geq L(1-L)$,

$$\text{MSE}(S', \mathcal{A}) \leq \text{MSE}(S, \mathcal{A}) \cdot \underbrace{\left(1 - \frac{2L(1-L)}{n(n-1)^2}\right)}_{\alpha}.$$

By definition $0 \leq \alpha < 1$. Thus, an interaction strictly decreases the mean square error. ■

Theorem 20 *As an execution tends to infinity, the sequence $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^n$ is decreasing, converging to zero.* □

Proof Follows directly from Theorem 19. ■

Chapter 7

External Inputs

In the previous chapter we described a distributed system that computed the average of *initial* values of agents using an algorithm in which a subset K of agents participating in an action set their values to the average of the values of agents in K. In the previous chapter, the desired final state, $S^\star$ was a function of the initial state S_0;

$$S^\star(k) = f(S_0).$$

An example of the system described in the previous chapter is a system that computes the average of sensor values, such as temperature, when these values do not change as computation proceeds. In practice, however, sensor values may change with time. This problem is an instance of the dynamic consensus problem. It is especially relevant to the control systems community to study problems such as mobile tracking [73, 74].

In this chapter we consider the a generalization of the algorithm examined in the previous chapter applied to dynamic systems in which values to be averaged *change* as computation proceeds. Consider a system in which each agent k has a value $C[k]$ where $C[k]$ is changed by an external environment to take on values $C_0[k], C_1[k], C_2[k], \ldots$, where the value at time t is $C_t[k]$. Let $\hat{S}_t$ be the desired global state at time t; then in this chapter $\hat{S}_t$ is a function of C_t

$$\forall t\colon \hat{S}_t = f(C_t).$$

Note that in the previous chapter the goal state did not change with t and was only a function

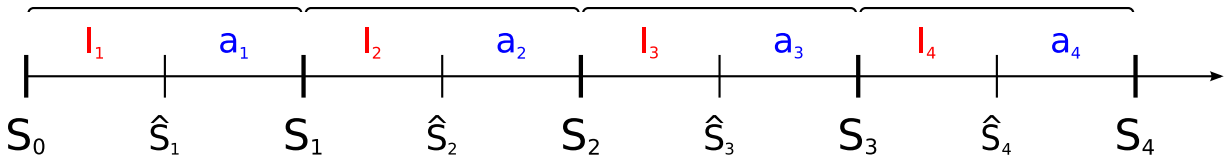


Figure 7.1: An execution fragment denoting the separation between external inputs and system actions.

of the initial state.

In the previous chapter, the error at a point t in the computation was defined to be:

$$d(S_t) = \| S_t - f(S_0) \| .$$

We showed that $d(S_t)$ does not increase with t, and if the error is positive then it decreases eventually. In this chapter, the error at a point t in the computation is defined to be:

$$d(S_t) = \| S_t - f(C_t) \| .$$

In this chapter $d(S_t)$ may increase with t because the environment may change C_t. Here we explore bounds on the error.

In the previous chapter we relied on a weak form of fairness: we required only that for every non-empty subset of agents, an action that includes an agent in the subset and an agent not in the subset is executed *eventually*. In this chapter, since the environment is changing values at every time step, such a weak form of fairness is insufficient to obtain error bounds. Now we need some notion of time because the environment changes values continuously and the system must react in some bounded time to these changes if errors are not to get arbitrarily large. The assumption about fairness in this chapter is much stronger than the assumption used in the previous chapter because we replace "eventually" with "within a given time τ." We now require that at each point t in time, for every non-empty subset of agents, an action that includes an agent in the subset and an agent not in the subset is executed within the next τ units.

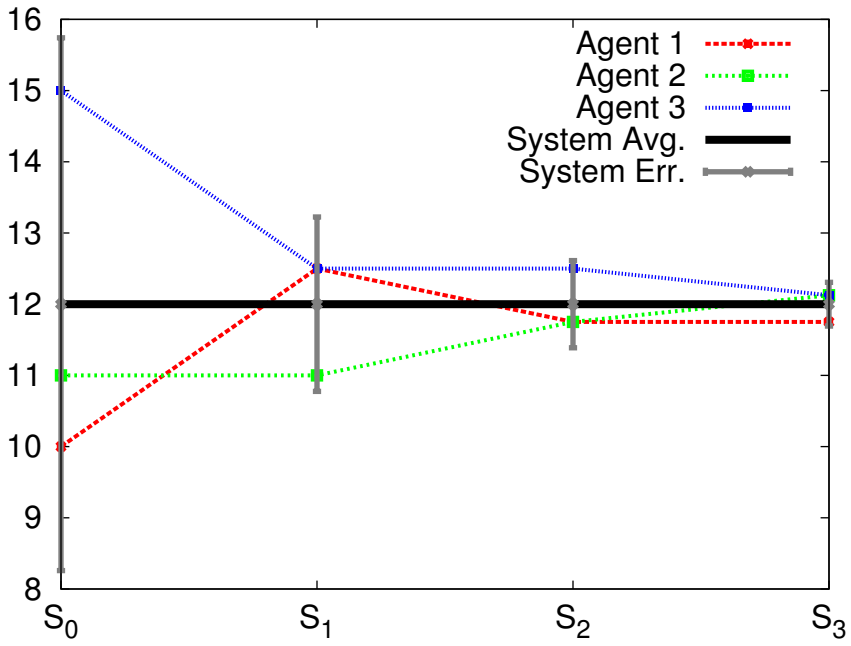


Figure 7.2: An execution fragment with no external input. The error of the system, defined by the two-norm, is denoted as error bars along the average curve, which is constant. As agents update their states, overall system error decreases.

Examples

Example 15 (Without External Input) Recall Example 13 from Chapter 6—given a set of 3 agents, the example detailed three system transitions which maintained the global average by maintaining the local average. We assume the same execution fragment herein, however the error function of interest is the two-norm, rather than the mean square error. Figure 7.2 is a reprint of Figure 6.1(a), with the two-norm overlayed as error bars. Note that error is decreasing as the execution proceeds. □

Example 16 (With External Input) Now consider an example of a system with external input. The system trajectory consists of discrete steps. Each step t consists of (1) the environment increments C_t by an amount I_t, and (2) the agents execute an action a^t. The following example shows a possible system trajectory.

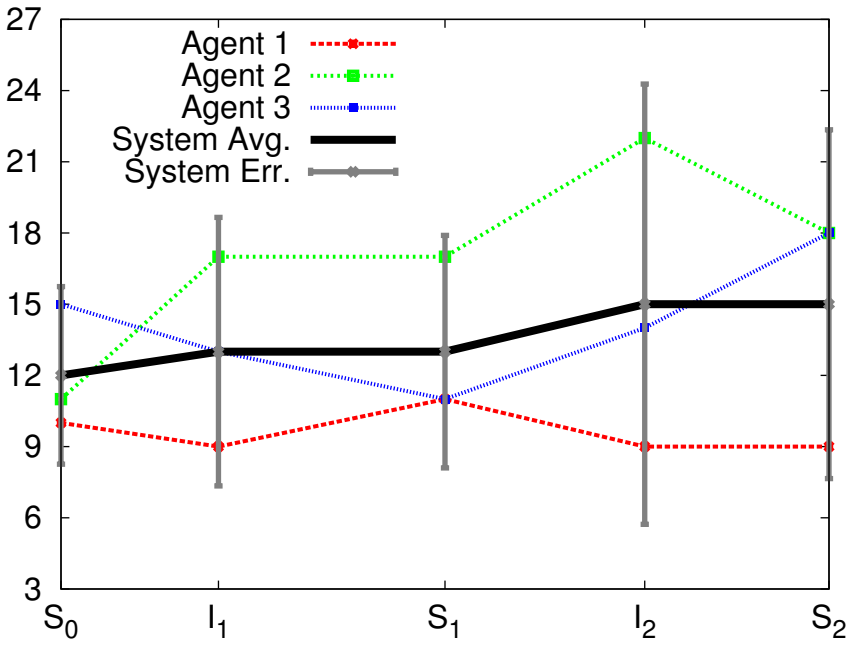


Figure 7.3: System dynamics with external inputs. Agents continue to converge toward the average between system transitions, but are interrupted by arbitrary alterations to their state. Our objective in analysing systems like these is to understand how the external input impacts the global error of the system.

State 0 The initial state of the system is

$$\begin{aligned} S_0 &= [10, 11, 15], \\ \hat{S}_0 &= \mathrm{AM}(S_0, \{0, 1, 2\}) = [12, 12, 12], \\ d(S_0) &= \| \; S_0 - \mathrm{AM}(S_0, \{0, 1, 2\}) \; \| \; . \end{aligned}$$

1. External Increment: The external environment adds increment $I_1 = [-1, 6, -2]$, giving the following value of C,

$$\begin{aligned} C_1 &= S_0 + I_1 \\ &= [10, 11, 15] + [-1, 6, -2] \\ &= [9, 17, 13]. \end{aligned}$$

The state of the system becomes

$$\begin{aligned}\hat{S}_1 &= S_0 + I_1 \\ &= [10, 11, 15] + [-1, 6, -2] \\ &= [9, 17, 13].\end{aligned}$$

2. Action: Agents 0 and 2 interact: $[9, 13] \longrightarrow [11, 11]$.

State 1 The interaction results in

$$\begin{aligned}S_1 &= [11, 17, 11], \\ \hat{S}_1 &= \mathrm{AM}(S_0, \{0, 1, 2\}) = [13, 13, 13], \\ d(S_1) &= \| \ S_1 - \mathrm{AM}(S_1, \{0, 1, 2\}) \ \| \ .\end{aligned}$$

1. External Increment: The external environment adds increment $I_1 = [-2, 5, 3]$, giving the following value of C,

$$\begin{aligned}C_2 &= C_1 + I_2 \\ &= [9, 17, 13] + [-2, 5, 3] \\ &= [7, 22, 16].\end{aligned}$$

The state of the system becomes

$$\begin{aligned}\hat{S}^2 &= S_1 + I_2 \\ &= [11, 17, 11] + [-2, 5, 3] \\ &= [9, 22, 14].\end{aligned}$$

2. Action: Agents 1 and 2 interact: $[22, 14] \longrightarrow [18, 18]$.

State 2 The interaction results in

$$\begin{aligned} S_1 &= [9, 18, 18], \\ \hat{S}_2 &= \mathrm{AM}(S_0, \{0, 1, 2\}) = [15, 15, 15], \\ d(S_1) &= \| \; S_1 - \mathrm{AM}(S_1, \{0, 1, 2\}) \; \| \; . \end{aligned}$$

. . . and so on. □

7.1 Model

Let C_t be an N-vector where N is the number of agents, and $C_t[j]$ is the element of the vector corresponding to the j-th agent. The goal state $\hat{S}_t$ is a vector whose elements are the average of the values of C_t; therefore:

$$\hat{S}_t = B \cdot C_t$$

where B is an $N \times N$ array, all of whose elements are $1/N$. Since C_t is specified as a sequence of incremental changes at each instant in time:

$$C_t = S_0 + I_1 + I_2 + \ldots + I_t$$

where S_0 and I_k, $1 \leq k \leq t$, are N-vectors. For convenience in notation, and without loss of generality, we assume that S_0 is the zero vector, 0. Thus,

$$\forall t > 0 : C_t = \sum_{j=1}^{t} I_j.$$

We consider a system trajectory with discrete steps in which step t consists of two parts: (i) the environment increments C_{t-1} by I_t, and (ii) a subset of agents executes an action a^t. An action a^t by a subset K of agents can be represented by an $N \times N$ array A_t with the

following properties. Let $|K|$ denote the cardinality of K. Then

$$\forall i, j \in K \colon A_t[i,j] = \frac{1}{|K|} \qquad \bigwedge$$
$$\forall i, j \notin K \colon (i \neq j \implies A_t[i,j] = 0) \wedge (A_t[i,i] = 1)\,.$$

In the t-th step, when the environment increments the state by I_t, the state becomes $S_{t-1}+I_t$. When agents execute an action a^t the state of the system transits from $\hat{S}_t$ to S_t, where $\hat{S}_t = S_{t-1} + I_t$ and to $S_t = A_t \cdot \hat{S}^t$. Then A is $A_t \cdot (S_{t-1} + I_t)$. Therefore,

$$S_t = A_t \cdot (S_{t-1} + I_t).$$

7.2 Theory

The goal state $\hat{S}_t$ is the vector in which elements are the average of C_t, and therefore

$$\begin{aligned}\hat{S}_t &= B \cdot C_t \\ &= B \cdot \sum_{j=1}^{t} I_j \\ &= \sum_{j=1}^{t} B \cdot I_j.\end{aligned}$$

Let $p(u, v)$ for $v \leq u$ be defined as:

$$p(u, v) = A_u \cdot A_{u-1} \cdot \ldots \cdot A_v.$$

Theorem 21

$$\forall t > 0 \colon S_t = \sum_{j=1}^{j=t} p(t, j) \cdot I_j$$

□

PROOF The proof follows by induction on j.

Base Case

$$\begin{aligned} S_1 &= \sum_{j=1}^{1} A_1 I_1 \\ &= A_1 \cdot (S_0 + I_1) \\ &= A_1 I_1. \end{aligned} \tag{7.1}$$

Equation 7.1 holds since S_0 is the identity.

Inductive Step Assume that the theorem holds for all $t \in [1, \ldots, T-1]$ and we prove the theorem for $t = T$. We have shown

$$S_T = A_T \cdot (S_{T-1} + I_T).$$

Substitute for S_{T-1} using the induction assumption:

$$S_{T-1} = \sum_{j=1}^{T-1} p(T-1, j) \cdot I_j.$$

Therefore,

$$\begin{aligned} S_T &= A_T \left(\sum_{j=1}^{T-1} p(T-1, j) \cdot I_j + I_t \right) \\ &= \sum_{j=1}^{T-1} p(T, j) \cdot I_j + A_t I_T \\ &= \sum_{j=1}^{T} p(T, j) \cdot I_j. \end{aligned}$$

■

Theorem 22

$$\forall t > 0 \colon d(S_t) \leq \sum_{j=1}^{t} \parallel p(t,j) \cdot I_j - B \cdot I_j \parallel$$

□

PROOF By definition

$$\begin{aligned} d(S_t) =& \parallel S_t - \hat{S}_t \parallel \\ =& \parallel \sum_{j=1}^{t} p(t,j) \cdot I_j - B \cdot I_j \parallel . \end{aligned}$$

The result follows from the triangle-inequality property of norms. ■

Theorem 23 *Let $g(t) = \lfloor t/\tau \rfloor$; thus $g(t)$ is the number of consecutive intervals of length T in t. Then,*

$$\forall t > 0 \colon d(S_t) \leq \sum_{j=1}^{t} \parallel I_j \parallel \cdot \alpha^{g(t-j)}$$

where $\alpha \in [0, 1)$. □

PROOF From Theorem 20, for any nonnegative integer k:

$$\parallel p(j + k\tau, j) \cdot I_j - B \cdot I_j \parallel \leq \alpha^k \cdot \parallel I_j \parallel .$$

The result follows. ■

Corollary 4 *Assume that the norms of the increments I_j, $1 \leq j \leq t$, are bounded by a value Δ. Then*

$$d(S_t) \leq \Delta\tau \cdot \frac{1 - \alpha^{g(t)+1}}{1 - \alpha}.$$

□

Chapter 8

Framework Extensions

8.1 Error Bounds with Changing Inputs

In this section we consider systems with changing inputs. Specifically, these are problems in which, at time t, each agent j receives an input value $y_t[j]$. An agent's input changes in a manner that is described later. Let H_t be the multiset of values $y_t[j]$ for all agents j. At each instant t, each agent computes an estimate of $f(H_t)$ where f is a given function from multisets of agent values to some type. Let the estimate computed by agent j at time t be $z_t[j]$. Define the error of agent j's estimate at time t to be $d_t[j]$ where:

$$d_t[j] = \parallel f(H_t) - z_t[j] \parallel .$$

Define the system error D_t at time t to be the maximum error at time t over all agents:

$$D_t = \max_j d_t[j].$$

We wish to determine a bound on D_t for all t.

Since the environment controls agent interactions, the error bound can become arbitrarily large. For example, consider the case where $f(H)$ is the minimum value in the multiset H, and where the system has two agents indexed 0 and 1. Assume that for $t > 0$, $y_t[0] = 0$ and

$y_t[1] = t$. Then

$$f(H_t) = \min\{y_t[0], y_t[1]\} = 0.$$

Suppose agent 1 never interacts with agent 0 in the interval $[0, T]$, for some T. Then for all t in the interval $[0, T]$ the estimate $z_t[1]$ for agent 1 is $y_t[1]$, and thus $z_t[1] = t$; therefore $D_t = t$. Thus, by making T arbitrarily large the error can be made arbitrarily large. We can bound the error by strengthening the fairness constraints by introducing time.

We use the concept of epochs introduced in chapter 7 to obtain a bound on the error D_t. An epoch is an interval of time during which agent interactions satisfy the following fairness constraint. If agent input values $y_t[k]$ are constant for all agents k, then within an epoch the value $x_t[j]$ of agent j, for all j, is the minimum value. In other words, an epoch is long enough to ensure that all agents obtain the desired value provided inputs do not change. Now consider the case where agent input values $y_t[j]$ can change, but the maximum rate of change, either increase or decrease, is Δ per unit time. Then, in an epoch of length T, the maximum change in the input value of any agent is $\Delta \cdot T$. It follows that the system error D_t at the end of an epoch is at most $\Delta \cdot T$. This idea can be used to be bound the system error at each time instant by having repeated epochs.

8.2 Termination Detection

The multi-agent systems considered in this thesis operate in unreliable or hostile environments. Algorithm designers do not know which agents will interact in a given computation. Termination detection algorithms in the literature usually assume that agents and communication channels are represented by static graphs. For example, termination detection of diffusing computations creates an overlay tree on a static graph. Since the graph in our case changes in a manner determined by the environment, a termination detection algorithms cannot create a tree overlaid on the underlying agent-interaction graph because the graph may change. The traditional termination detection algorithms cannot be used for the systems we consider. Moreover, for some cases we prove that the algorithm converges, because

the algorithm does *not* terminate. For example, the algorithm for computing the average does not terminate at the final solution but gets arbitrarily close to it.

Next we discuss a termination detection algorithm for the case where the ids of all agents are known. Note that agent ids were not given in the algorithms discussed earlier in the thesis. The idea is to start with a straightforward algorithm and then optimize it. The straightforward algorithm is that each agent keeps track of all the agents with which it has interacted directly or indirectly. We describe the algorithm in the context of computing the minimum.

First consider the case of computing the minimum of $y[j]$ over all j where $y[j]$ is a constant for all j. Each agent j keeps a local value $x[j]$ and a set $b[j]$ satisfying the invariant:

$$x[j] = \min_{k \in b[j]} y[k].$$

Initially, $b[j] = \{j\}$, and $x[j] = y[j]$. The messages agent j sends are pairs $(x[j], b[j])$. When an agent j receives a message (u, v) it sets:

$$x[j] := \min(x[j], u)$$
$$b[j] := b[j] \cup v.$$

Agent j has the final value when $b[j]$ is the set of all agents. The computation terminates when each agent determines that all agents have obtained their final values.

The same idea can be used for other problems. The core idea is for each agent j to gather the values $x[k]$ for every agent k and thus obtain the multiset H, and then compute $f(H)$. This is a brute-force solution because each agent merely obtains the states of all other agents and then carries out a local computation. In effect, each agent carries out a centralized algorithm. An alternative solution is for the agents to elect a leader which executes the centralized algorithm and then disseminates the result. The thesis explores multiagent systems that operate in hostile environments, and in such cases the goal is not necessarily to detect termination but to reach a state close to the desired state.

8.3 Limits of the Local-Global Framework

This section explores the limitations of the local-global framework. We consider algorithms that do not fit the framework, and show how to make them fit the framework, though doing so requires more computational time and more data to be exchanged between agents.

8.3.1 Framework Violations

Recall that for a function to exhibit a local-global relation, the following implication must hold (Theorem 1):

$$\forall K \subseteq \mathcal{A}, j \notin K\colon \left(S|_K \trianglerighteq S'|_K \bigwedge S(j) = S'(j)\right) \implies S|_{K\cup\{j\}} \trianglerighteq S'|_{K\cup\{j\}}.$$

Next we address the issue of what can be done to develop a distributed algorithm if the condition does not hold. Consider the problem at the beginning of this chapter. Each agent j has an input value $y[j]$. Consider the case where $y[j]$ is not time varying. Let H be the multiset of values $y[j]$ for all agents j. The problem is for all agents to reach a consensus $f(H)$ where f is a given function from multisets of y-values to some type R. The essential idea of a generic way of satisfying the local-global relation is given in the termination detection section of this chapter. The state of each agent j includes a local value $x[j]$ of type R and a set $b[j]$ of pairs $(y[k], k)$ where $h[j]$ is the multiset of y-values in $b[j]$ and

$$x[j] = f(h[j]).$$

For each j, the cardinality of set $b[j]$ never decreases, and eventually increases until it becomes N, the total number of agents. Initially

$$b[j] = \{(y[j], j)\}.$$

The message m that agent j sends at any point in the computation is $b[j]$. When an agent j receives a message containing a set of pairs $(y[k], k)$ it carries out the following step:

$$b[j] := b[j] \cup \{(y[k], k)\}$$
$$h[j] := h[j] \cup \{y[k]\}$$
$$x[j] = f(h[j]),$$

if $(y[k], k) \notin b[j]$. Agent j has the final value when the cardinality of $b[j]$ is the total number of agents, in which case $h[j] = H$, and therefore $x[j] = f(H)$. The computation terminates when each agent determines that all agents have obtained their final values.

While this approach satisfies local-global it is inefficient. Each agent sends the y-values of all agents that it has until every agent has the y-values of all agents in the system, and then they compute the desired result. Properties of f such as idempotence, can be exploited to obtain efficient algorithms. Next, we consider an example that shows how these properties can be exploited.

8.3.2 k-th Smallest Value

These problems are consensus problems in which the objective is to agree on the k-th smallest value in a set, where k is neither the minimum nor the maximum element.[1] Calculating the second smallest element, $k = 2$ for example, is an instance of this problem. Recall from Section 1.3.1 that the function to compute the second smallest value of a set is denoted $\min_2$; thus, $\min_2(\{1, 2, 3\}) = 2$. The function was shown to not exhibit a local-global relation, as the following example reiterates:

$$\min_2(\{2, 3, 4\}) = \min_2(\{1, 3, 6\}) \wedge 0 = 0 \implies \min_2(\{2, 3, 4\} \cup \{0\}) = \min_2(\{1, 3, 6\} \cup \{0\})$$
$$3 = 3 \wedge 0 = 0 \implies 2 = 1,$$

which is false.

[1] This corresponds to $k = 1$ or $k = N$ respectively, where N is the number of agents in the system.

Consider the following scenario. A system has 3 agents with ids 1, 2, 3, and $y[j] = j$. In this case, $H = \{1, 2, 3\}$ and $f(H)$ is the second-smallest element in the multiset: $f(H) = 2$. Initially $x[j] = y[j] = j$. Each agent j sends its $x[j]$ to agents that can receive the message. When an agent j receives a message m it sets $x[j]$ to the second smallest of the current value of $x[j]$ and m. Thus if agent 2 receives a message with value 3 from agent 3, then agent 2 sets $x[2]$ to the second smallest of 2 and 3, which is 3. If agent 2 then sends $x[2]$ to agent 1, then agent 1 sets $x[1]$ to the second smallest of 1 and 3, which is 3. Thus all agents have the x-value of 3. Then the consensus second-smallest value will be 3, which is an error.

Consider the same scenario with the generic algorithm. If the first message is from agent 3 to agent 2, then the message is $(y[3], 3)$ which is $(3, 3)$ and then agent 2 sets:

$$\begin{aligned} b[2] &:= \{(2, 2)\} \cup \{(3, 3)\} \\ h[2] &:= \{2, 3\} \\ x[2] &:= f(\{2, 3\}) = 3. \end{aligned}$$

Then if the next message is from agent 2 to agent 1, then the content of the message is $\{(2, 2), (3, 3)\}$, and agent 1 sets:

$$\begin{aligned} b[1] &:= \{(1, 1)\} \cup \{(2, 2), (3, 3)\} \\ h[1] &:= \{1, 2, 3\} \\ x[1] &:= f(\{1, 2, 3\}) = 2. \end{aligned}$$

Eventually, for all j:

$$\begin{aligned} b[j] &:= \{(1, 1), (2, 2), (3, 3)\} \\ h[j] &:= \{1, 2, 3\} \\ x[j] &:= f(\{1, 2, 3\}) = 2. \end{aligned}$$

We now make the algorithm more efficient. Agent j does not need to keep track of the entire

set $b[j]$; all it needs to keep track of is the *two* smallest values it has seen so far.

The problem can be converted into a local-global relation by saving information. Let the state of an agent is an ordered pair, (i, j), where $i, j \in \mathcal{T}$. Consider a set of agents K in state S; $\{S|_K\}_1$ denotes the set of first-elements in S restricted to the agents K while $S(k)_1$ denotes the first element of agent k's state. To calculate the second smallest value, as well as maintain that value throughout an execution, agents update their state as follows:

$$\forall k \in K : S'(k) = \{(i, j) \mid i = \min(\{S|_K\}_1) \ \wedge \ j = \min_2(\{S|_K\}_2)\}.$$

Initially both members of an agents ordered pair are equal.

8.4 Bounds on Information Exchange

Let *information exchange* be the number of messages sent within the system in order to solve the given problem. Depending on the "hostility" of the environment the amount of information exchanged within the system can vary. An ideal, low-hostility, environment is one in which every agent can interact with every other agent at all times and without error. In this setting one of the agents—such as the agent with the smallest id—is designated the leader. All agents subsequently send their information to this central leader, who computes the problem solution and responds. As was mentioned in Section 2.2, this centralized approach solves a superset of problems solvable by the local-global framework alone. Considering such an approach, however, is useful in studying the information exchange for local-global problems. In a high-hostility environment designers know nothing about agent interactions other than some weak fairness constraint; for example, that the network will not be partitioned permanently into non-communicating subsets. In this case, agents collectively solve the problem in an independent, localized fashion.

An advantage of the centralised approach is that the amount of information exchanged is minimal. Each agent sends its value to the controller, and the controller sends back the calculated value; thus, if there are n agents in the system, the number of messages sent

is $2n$. The number of messages has to be of the order of the number of agents. For algorithms operating in a hostile environment, agents exchange information opportunistically. This can lead to much more information exchanged than in the benign environment in which all agents can communicate with a leader agent. In the worst case the amount of data exchanged can be unbounded.

Computing the Average The algorithm in which each agent in a group of interacting agents sets its value to the average of the values of agents in the group, converges to the correct value—the average of all the agents in the system—under very weak fairness criteria. However, this algorithm may not terminate, and thus the number of messages exchanged can grow without bound. In the benign centralized solution, each agent sends its value to a leader which computes the average and sends the result back. Thus the ratio of the algorithm in the hostile environment to the algorithm in the benign environment can get arbitrarily large.

Computing the Minimum In the hostile environment, when an agent sends a message it does not know which agents, if any, will receive the message. Therefore, it sends the message repeatedly until it receives an acknowledgment. Thus, the hostile environment can cause the number of messages sent to be arbitrarily large.

Designers trade off robustness and performance. If we want an algorithm to operate correctly in extremely hostile environments then the algorithm can be much less efficient than an algorithm operating in a benign environment with perfect connectivity.

Chapter 9

Tools of Formal Methods

This chapter develops reliable distributed software using the principles of local-global relations. In this way, the chapter presents a practical side of the local-global theory presented thus far. Ultimately we implement many of the algorithms already considered, but do so starting from the theory. To this end, we examine the application of local-global relations to the tools of formal methods—theorem proving, model checking, and contract specification, in particular. As a result, our implementations reap the benefits of the theory: modularity and correctness.

We first consider a representation of local-global relations in a theorem prover. This not only provides a mechanical verification of the theory developed to this point, but creates a library of proofs around distributed systems. Our library is more than a collection of theories, however, it is a reusable tool for software developers. Thus, part of the goal in presenting the library is to provide an instruction manual on how to use it.

With respect to model checking, we examine the impact that local-global relations have on state-based verification. Using statistics produced by the model checker, we are able to compare various implementations of the same system; namely, models written with local-global relations in mind, and models written without. Although such metrics are interesting, the process that this section outlines is also beneficial.

We are able to specify implementation-level, meta-correctness through specification languages. This is code that lives alongside a traditional programming language, but is used to specify and ensure correctness both at compile- and run-time. We find that this type of tool

provides a nice mapping between the theory and the implementation.

The chapter concludes by outlining implementations of the system in different languages. The programs developed are not only based on the algorithms studied in previous chapters, but are a product of our local-global-based formal method process.

9.1 Theorem Prover

Our theory was also verified mechanically using PVS [1]. This process not only allowed us to gain confidence in our methodology, but to build a reusable library of theorems for proving correctness of concurrent systems. We were concerned with representing our theory mechanically, not necessarily automating its proofs. One of the benefits to this approach is that our library is prover agnostic—implementing it in other theorem provers is more an exercise in syntax rather than proof strategy.

Verifying theorems mechanically takes more work than proofs checked by hand, as nothing can be assumed and small steps must be verified. For example, that gcd is commutative requires lemmas about the definition of gcd, which includes properties about max and divisibility. A theorem prover that supports higher-order logic, as PVS does, has the added obligation of showing type-correctness as well. One way to alleviate some of this burden is to develop libraries of theorems. Most prover libraries deal with mathematics in general, defining properties of sets, functions, and numbers. Our contribution is a library that not only extends such traditional ones, but deals with concurrent composition and distributed systems.

The organization of our library closely follows the organization of previous chapters: we first consider local-global relations, then operations and system transitions that are local-global. One of the advantages to using PVS for such a task is its support for theorem modularity through inheritance. Inheritance in PVS is achieved through assumptions and parametrized theories [75, 76], which is a means of defining abstract properties in one library and making those properties concrete in another. Specifically, *assumptions* are lemmas assumed to be true within a theory. These lemmas must be discharged once the theory is

```
1  local_global[
2    A: TYPE,                                          % agents
3    T: TYPE,                                          % type
4    f: FUNCTION[[[A -> T], finite_set[A]] -> T],      % fold
5    >: (transitive?[T])                               % relation
6  ]: THEORY BEGIN
7    ASSUMING
8      pre, post: VAR [A -> T]
9      K: VAR nonempty_set[A]
10
11     lg_base: ASSUMPTION
12       FORALL (j: A | NOT member(j, K)):
13         relates?(pre, post, K) AND pre(j) = post(j) IMPLIES
14           relates?(pre, post, add(j, K))
15   ENDASSUMING
16
17   lg_relation: LEMMA
18     relates?(pre, post, K) AND
19     unchanged?(pre, post, complement(K)) IMPLIES
20       relates?(pre, post, nonempty_fullset)
21 END local_global
```

Figure 9.1: Local-global theory represented in PVS.

imported elsewhere. For example, to prove properties of fold we assume that the operator is commutative, associative, and monotonic—PVS assumptions are a way to force those that inherit from fold to actually discharge these assumptions.

9.1.1 Local-Global

Local-global relations formed the basis of our theory on distributed systems and do the same in our PVS library. Their PVS representation is outlined in Figure 9.1. The theory is parametrized (lines 2–5) with elements of our distributed model: `A` and `T` are the agents and their value type, respectively; the function `f` is a mapping from sets of agents to a value type; and `>` is a transitive relation. Transitivity is enforced by the PVS prelude-defined predicate `transitive?`. The predicates `relates?` and `unchanged?` represent the $\trianglerighteq$ and equality components in the definition of local-global relations (Definition 3):

```
relates?(pre, post: FUNCTION[A->T], K: nonempty_set[A]): bool =
  f(pre, K) > f(post, K)

unchanged?(pre, post: FUNCTION[A->T], K: set[A]): bool =
  FORALL (j: A): member(j, K) IMPLIES pre(j) = post(j)
```

The `local_global` library *assumes* that the parametrized function `f` is actually local-global (lines 7–15). The local-global assumption, `lg_base`, is actually a PVS representation of the assumptions in Theorem 1. Just as we did in Section 2.2, we use this construct to aid in proving local-global relations over entire state-spaces, where `lg_relation` is a PVS representation of Theorem 1.

9.1.2 Fold

The fold theory defines monoid composition, as described in Equation 3.2, and provides lemmas used to prove that it is local-global. Its representation in PVS is outlined in Figure 9.2. As was the case in the `local_global` theory, `fold` is parametrized with agents and their type, `A` and `T`, respectively, along with a transitive binary relation, `<`. Whereas `local_global` expected a function over states, `fold` expects a monoid. The monoid is expressed in its parts using a binary operator over `T`, denoted $\circ$, and an identity element `zero`. The assumptions in this package represent those made in our theory: that $\circ$ and `T` form a commutative monoid that is monotonic with respect to `<`. Again, these assumptions are taken as fact within `fold`, and must be discharged by users of fold.

The fold function, lines 20–24, uses the theorem supplied operator, $\circ$, to compose agent values. The predicate `empty?`, along with functions `rest`, and `choose`, are predefined in the PVS prelude. Measure functions (line 24) are unique to recursive definitions. They are required to provably decrease with each call to the function. In this way, PVS can ensure that the recursive function is total, and will eventually terminate.

The final action of the fold theory is to import the local-global theory (line 26). In general, importing brings lemmas and function definitions into a theories namespace. It also, however, presents importers with lemmas to discharge in the form of type-correctness claims (TCCs). In this case, the TCCs generated by importing `local_global` require that

```
fold[
  A: TYPE,                  % agents
  T: TYPE,                  % monoid type
  o: FUNCTION[T, T -> T], % monoid operator
  zero: T,                  % identity element
  >: (transitive?[T])       % transitive relation
]: THEORY BEGIN
  ASSUMING
    u, v, w: VAR T
    SS: VAR set[T]

    zero_identity: ASSUMPTION u o zero = u AND zero o u = u
    o_commutative: ASSUMPTION u o v = v o u
    o_associative: ASSUMPTION (u o v) o w = u o (v o w)
    o_monotonic:   ASSUMPTION u > v IMPLIES u o w > v o w
    o_closed:      ASSUMPTION member(u, SS) AND member(v, SS)
                              IMPLIES member(u o v, SS)
  ENDASSUMING

  fold(S: [A -> T], K: finite_set[A]): RECURSIVE T =
    IF empty?(K) THEN zero
    ELSE fold(S, rest(K)) o S(choose(K))
    ENDIF
  MEASURE card(K)

  IMPORTING local_global[A, T, fold, >]
END fold
```

Figure 9.2: Monoid composition in PVS.

fold and > are local-global:

```
IMP_local_global_TCC1: OBLIGATION
  FORALL (K: nonempty_set[A], post, pre: state[A, T, fold, >],
          j: A | NOT member[A](j, K)):
    relates?[A, T, fold, >](pre, post, K) AND
    pre(j) = post(j) IMPLIES
      relates?[A, T, fold, >](pre, post, add[A](j, K));
```

Recall that this is the assumption made in the local-global theory (Section 9.1.1) instantiated with fold.

```
1  S: VAR [A -> T]
2  K: VAR finite_set[A]
3  j: VAR A
4
5  fold_emptyset: LEMMA
6    fold(S, emptyset) = zero
7
8  fold_singleton: LEMMA
9    fold(S, singleton(j)) = S(j)
10
11 fold_x: LEMMA
12   member(j, K) IMPLIES fold(S, K) = S(j) o fold(S, remove(j, K))
13
14 fold_add: LEMMA
15   fold(S, add(j, K)) = fold(S, K) o
16     IF member(j, K) THEN zero ELSE S(j) ENDIF
```

Figure 9.3: Lemmas used to prove that fold is local-global.

9.1.3 Monoids

We now consider concrete instantiations of the abstract monoid fold relied upon. Recall Chapter 3, where several operators for commutative monoids were considered. The process of adding those monoids to our PVS library required defining the operator, using it to instantiate fold, and discharging the algebraic assumptions imposed by fold. As was the case when `fold` imported `local_global`, these assumptions were presented in the form of TCCs specialized with the instantiating operator.

Figure 9.4 shows several operators implemented in our PVS library. The four presented in the figure were already defined in standard PVS libraries[1]; they are presented here for completeness. In the definitions of min and max, there is a restriction on the return type (lines 2 and 9, respectively) to aid in proving correctness of the implementation. The implementation of gcd avoids division by zero with its restricted parameter type (line 16). Finally, both gcd and lcm take advantage of `divides`, which is defined in NASA's extended libraries.

Consider the implementation of min (lines 1–6). The TCCs generated when it imports

[1]The PVS prelude defines min and max; the extended NASA libraries define gcd and lcm.

```
1  min: THEORY BEGIN
2    min(m, n: real): {p: real | p <= m AND p <= n} =
3      IF m > n THEN n ELSE m ENDIF
4
5    IMPORTING fold[posnat, real, min, posinf, >=]
6  END min
7
8  max: THEORY BEGIN
9    max(m, n: real): {p: real | p >= m AND p >= n} =
10     IF m < n THEN n ELSE m ENDIF
11
12   IMPORTING fold[posnat, real, min, neginf, <=]
13 END max
14
15 gcd: THEORY BEGIN
16   gcd(i:int, j: {jj:int| i = 0 IMPLIES jj /= 0}): posnat =
17     max({k: posnat | divides(k,i) AND divides(k,j)})
18
19   IMPORTING fold[posnat, int, gcd, 0, =]
20 END gcd
21
22 lcm: THEORY BEGIN
23   lcm(m1,m2: posnat): int =
24     min({k: posnat | divides(m1,k) AND divides(m2,k)})
25
26   IMPORTING fold[posnat, int, lcm, 1, =]
27 END lcm
```

Figure 9.4: Monoid implementations in PVS and their corresponding `fold` instantiations.

fold can be found in Figure 9.5; TCCs for other operators were analogous. While PVS provided definitions of the operators, it did not discharge the required algebraic properties—much of that work was our own. In the case of gcd, for example, PVS provided several basic lemmas that were composed to prove the necessary properties. Unfortunately, it did not provide the same infrastructure for lcm, so we were required to reproduce the gcd effort ourselves.

Convex hull, a commutative monoid considered in Section 3.3, was also implemented. What is interesting about convex hull is that it can be described as an abstract operation, independent of an actual implementation. That is, there are several convex hull algorithms

```
1  IMP_fold_TCC1: OBLIGATION % identity
2    FORALL (u: real):
3      min(u, posinf[real]) = u AND min(posinf[real], u) = u;
4
5  IMP_fold_TCC2: OBLIGATION % associativity
6    FORALL (u, v, w: real): min(u, min(v, w)) = min(min(u, v), w);
7
8  IMP_fold_TCC3: OBLIGATION % commutativity
9    FORALL (u, v: real): min(u, v) = min(v, u);
10
11 IMP_fold_TCC4: OBLIGATION % closed
12   FORALL (S: set[real]): FORALL (u, v: real):
13     member(u, S) AND member(v, S) IMPLIES member(min(u, v), S);
14
15 IMP_fold_TCC5: OBLIGATION % monotonic
16   FORALL (u, v, w: real): u >= v IMPLIES min(u, w) >= min(v, w);
```

Figure 9.5: TCCs produced by importing fold. Each correspond to the monoid-related assumptions of the fold library (Figure 9.2, lines 8–18). These TCCs are specialized for min, using the parameters provided to the importing command.

that can be collectively described in a generic fashion: as a function over sets of points that is super idempotent. We take this abstraction into account in our PVS implementation: outlined in Figure 9.6, `hull` defines convex hull generically. The theory `giftwrap`, Figure 9.7, defines the well-known gift wrapping algorithm and imports `hull` using it. Much of the code presented in Figure 9.7 is based on functions defined within our geometric library. Briefly, a `plot` is a finite set of points, where a point is any native type. A `polygon` is a plot with at least 3 points; an important quality as it simplifies the implementation of the gift wrapping algorithm. The function `rightof` returns a point to the "right of" a given point, where orientation is towards the origin. Finally, `min_y` returns the minimum point within a polygon with respect to the set of y-coordinates.

9.1.4 Discussion

Although we were confident in our hand-proofs, checking our work in a theorem prover was a valuable exercise. First, it forced us into a very particular kind of thought process. On occasion, we had to rethink our proof based on an inability to show it in PVS. The result

```
1  hull[T: TYPE,
2       f: FUNCTION[finite_set[T] -> finite_set[T]]
3  ]: THEORY BEGIN
4    ASSUMING
5      super_idempotent: ASSUMPTION
6        FORALL (a, b: finite_set[T]): f(union(a, b)) = f(union(a, f(b)))
7    ENDASSUMING
8
9    cvx_hull(a, b: finite_set[T]): finite_set[T] = f(union(a, b))
10
11   IMPORTING fold[posnat, finite_set[T], cvx_hull, emptyset, =]
12 END hull
```

Figure 9.6: Convex hull.

of such rethought was often a simplification in our approach. Complexity in our model came from simple building blocks, which is the most efficient way of using a theorem prover. The second advantage of working with a theorem prover was that it provided a tangible representation of our verification process. Rather than having several ideas that were loosely related, when implemented in PVS we had to think about the problem in a more structured fashion—exactly how fold fit into the local-global idea had to be well defined. Such structure ultimately improved our methodology and helped to bridge the gap between theory and implementation.

9.1.5 Related Work

Nipkow and Paulson develop a similar library based on fold using Isabelle/HOL [77]. The most significant difference in their implementation is that it does not rely on the axiom of choice. Rather, they relate finite sets to natural numbers and extract elements based on set cardinality. Axiom of choice was used in our implementation because of its native support in PVS. The authors conclude that the two implementations require approximately the same effort to prove properties about. Like us, they extend their library to other associative and commutative operations. However, they are focused on *defining* such operations using fold rather than *applying* them. For example, where K is a well-ordered set, Nipkow and Paulson

```
1  giftwrap: THEORY BEGIN
2    IMPORTING libext@geometry[real] % define plot and polygon
3
4    giftwrap_r(P: plot, Q: (polygon?), p0: (Q)): RECURSIVE plot =
5      LET p1 = rightof(Q, p0) IN
6        IF member(p1, difference(Q, P)) THEN singleton(p0)
7        ELSE add(p0, giftwrap_r(remove(p1, P), Q, p1))
8        ENDIF
9    MEASURE card(P)
10
11   giftwrap(P: plot): plot =
12     IF polygon?(P) THEN giftwrap_r(remove(p, P), P, p)
13       WHERE p = min_y(P)
14     ELSE P
15     ENDIF
16
17   IMPORTING hull[point, giftwrap]
18 END giftwrap
```

Figure 9.7: Convex hull gift wrapping algorithm in PVS.

define the minimum value as

$$\text{fold}(K, \min),$$

where min is the binary operation; in PVS, however, the definition follows:

$$\min(K) = \{k \mid K(k) \wedge (\forall j \colon K(j) \implies k \leq j)\}.$$

Based on our use of fold, we obtained set operations in the spirit of Nipkow and Paulson implicitly.

9.2 Model Checker

Model checking is a formal method that, given a specification, verifies that a given model meets that specification. In this case, that model is an abstraction of a more concrete software implementation. To verify that a given specification holds, model checkers consider various execution paths of a system looking for violations of the specification; in some cases, this

means analyzing the entire state-space of a system. Such thorough consideration is especially useful when verifying distributed applications, where process communication, interleaved executions, and data-sharing can cause the system to deviate from expected behavior. Model checkers perform this verification at a level of abstraction that is closer to actual system implementation than is considered by other tools of formal methods, such as theorem provers. However, practically searching the state space of a program can be challenging, as state-spaces, even for relatively simple models, can be very large; this is commonly referred to as the *state explosion problem*. With such state-spaces, exhaustive consideration of a systems execution is infeasible for modern computer hardware. Work to address this issue includes novel methods of state representation and storage optimization, as well extensions to multi-core environments [78]. However, application of local-global methods to model generation can also help to reduce the state-space, as is examined this chapter. Specifically, we consider distributed consensus, outlined in Chapter 3, and analyze the difference in states required for verification. Although the results and conclusions from this comparison are problem-specific, our process of applying local-global relations to model checking can be applied to other domains.

9.2.1 Overview

Using the SPIN model checker [2], we studied distributed consensus as described in Chapter 3. Specifically, we considered disjunctive consensus, using the monoid $\langle \vee, \{0, 1\} \rangle$. This type of consensus was chosen for the operators native support in SPIN. Three models of the problem were developed:

Global model Checked, on the global state of the system, that consensus was maintained and eventually reached.

Local-global model Checked that a given relation was maintained locally between state transitions. This model was useful for studying the effect that local-global relations have on model checking complexity.

Hybrid model Checked that a given relation was maintained locally between state transitions. It also ensured that consensus was maintained globally. This model verified that our theory was valid in practice.

We were not only interested in how local-global properties could be applied to model checking, but also what it meant for the performance of the model checker overall. In particular, that using local-global relations reduces verification complexity—modeling Boolean consensus is a straightforward means of testing this hypothesis.

All models represented agents as separate processes that communicated via message passing in a unidirectional ring. The number of agents in the system ranged from 1 to 5, depending on the execution. Each communication channel had a buffer size of 1 that remained constant throughout our analysis. The communication channel, `wire` was an array of Boolean values, where each agent in the system had a unique index with which to send and receive messages. Agents used channel assertions, `xr` and `xs`, to optimize SPIN's operation.

9.2.2 Global Model

The global model is outlined in Figure 9.8. Upon receiving a message, an agent updates its state and determines whether or not global consensus has been achieved. Its state is updated by applying the disjunctive monoid to its existing value and the value received from its neighbor (line 24). This is an instance of the more general group transition described in Section 3.2. To perform the global check of consensus (lines 25–36), the value of each agent is compared to the value determined to be consensus during system initialization (lines 13–17). A running tally of this comparison is kept; if any agent in the system does not match initial consensus, global consensus is set to false.

Specifying Fairness

Correctness in the global model is a temporal property: *eventually* all values within the state array will be equal to some constant value. SPIN "never" claims are a mechanism commonly used to express such properties about a system. When combined with program labels, they

```
1  bool consensus, initial, state[AGENTS]
2  chan wire[AGENTS] = [BUFFER] of { bool }
3
4  active [AGENTS] proctype agent() {
5    byte n;
6    bool msg;
7
8    chan out = wire[(_pid + 1) % AGENTS];
9    chan inp = wire[_pid];
10   xs out;
11   xr inp;
12
13   if
14     :: state = true;
15        initial = true
16     :: state = false
17   fi;
18
19   out!state[_pid];
20 end:
21   do
22     :: inp?msg ->
23 rcv:
24        state[_pid] = (state[_pid] || msg);
25        n = 0;
26        consensus = true;
27        do /* Check that consensus is maintained globally */
28          :: n < AGENTS ->
29             if
30               :: (state[n] != initial) ->
31                  consensus = false;
32                  break
33               :: else -> n++
34             fi
35          :: else -> break
36        od;
37        out!state[_pid];
38   od
39 }
```

Figure 9.8: Promela "global model." Consensus is verified globally, across all agent states. The number of agents in the system, AGENTS, was set at compile time.

```
1  never {    /* !(<>[]consensus) && ([]<>a1rcv) && ([]<>a2rcv) */
2  T0_init:
3          if
4          :: (! ((consensus)) && (a1rcv) && (a2rcv)) ->
5             goto accept_S570
6          :: (! ((consensus)) && (a1rcv)) -> goto T2_S570
7          :: (! ((consensus))) -> goto T1_S570
8          :: (1) -> goto T0_init
9          fi;
10 accept_S570:
11         if
12         :: (1) -> goto T0_init
13         fi;
14 T2_S570:
15         if
16         :: ((a2rcv)) -> goto accept_S570
17         :: (1) -> goto T2_S570
18         fi;
19 T1_S570:
20         if
21         :: ((a1rcv) && (a2rcv)) -> goto accept_S570
22         :: ((a1rcv)) -> goto T2_S570
23         :: (1) -> goto T1_S570
24         fi;
25 }
```

Figure 9.9: Spin never claim to ensure fairness and progress amongst two agents.

can also be used to define fairness for that system. We utilize both aspects of Spin never claims to verify correctness of the naive model.

By default Spin provides fairness it at the process-level, ensuring that all processes have a chance to run. This, in conjunction with agents communicating a ring, fulfills one aspect of fairness as defined and assumed throughout this thesis: that agents will never be permanently partitioned. However, we also have an implicit assumption that agents are "active" participants toward the global goal. That is, agents are diligently managing and communicating their internal state. This is realized in our model by annotating agents with a `rcv` label (Figure 9.8, line 23), and informing Spin to only consider executions where control flow goes through `rcv` with some frequency.

Correctness and fairness in the global model were both temporal properties; Spin never

claims are able to recognize and verify such properties by converting LTL statements into automata. The SPIN-accepted LTL statement for two agents corresponding to the aforementioned correctness and fairness requirements was as follows:

```
!(<>[]consensus) && ([]<>(agent[1]@rcv)) && ([]<>(agent[2]@rcv))
```

where `consensus` is the global variable defined in the model (Figure 9.8, line 1). The `agent[n]@rcv` syntax specifies that control flow of the given agent, 1 and 2 in this case, is at the specified label, `rcv`. Thus, the LTL statement instructs SPIN to verify that consensus eventually holds forever in executions where each agent always receives eventually. Again, SPIN can convert such a statement into a Promela-encoded automata that can in turn be inserted into an existing model. For the complete code generated using this LTL statement, see Figure 9.9.

9.2.3 Local-Global Model

The local-global model is outlined in Figure 9.10. Upon receiving a message, an agents calculates its new value by applying the monoid to its existing value and the message received from its neighbor; it subsequently verifies that the local relation is maintained (lines 19–23). The premise behind this optimization is based on the theoretic analysis from Chapter 3, where the general consensus problem was described as a local-global relation. Recall that it was shown when subsets of agents update their states using a commutative monoid, consensus amongst the global set would eventually be reached. Our implementation of the problem in Promela is a subset of this model, as there is a single agent that updates its state during each transition. Let k be the updated agent, and *false* in Boolean algebra be less-than *true*; The local-global relation is as follows:

$$\underbrace{\bigvee_{j\in\{k\}} S(j) \leq \bigvee_{j\in\{k\}} S'(j)}_{\text{local relation}} \bigwedge \underbrace{\forall j \in \mathcal{A}\setminus\{k\} : S(j) = S(j)}_{\text{unchanged}} \implies \underbrace{\bigvee_{j\in\mathcal{A}} S(j) \leq \bigvee_{j\in\mathcal{A}} S'(j)}_{\text{global relation}}, \quad (9.1)$$

```
1  chan wire[AGENTS] = [BUFFER] of { bool }
2
3  active [AGENTS] proctype agent() {
4    bool msg, state, post;
5
6    chan out = wire[(_pid + 1) % AGENTS];
7    chan inp = wire[_pid];
8    xs out;
9    xr inp;
10
11   if
12    :: state = true
13    :: state = false
14   fi;
15
16   out!state;
17 end:
18   do
19     :: inp?msg ->
20        post = (state || msg);
21        assert(state <= post); /* Verify the local relation */
22        state = post;
23        out!state
24   od;
25 }
```

Figure 9.10: Promela "local-global model." Consensus is verified locally, which results in a check only for the updated agent.

which holds by Theorem 1. Where the global model ensured that the "global relation" in Equation 9.1 was maintained, the local model ensured that "local relation" was maintained. This can be seen in Figure 9.10, line 21.

Hybrid Model

The hybrid model, outlined in Figure 9.11, not only checks the local-global relation, but also ensures that consensus is preserved. In this way, it is verifying that the local-global relation in practice is consistent with the theory. System initialization, in this context, is similar to the global model (Figure 9.11, lines 13–17), while agent-state updates are similar to the local model (Figure 9.11, lines 23–25).

```
1  bool initial, state[AGENTS]
2  chan wire[AGENTS] = [BUFFER] of { bool }
3
4  active [AGENTS] proctype agent() {
5    byte n;
6    bool msg, preserved, post;
7
8    chan out = wire[(_pid + 1) % AGENTS];
9    chan inp = wire[_pid];
10   xs out;
11   xr inp;
12
13   if
14     :: state = true;
15        initial = true
16     :: state = false
17   fi;
18
19   out!state[_pid];
20 end:
21   do
22     :: inp?msg ->
23        post = (state[_pid] || msg);
24        assert(state[_pid] <= post);
25        state[_pid] = post;
26
27        n = 0;
28        preserved = false;
29        do
30           :: n < AGENTS ->
31             if
32               :: (state[n] == initial) ->
33                  preserved = true;
34                  break
35               :: else -> n++
36             fi
37          :: else -> break
38        od;
39        assert(preserved);
40
41        out!state[_pid]
42   od
43 }
```

Figure 9.11: Promela "hybrid model." This model performs both a local and a global verification that consensus is maintained after each step.

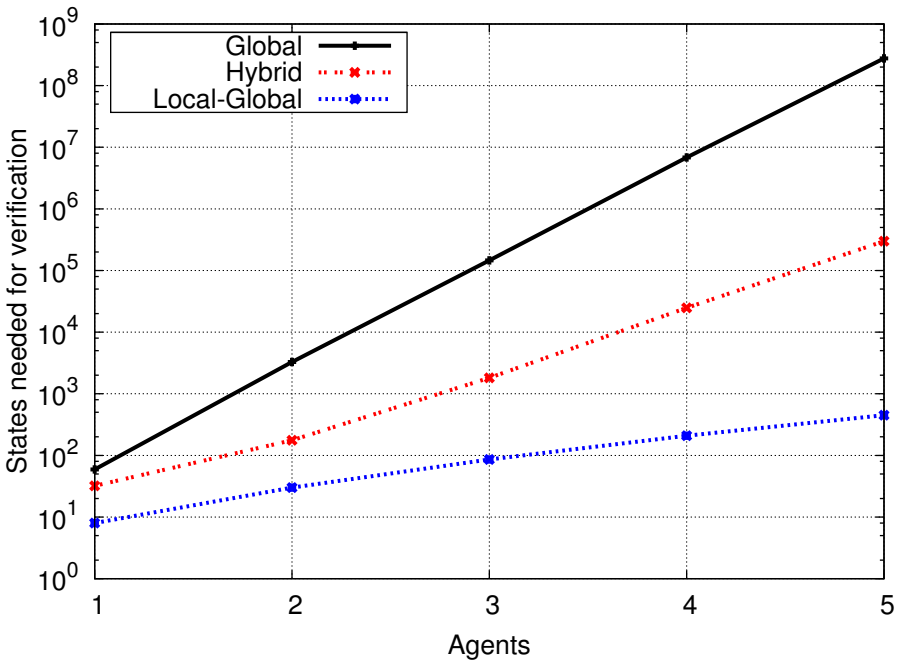


Figure 9.12: Number of states required by SPIN to verify each model as a function of agents in the system.

Once the state is updated, the model verifies that consensus is preserved by checking that the target value within the system has not changed; lines 27–39 detail this process. Recall, `initial` was that target value—the model checks that at least one agents value is the target.

9.2.4 Experimentation

Methodology

Model performance was used to asses the impact that local-global relations have on model checker performance; where performance was measured by the number of states SPIN generated to verify each model, as well the amount of time it took to do so. Measurements, with respect to each, were taken from SPIN's "Pan" output. Verification was run with a varying number of agents—from 1 to 5—and with constant communication buffer size.

The process of verifying a model in SPIN is to first write the model in Promela—the code presented in this chapter corresponds to this step. The Promela file is then given to SPIN to

generate a model checker in C, which is then compiled into a binary executable. This process provides several places for customization of the model, depending what the implementer is interested in verifying and how they want SPIN to operate in doing so. For all models in this section, when generating the model checker in C from Promela, SPIN was passed the `-a` option. The corresponding C files were then compiled with COLLAPSE defined. This option enhances state compression so that system memory is optimized [79]. Along with this option, the local-global models were also compiled with SAFETY defined. At run-time, all models were given a search depth of 10^7 states, which was found to be sufficient for SPIN to perform complete verifications given our physical hardware constraints.[2] Finally, the global model was also passed `-a` at run-time, informing the model checker that verification of a never claim was required.

Results

The state space of each model is shown in Figure 9.12. These values were based on the "states, stored" as reported by the final Pan output. Figure 9.13 shows the verification time required for each model. As the number of agents in the system increase, the local model shows a significant reduction in the amount of resources—both time and states—required for verification.

The primary difference between local-global models and the global version, was the lack of a never claim. Where the global version relied on temporal properties to verify consensus, and thus required a never claim, the local-global models relied on the relation between pre- and post-states of the system. Because system updates only involved a single agent, this meant verifying that a single agents' update maintained the relation. Removing the temporal property had several implications for the model. First, the exclusion of the never claim meant that SPIN was not explicitly verifying that consensus eventually-always held, nor was it focusing its exploration to fair executions. Instead, in the local-global models, SPIN considered all trajectories of the system, as it was only concerned with finding violations

[2] Dual quad-core Intel E5410 processors with approximately 16 GB of allotted system memory.

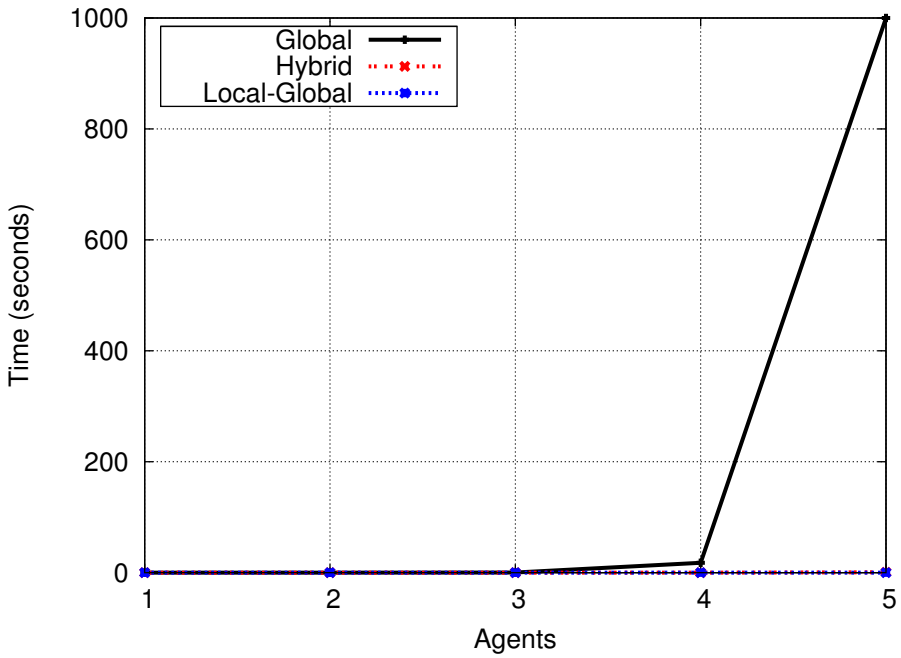


Figure 9.13: Time required by SPIN to verify each model.

of `local_global`. Although it did consider paths that were not fair, it also considered paths that were—it was verification of this subset that was of most concern, as they had implications for correctness.

The number of global variables required for verification for each model also differed. Not counting the channel, the global version used three, the hybrid model kept two, and the local-global model required none. Moreover, in both the global and hybrid case, one of those variables was an array that depended on the number agents. Not only is there a correlation between the number of variables within a model and the amount of memory verifying that model requires, but there is also a correlation between variables and the number of states produced by SPIN. This is largely due to the fact that SPIN must consider all possible values for given variable types and maintain those variables during model execution.

9.2.5 Discussion

The primary objective of this section was to showcase the benefits of trusting the local-global theory when model checking, and to verify, using a model checker, that the theory can be trusted. With respect to the former, in the local-global model, the model checker is just verifying an invariant of the system: that once a Boolean value is true, no disjunctive update can falsify it. In this case, it is that the programmer is trusting the overall theory that was presented in the first portion of the thesis, and wants to ensure that the basic property is maintained in the model. While in general one does not require a model checker to verify such a property, the exercise of doing so for this problem puts the focus on the "states, stored"; there is no ambiguity or confusion (for the reader) over the underlying model.

We also employed a model checker to verify the local-global theory using a particular example; in this case Boolean consensus. To do this in general, one would have to verify more than just the invariant—there would also have to be a liveness component as well. Our global model includes this, but our local-global model does not. That the verification still succeeds in both cases implies that local-global model is sound over the problem space. Our third, hybrid model, which sat at the intersection of the two, substantiated this conclusion.

9.2.6 Related Work

Casadei and Viroli used a model checker to verify self-organizing systems [80, 81]. Their motivation for the use of formal methods was similar to ours: that correctness of a distributed algorithm is difficult to obtain at run-time. In this case, they were interested in the dynamics, and convergence properties, of the collective sort problem [82, 83], whereby agents cluster objects based on a well-defined similarity between those objects. The authors acknowledged that the solution to this problem is dependent on local actions of agents; however, unlike our work, they did not exploit this fact to simplify the verification process.

9.3 Implementation

Our algorithm analysis, along with its representation and verification in Spin and PVS, provided a well-defined structure with which to create executable programs. This section outlines those programs, and looks at methods for ensuring their correctness. Whereas the concepts of local-global relations had an explicit impact on algorithm verification, they had an implicit impact on system design and implementation.

We consider the consensus and algebraic path problems studied earlier. The code-base deviates slightly from the theoretical presentation by including monoid and average consensus within the same framework; however, the principles presented thus far remain. The library was built using two different programming paradigms: object-oriented and functional, with Java and C# being used for the former, and Erlang for the latter. Our presentation of the code follows this organization, with algorithms sectioned by coding practice. Unlike more automated efforts [84, 85, 86, 6], our transformations were done manually.

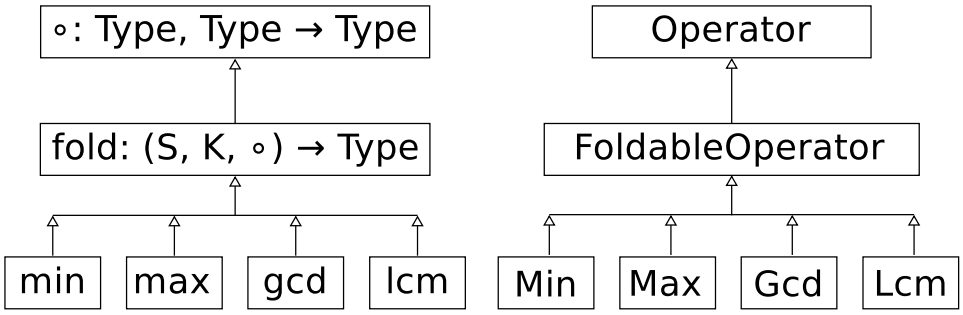
Java and C# are common languages; it is assumed that the reader is familiar with them in particular, or with at least their representation of object-oriented programming in general. Erlang, however, is not quite as popular and therefore deserves a brief overview [8]: Erlang is an interpreted, functional, language that was originally developed by Ericsson to build software for telephone exchange [87]. Software in this domain requires support for large numbers of concurrent and distributed processes—providing that support was a major design requirement for the language. To this end, Erlang has its own implementation of processes, which are neither operating system processes nor user-level threads. The creation and destruction of processes within Erlang require nominal system overhead as they are very lightweight structures that do not share any data. When programming in Erlang, the programmer is encouraged to think of their application as a set of such processes that interact via asynchronous message passing [88]. Sending and receiving messages are native operations within the language to facilitate such design. Thus, not only did Erlang provide a second paradigm in which to study local-global relations, it was an appropriate tool for our algorithms given its support for distributed systems.

9.3.1 Consensus

All three implementations were made up of three components: operators, agents, and infrastructure. Agents exchanged values, updating their state by applying those values to operators. The infrastructure acted as a facilitator for such interactions by providing an implementation of the transition system. Specifically, it composed groups of agents and provided an atomic environment in which they could share and update their state. In this way, the infrastructure had to satisfy the system assumptions from Section 2.1. We did not formally verify that this was the case—we refer the reader to the wide body of literature which addresses this concern [89, 90, 91, 92, 93]. For example, we assumed that the communication amongst agents was fair. This required the communication medium to be reliable, but also had implications for group generation. Groups of agents were created using a random number generator: given the entire set of agents within the system, agents were selected at random and assigned to a given group (agents without a group assignment were thought of as being in a group consisting of only themselves). We assumed that over the course of an execution, random group assignment would never create a permanent partition amongst the agent set.

Object Oriented Implementation

The aforementioned components—operators, agents, and infrastructure—were each abstract objects in the object oriented design. The operator class was refined to obtain specific functions of interest, such as max and gcd. It was analogous to the abstract representation of operators in PVS that were instantiated by concrete theorems. As shown in Figure 9.14, the correspondence between the PVS library structure and the actual object refinement was one-to-one. The parent operator class defined an abstract method, `operate`, that all children were required to implement. This method was a function from a collection of values to a single value. There were two children of this class: one for monoid composition, and another for average. The former defined `operate` as the fold method from PVS (Section 9.1.2). Traditional operators—min, gcd, convex hull—were children of this class. Recall that although



(a) Theory diagram

(b) Object diagram.

Figure 9.14: Operator component diagram.

average is a local-global relation, it is not monoid-composable; thus, it required its own branch in the object hierarchy.

The agent class, also abstract, represented a single agent in our system. To simulate an actual system with multiple agents, several threads were spawned, each containing an agent instantiation. Communication between agents took place via message passing. Each agent stored an operator in its private data. Concrete implementations of the class defined the post-state state-commit semantics of an agent. Depending on the operator, agents were either free to request a new group after updating their state, or waited for all other agents to also update their state before moving on. For example, in the case of min consensus, once an agent updates its state with the lowest of the group's values, that agent is free to move on to another group, irrespective of its neighbors. If such an agent begins a transaction with new neighbors while its previous neighbors have not, its update with new neighbor values will not affect the ability for the system to converge. However, in the case of average consensus, agents cannot move out of their group until all agents have updated their value; doing so could change the overall average of the group. Thus, concrete agent representations within our library had to implement a method expressing how an agent should handle post-state commits.

Functional Implementation

In Erlang, all components of the system were independent processes that implemented a particular component. Thus, unlike the object oriented implementation, agents did not contain an operator; rather, there was a single operator process that they interacted with. Agents were also separate, concurrent, processes. This was much like the object-oriented approach in which they were separate threads. The underlying infrastructure management layer was also similar, starting the operator and agent processes, and assigning agents to groups. Erlang is a message-based language, so transitions commenced through message passing: first between manager and agent, then between agents, then agents to the operator, and finally agents back to managers. This cycle continued until consensus was reached.

9.3.2 Algebraic Path

Each implementation used graphs, agents, and semirings as their underlying, abstract components. Graphs focused merely on traversal, applying agent values to semirings where necessary, and making traversal decisions based on it. As was the case in the consensus implementation, such separation in this case provided a natural design that was modular.

Object Oriented Implementation

Directed graph, graph traversal, edge value, and semiring were the four abstract classes that implemented our graph theory. Given a semiring and directed graph, graph traversal performed a depth first walk, applying the semiring to each vertex it encountered. The traversal class relied on the semiring object to decide whether or not to continue down a particular path. The semiring produced a new edge value, following Definition 18. If the new value was not less-than the old value, traversal along the path was aborted.

Semirings operated over edge values, both were abstract, refined to perform specific types of graph operations. For example, for reachability (Example 9) the edge value was refined to Boolean types, while semiring addition and multiplication were refined to disjunction and

conjunction, respectively. Edge value was also responsible for defining `compareTo`[3] so that they could be compared by the traversal object.

Functional Implementation

Vertices of the graph were autonomous agent processes. Each agent maintained a list of neighbors, which represented edges. Rather than a static traversal algorithm, as with the object-oriented implementation, solutions to path problems were calculated using diffusion. Specifically, traversal started started at a root node, with that node sending requests to each of its neighbors. These neighbors, in parallel, did the same, continuing the request to each of their neighbors. Once a request reached the end of a path, it traversed its way back to the root node via the path from which it originated.

In this way, the algebraic path problem was actually a distributed protocol: agents received a request packet with the current calculation result; they enlisted the semiring agent to update the packet with their own state; and finally, sent a calculation request to their neighbors that contained the new packet. The calculation request, and packet updates, continued throughout the graph until they reached an agent that either had no neighbors, or had already received the said request. At that point, the packet was sent back "up" the path. Note that agents did not respond to the calculation request until they had heard from all of their children.

Much like operator processes in the Erlang consensus implementation, semirings were daemon processes that responded to semiring requests, such as times, plus, and zero. Extending our Erlang implementation, required creating a new daemon that answered to those requests appropriately.

9.3.3 Contract Specification

Our Java implementation uses the Java Modeling Language (JML) [7] to enforce system verifications. The purpose of JML, in general, is to specify the behavior of Java objects, and

[3]Derived from `Comparable` in Java and `IComparable` in C#/Spec#.

```
1 min: THEORY BEGIN
2   min(m, n: real): {p: real | p <= m AND p <= n} =
3     IF m > n THEN n ELSE m ENDIF
4 END min
```

(a) PVS min.

```
1 /*@
2   @ assert(\result <= x && \result <= y &&
3   @        (\result == x || \result == y));
4   @*/
5 double min(double m, double n) {
6   return Math.min(m, n);
7 }
```

(b) Java min augmented with JML.

Figure 9.15: Implementation of min in PVS and Java. The Java implementation is annotated with JML.

to check during run-time that those specifications are upheld. The tool allows programmers to prepend object methods with pre- and post-conditions, and to annotate arbitrary lines of code with system invariants. An advantage of JML is that it supports abstract programming concepts that are closer to mathematics than standard Java, such as set theory and quantification.

The C# implementation was extended to Spec# [94], a programming language based on C# that has native support for correctness specification. Spec# is much like JML in that the programmer can specify pre- and post-conditions, and object invariants; however, it is not a meta-language—code written in Spec# cannot be compiled by a C# compiler.

Consensus

There were two aspects of our implementation that were verified: that the operators performed as expected, and that system state, specifically agent interactions, obeyed our theory. Correctness of either condition can be derived from our theoretical analysis. However, as this is an examination in the process of formal methods, we base such correctness on our derivation of correctness from those methods. In the case of operator verification, this meant

```
1  public class LocalGlobal {
2    private boolean commence;
3    private Operator operator;
4
5    private Vector preState;
6    private Vector postState;
7
8    public LocalGlobal(Operator o) {
9      operator = o;
10     commence = false;
11   }
12
13   public void setPreState(Vector state) {
14     preState = new Vector(state);
15   }
16
17   public void setPostState(Vector state) {
18     postState = new Vector(state);
19     commence = true;
20   }
21
22   invariant(commence ==> operator.relates(preState, postState));
23 }
```

Figure 9.16: An object added to ensure a local-global relation was maintained between states.

that the implementations were equivalent to their PVS representation. Figure 9.15 outlines this process for min: Figure 9.15(a) is the PVS representation of min (previously displayed in Figure 9.4); Figure 9.15(b) is the Java representation of min (lines 5–7), annotated with JML (lines 1–4). Based on the min implementation in PVS, a valid functional transform is one that places a restriction on its return type (line 2), and whose value is equal to one of its input parameters. Both are captured with the JML post-condition checks on lines 2 and 3, respectively. These assertions alleviate the need to know how min is actually implemented, making JML a crucial step in the transformation from PVS to correct software.

Verification of agent interactions ensured that the local-global relation was preserved between transitions. In this case, we used the SPIN model as example. Both JML and Spec# allow the programmer to specify object invariants, which are verified each time the class is modified. A new object was added to the system, `LocalGlobal`, where an invariant based

```
1  public class Infrastructure {
2    private LocalGlobal lg;
3
4    public void groupSubscribe(Agent agent) {
5      /* ... */
6      lg.setPreState(groupTransitionSet);
7      /* ... */
8    }
9
10   public void groupUnsubscribe(Agent agent) {
11     /* ... */
12     lg.setPreState(groupTransitionSet);
13     /* ... */
14   }
15 }
```

Figure 9.17: Maintenance of the `LocalGlobal` object. The pre-state was set prior to group formation, while the post-state was set once all agents were finished with their transition.

on local-global relations was added (Figure 9.16). `LocalGlobal` maintained three variables: an abstract operator, and two collections of agents corresponding to versions of the agent set before and after a transition. The operator class presented in Section 9.3.2 was updated with an abstract `relates` method. Concrete instances of the `Operator` implemented this method with the corresponding local-global relation between states. In the case of min consensus, for example, the returned value was a less-than relationship between states. The infrastructure, responsible for group formation, passed snapshots of the agents to `LocalGlobal` before and after a transition (Figure 9.17). Each time this value was set (lines 6 and 12), the invariant in `LocalGlobal` (line 22) was evaluated.[4] The variable `commence` ensured that the invariant was fully evaluated only after the post-state was set.

We also monitored progress by comparing the result of variant functions applied to the pre- and post-state. Progress was made when the variant function strictly decreased. A terminal state was declared if the variant function in the post state was zero. If enough system transitions were made without strict progress produced a system error was generated. Specification languages were not employed for this check, as it was not an invariant; thus,

[4]The `invariant` keyword and implication operator `==>` were defined in both JML and Spec#.

```
1  public class LocalGlobal {
2    private DirectedGraph preState;
3    private DirectedGraph postState;
4
5    public static boolean relation(DirectedGraph pre,
6                                   DirectedGraph post) {
7
8      while (post.EdgeIterator().hasNext()) {
9        postEdgeValue = post.getEdgeValue(post.EdgeIterator());
10       preEdgeValue = pre.getEdgeValue(post.EdgeIterator());
11
12       if (postEdgeValue.compareTo(preEdgeValue) > 0) {
13           return false;
14       }
15     }
16
17     return true;
18   }
19
20   invariant(postState != null ==>
21             LocalGlobal.relation(preState, postState));
22 }
```

Figure 9.18: Psuedo-Java outlining local-global verification within the algebraic path implementation. Once `postState` is instantiated, the invariant is evaluated. This runs `relation`, which ensures that all edges in the post-state are less-than or equal-to their corresponding edges in the pre-state.

exceptions were thrown to indicate abnormal conditions.

Algebraic Path

Verification concepts within algebraic path were similar to those in consensus: ensure that operators were algebraically sound, and enforce the transition semantic.

Transitions were again specified as object invariants. A `LocalGlobal` class was defined that kept copies of pre- and post-transition graphs, and provided a means of checking that the two graphs fit a local-global relation. Figure 9.18 outlines this process.[5] The invariant is only evaluated if the post-state has been set. When this is the case, the `relation` method

[5]The code is presented in psuedo-Java for space reasons, with variable declarations undefined and JML descriptors removed, however, the idea should be clear. Moreover, it is generic enough that a conversion to Spec# should be straightforward.

checks that, for each edge in the post-state, its value is less-than or equal-to the corresponding edge value in the pre-state (line 12). An instance of `LocalGlobal` was kept, and maintained, in the object responsible for graph traversal. Traversal was a recursive function where the pre-state graph was set prior to the function body and post-state graph set prior to its return.

9.3.4 Discussion

Our verification methodology produced several modular components whose correctness—both in their assumptions and their execution guarantees—was well understood. Knowing what these components were helped to direct the focus of our implementation, along with its implementation. For example, when developing distributed consensus, local-global relations dictated how agents and operators should behave locally, and what guarantees those actions should provide on system transition globally. In algebraic path, we were able to separate graph traversal from system agent interaction.

What to specify within a code-base is a known challenge in practice. It is sometimes too difficult to distinguish when specifications are contributing to the soundness of an implementation versus the correctness of the implemented algorithm. In a recent survey of formal methods used in industry, respondents mentioned such [10]. Local-global relations help alleviate this burden by focusing the specification effort to areas necessary for overall correctness. While other specifications, such as catching null objects or other unexpected value types, are necessary, local-global relations help to separate the domain of the two.

Chapter 10

Conclusion and Future Work

10.1 Future Work

10.1.1 Tools of Formal Methods

Local-global relations should be a native concept in the tools of formal methods. In particular, this would be applicable to tools such as JML and Boogie [95], where programmers augment executable code to check for errors. In order to do this, calls would be inserted at state transitions that specified the pre- and post-state to checked. Such calls would also specify the relation that should be maintained. The tools, in turn, would statically check that these relations were maintained, and further ensure that these relations were maintained during system execution.

With respect to model checking, the concept of local-global relations should be extended to other problems. It has been shown that knowledge of the framework can improve checking performance in a specific case; how to do so in the general case is unclear. Based on the domain-specific assumptions we made in Section 9.2, one might argue that applying local-global relations within a model checker takes insight into the problem, and intuition into how the framework can be applied. Thus, little can be used from the example provided in this thesis when applying the framework to other problems. However, one cannot, at this point, make such a statement definitively: further work—specifically applying the framework to other problems—would determine such applicability.

10.1.2 Other Algebraic Structures and Problems

A natural extension of the framework is to other algebraic structures. Just as the monoid and semiring allowed us to reason about a given class of problems, other structures can potentially allow us to do the same. For example, lattices can be used to describe distributed resource sharing. Let agents in a system be nodes in a lattice; the direction of their outgoing edge determines their priority for a resource. Depending on how the problem is modeled, the agent holding the resource would be at the infimum or supremum of the lattice. If there are several resources in the system, this position would local. When the resource is exchanged, the edges of the agent previously holding the resource would be swapped, effectively making putting it at the other extreme of the lattice. The challenge is in defining the state transition in such a way that agents waiting on a resource locally will receive that global resource eventually.

Local-global relations might also be applied to distributed image analysis. Mathematical morphology [96] is an algebraic method of analysis and processing geometric structures. At its basis are two primitive operations of erosion and dilation, which can be defined using set intersection and union, respectively. Other work has applied the concept to analysing images using associative processors [97]. At the surface morphology has the requirements—an algebraic basis and is parallelizable—to be applicable to our framework.

10.1.3 Other Problem Domains

The problems considered in this thesis were homogeneous in nature: all agents within the system performed the same operation to achieve a common global goal. There are several distributed systems, however, for which this is not the model. These systems are heterogeneous in nature, comprised of a set of components that perform different tasks. An example of such system are web services. Web services are a set of distributed components that interact to provide a particular service, where that service is the result of a remote program execution. A challenge in the field is ensuring that component interactions obey given properties such that their cooperation achieves a given goal [98]. Algebra has been used to assist with such reasoning [99]. Given the locally interactive nature of the problem, and this algebraic basis,

there is potential to utilize the local-global framework for reasoning about system evolution. The extensions required to deal with such heterogeneous systems would add value to the framework overall.

10.1.4 The Tradeoff Between Robustness and Efficiency

Systems that are robust operate correctly in a variety of environments; however, they may operate suboptimally in each environment. The system is robust because even if the environment changes—for example if communication channels fail—the system continue to operate correctly, though possibly with degraded efficiency. By contrast, a fragile system that operates correctly in only one environment can be made extremely efficient for that specific environment; however, if the environment changes the system crashes.

In this thesis we explored the relationship between robustness and efficiency by studying multiagent systems that operate in extremely unreliable and possibly hostile environments. An evaluation of the tradeoff between robustness and efficiency should explore a range of designs from extremely fragile and extremely efficient at one extreme, to extremely robust and possibly very inefficient at the other extreme. This thesis studies only one extreme—robust systems operating in unreliable environments. The thesis briefly compares systems at this extreme with systems at the other extreme—all agents can interact at all times perfectly. A great deal of work remains to be done in exploring the robustness-efficiency tradeoff across the spectrum of designs from fragile systems designed to operate efficiently in a limited number of environments to robust systems that operate correctly even when an adversary controls aspects of the system.

10.2 The Applicability of Local-Global Relations

Local-global relations are a framework for studying a class of distributed systems in which agents are homogeneous in the actions they perform. They allow a developer to reason about overall system dynamics by concentrating on interactions amongst subsets of processes. An

advantage of this methodology is a simpler means of showing system correctness, both in theory, and in practice.

This thesis provided examples of the frameworks usability by applying it to established algorithms within the field, such as consensus and graph problems. In such cases, local-global relations were identified and then used to show invariant and liveness properties. Working with such relations, rather than complete system models, reduced the proof obligations required to show correctness.

The framework was also included in the software development cycle by applying it to tools of formal methods. Local-global relations promoted an algebraic representation of the system, which subsequently simplified their representation within a theorem prover. By concentrating the verification effort on ensuring that process interactions were correct, efficiency in the use of model checkers and specification languages was also improved. Finally, in creating executable code, software components such as Java classes and PVS libraries were developed that both met a specification, and were reusable.

Traditional approaches to studying algorithm and software correctness have a bias toward global analysis by considering system dynamics and convergence as a whole. The approach is natural, given that correctness is generally specified as a global system property. However, as the number of autonomous processes making up those systems increase, so too will the complexity involved with studying correctness from that vantage. For system design and analysis to meet future demands, alternative methods must be developed—local-global relations are a contribution towards this effort.

Appendix A

Auxiliary Proofs

A.1 Reverse Induction

The objective of this section is to prove a generic reverse induction theorem over finite non-empty sets. Throughout the thesis, we assumed the set of agents within a system was finite and non-empty; thus, the theorems proved herein are valid for agent sets.

Consider a non-empty finite set $\mathcal{K}$. Denote by K and by k a subset and an element of $\mathcal{K}$, respectively. Further, $\hat{K}$ represents the full set of $\mathcal{K}$. Finally, R is a predicate over sets of agents. Superscript notation is used to differentiate instances of a given type from generalizations. For example, if $\forall K : K \neq \emptyset$, then K^1 would denote an instance of K that is not empty.

Theorem 24

$$R(\hat{K}) \wedge \big(\forall k, K : k \notin K \wedge R(K \cup \{k\}) \implies R(K)\big) \implies \forall K : R(K) \tag{A.1}$$

□

The strategy for proving Theorem 24 is to map the set K to natural numbers using cardinality properties over finite sets. To do so, we first need to define a corresponding induction theorem over natural numbers.

Theorem 25 (Reverse Induction) *Let* $n \in \mathbb{N}_1$ *where* $n \leq \left|\hat{K}\right|$

$$P(\left|\hat{K}\right|) \wedge \left(\forall n \colon n > 1 \wedge P(n) \implies P(n-1)\right) \implies \forall n \colon P(n) \tag{A.2}$$

□

Lemma 10

$$R(\hat{K}) \wedge \left(\forall k, K \colon R(K \cup \{k\}) \implies R(K)\right) \implies \forall K \colon R(K) \tag{A.3}$$

□

Proof Let $n \in \mathbb{N}_1$ where $n \leq \left|\hat{K}\right|$. Assume

$$\forall n, K \colon |K| = n \implies R(K). \tag{A.4}$$

There are three cases to consider:

1. That $|K|$ meets the type restriction on n; that is, $|K| \geq 1$. Because K is a non-empty set, this is trivially true.

2. That A.4 holds: $(\forall n, K \colon |K| \implies R(K)) \implies R(K^1)$. The implication holds when $K = K^1$ and $n = |K^1|$.

3. That A.4 does not hold:

$$(\forall k, K \colon R(K \cup \{k\}) \implies R(K)) \implies (\forall n, K \colon |K| = n \implies R(K)).$$

   This is shown using reverse induction on n (Theorem 25)

   Base Case Show that the predicate R holds given that the size of K is equal to the

size of the full set.

$$\left|K^1\right| = \left|\hat{K}\right| \qquad \bigwedge \tag{A.5}$$

$$R^1(\hat{K}) \qquad \bigwedge$$

$$\forall k, K : R^1(K \cup \{k\}) \implies R^1(K)$$

$$\implies$$

$$R^1(K^1). \tag{A.6}$$

From A.5, $K^1 = \hat{K}$. Replacing K^1 with $\hat{K}$ in A.6, the implication holds.

Inductive Step Show that the predicate R holds irrespective of the cardinality of K.

$$\forall K : |K| = n^1 \implies R(K) \qquad \bigwedge \tag{A.7}$$

$$\forall k, K : R(K \cup \{k\}) \implies R(K) \qquad \bigwedge \tag{A.8}$$

$$n^1 > 1 \qquad \bigwedge \tag{A.9}$$

$$\left|K^1\right| = n^1 - 1 \tag{A.10}$$

$$\implies$$

$$R^1(K^1)$$

Based on the cardinality assumptions, the following can be inferred:

$$\underbrace{n^1 > 1}_{A.9} \wedge \underbrace{\left|K^1\right| = n^1 - 1}_{A.10} \wedge \underbrace{n \leq \left|\hat{K}\right|}_{def.} \implies \left|K^1\right| < \left|\hat{K}\right|$$

$$\implies \exists k : k \notin K^1.$$

Let k^1 be the instance of $k \notin K^1$. Let K equal K^1 in A.8 and $K^1 \cup \{k^1\}$ in A.7.

The theorem follows:

$$\begin{aligned} \left|K^1\right| = n^1 - 1 &\implies \left|K^1 \cup \left\{k^1\right\}\right| = n^1 \\ &\implies R^1(K^1 \cup \left\{k^1\right\}) \\ &\implies R^1(K^1). \end{aligned}$$

■

Using Lemma 10, we can prove Theorem 24.

Proof (Theorem 24)

$$R^1(\hat{K}) \wedge \left(\forall k, K : k \notin K \wedge R^1(K \cup \{k\}) \implies R^1(K)\right) \implies R^1(K^1) \tag{A.11}$$

Assuming Lemma 10, where R is R^1,

$$\left(\forall k, K : R^1(K \cup \{k\}) \implies R^1(K)\right) \implies \forall K : R^1(K), \tag{A.12}$$

there are two cases to discharge:

1. The consequent in A.12, along with the antecedent in A.11, implies $R^1(K^1)$. This is true since, even without the antecedents in A.11,

   $$\forall K : R^1(K) \implies R^1(K^1)$$

   when $K = K^1$.

2. The assumptions in A.11 imply either the antecedent in A.12 or $R^1(K^1)$. Formally,

$$\begin{array}{ll} R^1(\hat{K}) & \bigwedge \\ \forall k, K : k \notin K \wedge R^1(K \cup \{k\}) \implies R^1(K) & \\ \implies & \\ \forall k, K : R^1(K \cup \{k\}) \implies R^1(K) & \bigvee \\ R^1(K^1). & \end{array}$$

Skolemizing and simplifying, we have

$$\begin{array}{ll} R^1(K^2 \cup \{k^1\}) & \bigwedge \\ \forall k, K : k \notin K \wedge R^1(K \cup \{k\}) \implies R^1(K) & \text{(A.13)} \\ \implies & \\ R^1(K^2). & \end{array}$$

Let $k = k^1$ and $K = K^2$ in A.13. The theorem follows, since

$$R^1(K^2 \cup \{k^1\}) \wedge K^2(k^1) \implies R^1(K^2).$$

■

A.2 Monoids

A.2.1 Abstract Composition

In the definition of $\oplus$-composition (Equation 3.2), an agent is arbitrarily selected from the set of agents K. That selection is assumed to be based on the axiom of choice [100]. Throughout this subsection, we refer to the function defining such a choice with $\epsilon\colon 2^{\mathcal{A}} \to \mathcal{A}$.

Lemma 11 *Let k be an element of K,*

$$\bigoplus_{k \in K} S(k) = S(k) \oplus \bigoplus_{j \in K \setminus \{k\}} S(j).$$

□

PROOF The proof is by induction on K:

Base Case When $K = \emptyset$:

$$k \in K \wedge K = \emptyset \implies \bar{0} = S(k) \oplus \bigoplus_{j \in K \setminus \{k\}} S(j).$$

Inductive Step Let an element raised to the n denote a bound, or skolemized, instance of its respective type, where $n \in \mathbb{N}_0$. For example, K^1 is an in instance of $K \subset \mathcal{A}$ and S^1 is an instance of $S \in \mathcal{S}$. Elements without superscript notation are the opposite. Assume $K^1 \neq \emptyset$, and $S \in \mathcal{S}$, $K \subset \mathcal{A}$, and $k \in K$,

$$|K| < \left|K^1\right| \implies \left(k \in K \implies \bigoplus_{k \in K} S = S(k) \oplus \bigoplus_{K \setminus \{k\}} S \right) \qquad \bigwedge \tag{A.14}$$

$$k^1 \in K^1 \tag{A.15}$$

$$\implies$$

$$S^1(\epsilon(K^1)) \oplus \bigoplus_{K^1 \setminus \{\epsilon(K^1)\}} S^1 = S^1(k^1) \oplus \bigoplus_{K^1 \setminus \{k^1\}} S^1. \tag{A.16}$$

First, we assume two instances of A.14. We must choose appropriate instantiations for the free variables in each. In the first, let $K = K^1 \setminus \{k^1\}$, $S = S^1$, and $k = \epsilon(K^1)$; in the second, $K = K^1 \setminus \{\epsilon(K^1)\}$, $S = S^1$, and k^1. Choices for K are both valid assumptions since $|K^1 \setminus \{k^1\}| < |K^1|$ and $|K^1 \setminus \{\epsilon(K^1)\}| < |K^1|$. Substituted into A.14, with

cardinality assumptions removed for brevity, we have

$$\epsilon(K^1) \in K^1 \setminus \left\{k^1\right\} \implies \bigoplus_{K^1 \setminus \{k^1\}} S^1 = \left(S^1(\epsilon(K^1)) \oplus \bigoplus_{K^1 \setminus \{k^1\} \setminus \{\epsilon(K^1)\}} S^1 \right) \bigwedge \quad \text{(A.17)}$$

$$k^1 \in K^1 \setminus \left\{\epsilon(K^1)\right\} \implies \bigoplus_{K^1 \setminus \{\epsilon(K^1)\}} S^1 = \left(S(k^1) \oplus \bigoplus_{K^1 \setminus \{\epsilon(K^1)\} \setminus \{k^1\}} S^1 \right). \quad \text{(A.18)}$$

Before we can apply A.17 or A.18 to A.14, we must discharge their respective antecedents.

1. From A.17,

$$\epsilon(K^1) \notin K^1 \setminus \left\{k^1\right\} \implies S^1(\epsilon(K^1)) \oplus \bigoplus_{K^1 \setminus \{\epsilon(K^1)\}} S^1 = S^1(k^1) \oplus \bigoplus_{K^1 \setminus \{k^1\}} S^1,$$

   which holds since, $\epsilon(K^1) \notin K^1 \setminus \{k^1\} \implies \epsilon(K^1) = k^1$.

2. From A.18,

$$k^1 \notin K^1 \setminus \left\{\epsilon(K^1)\right\} \implies S^1(\epsilon(K^1)) \oplus \bigoplus_{K^1 \setminus \{\epsilon(K^1)\}} S^1 = S^1(k^1) \oplus \bigoplus_{K^1 \setminus \{k^1\}} S^1,$$

   which holds since, $k^1 \notin K^1 \setminus \{\epsilon(K^1)\} \implies k^1 = \epsilon(K^1)$.

$$\bigoplus_{K^1 \setminus \{k^1\}} S^1 = \left(S^1(\epsilon(K^1)) \oplus \bigoplus_{K^1 \setminus \{k^1\} \setminus \{\epsilon(K^1)\}} S^1 \right) \bigwedge \quad \text{(A.19)}$$

$$\bigoplus_{K^1 \setminus \{\epsilon(K^1)\}} S^1 = \left(S(k^1) \oplus \bigoplus_{K^1 \setminus \{\epsilon(K^1)\} \setminus \{k^1\}} S^1 \right) \bigwedge \quad \text{(A.20)}$$

$$k^1 \in K^1$$

$$\implies$$

$$S^1(\epsilon(K^1)) \oplus \bigoplus_{K^1 \setminus \{\epsilon(K^1)\}} S^1 = S^1(k^1) \oplus \bigoplus_{K^1 \setminus \{k^1\}} S^1 \quad \text{(A.21)}$$

Since $K^1 \setminus \{k^1\} \setminus \{\epsilon(K^1)\} = K^1 \setminus \{\epsilon(K^1)\} \setminus \{k^1\}$, we can replace A.19 and A.20 in A.21:

$$S^1(\epsilon(K^1)) \oplus S(k^1) \oplus \bigoplus_{K^1 \setminus \{\epsilon(K^1)\} \setminus \{k^1\}} S^1 = S^1(k^1) \oplus S^1(\epsilon(K^1)) \oplus \bigoplus_{K^1 \setminus \{\epsilon(K^1)\} \setminus \{k^1\}} S^1.$$

The equality holds from the commutativity of $\oplus$. ■

A.2.2 Examples

Min/Max

Definition 30 The minimum and maximum values over a set of natural numbers S is defined:

$$\min\colon i, j \to \text{if } i < j \text{ then } i \text{ else } j$$
$$\max\colon i, j \to \text{if } i < j \text{ then } j \text{ else } i.$$ □

Theorem 26 (Min is Idempotent)

$$\forall i\colon \min(i, i) = i$$ □

PROOF This follows from the definition: $i \not< i$, so the result will be i. ■

Theorem 27 (Min is Commutative)

$$\forall i, j\colon \min(i, j) = \min(j, i)$$ □

PROOF From the definition, the theorem can be rewritten

$$\Big(\text{if } i < j \text{ then } i \text{ else } j\Big) = \Big(\text{if } j < i \text{ then } j \text{ else } i\Big),$$

which is equivalent. ■

Theorem 28 (Min is Associative)

$$\forall i, j, k \colon \min(i, \min(j, k)) = \min(\min(i, j), k)$$

□

PROOF Using the definition of min, we have several if-then branches to consider. However, as was the case in the proof of Theorem 27, exhaustive analysis shows their equivalence. ■

Theorem 29 (Min is Closed)

$$\forall i, j \in \mathbb{Z} \colon \min(i, j) \in \mathbb{Z}$$

□

PROOF Follows from the definition of min. ■

Proofs for max are symmetric with respect to the ordering relation.

GCD/LCM

The greatest common divisor (gcd) and least common multiple (lcm) are two binary operators that fit our operator specification. Inherent in their definition is the concept of divisibility: we say that a number i "divides" another number j.

Definition 31 (Divisibility)

$$\text{divides} \colon i, j \to \exists x \colon j = i \times x$$

□

Definition 32 (Greatest Common Divisor)

$$\gcd \colon i, j \to \max\Big(\{k \mid \text{divides}(k, i) \wedge \text{divides}(k, j)\}\Big)$$
$$\text{lcm} \colon i, j \to \min\Big(\{k \mid \text{divides}(i, k) \wedge \text{divides}(j, k)\}\Big)$$

where i, j, and k are all positive natural numbers.[1] □

[1] In the case of lcm, it is imperative that neither i nor j be zero. For gcd, however, this condition can be relaxed: either variable can be zero, but not both. In the case of our proof sketch, we assume the

Whereas proofs for max and min were completely symmetric, the same relationship between gcd and lcm is not as straightforward; thus we give their proofs separate treatment. We start with gcd, specifically, a supporting lemma for its idempotence.

Lemma 12

$$\forall i\colon\ \gcd(0, i) = i$$

□

Theorem 30 (GCD is Idempotent)

$$\forall i\colon\ \gcd(i, i) = i$$

□

Proof

$$\begin{aligned}
\gcd(i, i) &= i \\
\{\text{From Lemma 12}\} & \\
\gcd(i, i) &= \gcd(0, i) \\
\{\text{From the definition of gcd}\} & \\
\max\Big(\{\text{divides}(k, i) \land \text{divides}(k, i)\}\Big) &= \max\Big(\{\text{divides}(0, i) \land \text{divides}(0, i)\}\Big) \\
\{\text{Applying extensionality and simplifying}\} & \\
\text{divides}(k, i) &= \text{divides}(k, 0) \land \text{divides}(k, i) \\
\{\text{if and only if equivalence}\} & \\
\text{divides}(k, i) &\iff \text{divides}(k, 0) \land \text{divides}(k, i)
\end{aligned}$$

lcm conditions for both operators; in our formal setting, however, gcd's possible zero value is taken into consideration.

The final line yields three subgoals

$$\text{divides}(k, i) \implies \text{divides}(k, 0) \tag{A.22}$$

$$\text{divides}(k, i) \implies \text{divides}(k, i) \tag{A.23}$$

$$\text{divides}(k, 0) \wedge \text{divides}(k, i) \implies \text{divides}(k, i). \tag{A.24}$$

Equation A.22, even without the antecedent, is true the definition. Equation A.23 and Equation A.24 hold trivially. ■

Theorem 31 (GCD is Commutative)

$$\forall i, j, k\colon\ \gcd(i, j) = \gcd(j, i)$$

□

Proof

$$\gcd(i, j) = \gcd(j, i)$$

$$\{\text{By definition}\}$$

$$\max\Big(\{\text{divides}(k, i) \wedge \text{divides}(k, j)\}\Big) = \max\Big(\{\text{divides}(k, j) \wedge \text{divides}(k, i)\}\Big)$$

$$\{\text{Applying extensionality and simplifying}\}$$

$$\text{divides}(k, i) \wedge \text{divides}(k, j) = \text{divides}(k, j) \wedge \text{divides}(k, i)$$

$$\{\text{if and only if equivalence}\}$$

$$\text{divides}(k, i) \wedge \text{divides}(k, j) \iff \text{divides}(k, j) \wedge \text{divides}(k, i)$$

$$\{\text{True by conjunctive commutativity}\}$$

■

The following lemmas relate gcd and divides over three positive natural numbers. They aid in the subsequent proof of gcd associativity.

Lemma 13

$$\forall i, j, k\colon\ \text{divides}(i, \gcd(j, k)) \implies \text{divides}(i, j) \wedge \text{divides}(i, k)$$

□

Lemma 14

$$\forall i, j, k : \text{divides}(i, j) \wedge \text{divides}(i, k) \implies \text{divides}(i, \gcd(j, k))$$

□

Theorem 32 (GCD is Associative)

$$\forall i, j : \gcd(i, \gcd(j, k)) = \gcd(\gcd(i, j), k)$$

□

Proof

$$\gcd(i, \gcd(j, k)) = \gcd(\gcd(i, j), k)$$

{From the definition of gcd}

$$\max\Big(\{\text{divides}(z, i) \wedge \text{divides}(z, \gcd(j, k))\}\Big) = \max\Big(\{\text{divides}(z, \gcd(i, j)) \wedge \text{divides}(z, k)\}\Big)$$

{Applying extensionality and simplifying}

$$\text{divides}(z, i) \wedge \text{divides}(z, \gcd(j, k)) = \text{divides}(z, \gcd(i, j)) \wedge \text{divides}(z, k)$$

{if and only if equivalence}

$$\text{divides}(z, i) \wedge \text{divides}(z, \gcd(j, k)) \iff \text{divides}(z, \gcd(i, j)) \wedge \text{divides}(z, k)$$

The final line yields four subgoals

$$\text{divides}(z, i) \wedge \text{divides}(z, \gcd(j, k)) \implies \text{divides}(z, \gcd(i, j)) \tag{A.25}$$

$$\text{divides}(z, i) \wedge \text{divides}(z, \gcd(j, k)) \implies \text{divides}(z, k) \tag{A.26}$$

$$\text{divides}(z, \gcd(i, j)) \wedge \text{divides}(z, k) \implies \text{divides}(z, i) \tag{A.27}$$

$$\text{divides}(z, \gcd(i, j)) \wedge \text{divides}(z, k) \implies \text{divides}(z, \gcd(j, k)). \tag{A.28}$$

Equation A.25 and A.28 hold from Lemma 13 and Lemma 14; Equation A.27 and A.26 hold directly from Lemma 13.

■

We turn our attention to equivalent proofs with respect to the lcm. As mentioned earlier, the subtle difference is significant enough to warrant separate treatment.

Lemma 15

$$\forall i\colon\ \mathrm{lcm}(1,i) = i$$

□

Theorem 33 (LCM is Idempotent)

$$\forall i\colon\ \mathrm{lcm}(i,i) = i$$

□

Proof

$$\begin{aligned}
&\mathrm{lcm}(i,i) = i \\
&\{\text{From Lemma 15}\} \\
&\mathrm{lcm}(i,i) = \mathrm{lcm}(1,i) \\
&\{\text{From the definition of lcm}\} \\
&\min\Big(\{\mathrm{divides}(\mathrm{lcm}(1,i),k) \wedge \mathrm{divides}(\mathrm{lcm}(1,i),k)\}\Big) = \min\Big(\{\mathrm{divides}(1,k) \wedge \mathrm{divides}(i,k)\}\Big) \\
&\{\text{Applying extensionality and simplifying}\} \\
&\mathrm{divides}(\mathrm{lcm}(1,i),k) = \mathrm{divides}(1,k) \wedge \mathrm{divides}(i,k) \\
&\{\text{if and only if equivalence}\} \\
&\mathrm{divides}(\mathrm{lcm}(1,i),k) \iff \mathrm{divides}(1,k) \wedge \mathrm{divides}(i,k)
\end{aligned}$$

The final line yields three subgoals

$$\mathrm{divides}(\mathrm{lcm}(1,i),k) \implies \mathrm{divides}(1,k) \tag{A.29}$$

$$\mathrm{divides}(\mathrm{lcm}(1,i),k) \implies \mathrm{divides}(i,k) \tag{A.30}$$

$$\mathrm{divides}(1,k) \wedge \mathrm{divides}(i,k) \implies \mathrm{divides}(\mathrm{lcm}(1,i),k). \tag{A.31}$$

Equation A.29 holds trivially. Equation A.30 and A.31 hold by Lemma 15. ■

Theorem 34 (LCM is Commutative)

$$\forall i, j : \ \mathrm{lcm}(i, j) = \mathrm{lcm}(j, i)$$

□

Proof

$$\begin{aligned}
&\mathrm{lcm}(i, j) = \mathrm{lcm}(j, i) \\
&\{\text{From the definition of lcm}\} \\
&\max\Big(\{\mathrm{divides}(i, k) \wedge \mathrm{divides}(j, k)\}\Big) = \max\Big(\{\mathrm{divides}(j, k) \wedge \mathrm{divides}(i, k)\}\Big) \\
&\{\text{Applying extensionality and simplifying}\} \\
&\mathrm{divides}(i, k) \wedge \mathrm{divides}(j, k) = \mathrm{divides}(j, k) \wedge \mathrm{divides}(i, k) \\
&\{\text{if and only if equivalence}\} \\
&\mathrm{divides}(i, k) \wedge \mathrm{divides}(j, k) \iff \mathrm{divides}(j, k) \wedge \mathrm{divides}(i, k) \\
&\{\text{True by conjunctive commutativity}\}
\end{aligned}$$

■

Lemma 16

$$\forall i, j, k : \ \mathrm{divides}(\mathrm{lcm}(i, j), k) \implies \mathrm{divides}(i, k) \wedge \mathrm{divides}(j, k)$$

□

Lemma 17

$$\forall i, j, k : \mathrm{divides}(i, k) \wedge \mathrm{divides}(j, k) \implies \mathrm{divides}(\mathrm{lcm}(i, j), k)$$

□

Theorem 35 (LCM is Associative)

$$\forall i, j, k : \ \mathrm{lcm}(i, \mathrm{lcm}(j, k)) = \mathrm{lcm}(\mathrm{lcm}(i, j), k)$$

□

Proof

$$\begin{aligned}
&\operatorname{lcm}(i, \operatorname{lcm}(j,k)) = \operatorname{lcm}(\operatorname{lcm}(i,j),k) \\
&\qquad \{\text{From the definition of lcm}\} \\
&\min\Big(\{\operatorname{divides}(i,z) \wedge \operatorname{divides}(\operatorname{lcm}(j,k),z)\}\Big) = \min\Big(\{\operatorname{divides}(\operatorname{lcm}(i,j),z) \wedge \operatorname{divides}(k,z)\}\Big) \\
&\{\text{Applying extensionality and simplifying}\} \\
&\qquad \operatorname{divides}(i,z) \wedge \operatorname{divides}(\operatorname{lcm}(j,k),z) = \operatorname{divides}(\operatorname{lcm}(i,j),z) \wedge \operatorname{divides}(k,z) \\
&\qquad \{\text{if and only if equivalence}\} \\
&\qquad \operatorname{divides}(i,z) \wedge \operatorname{divides}(\operatorname{lcm}(j,k),z) \iff \operatorname{divides}(\operatorname{lcm}(i,j),z) \wedge \operatorname{divides}(k,z)
\end{aligned}$$

The final line yields four subgoals

$$\operatorname{divides}(i,z) \wedge \operatorname{divides}(\operatorname{lcm}(j,k),z) \implies \operatorname{divides}(\operatorname{lcm}(i,j),z) \tag{A.32}$$

$$\operatorname{divides}(i,z) \wedge \operatorname{divides}(\operatorname{lcm}(j,k),z) \implies \operatorname{divides}(k,z) \tag{A.33}$$

$$\operatorname{divides}(\operatorname{lcm}(i,j),z) \wedge \operatorname{divides}(k,z) \implies \operatorname{divides}(i,z) \tag{A.34}$$

$$\operatorname{divides}(\operatorname{lcm}(i,j),z) \wedge \operatorname{divides}(k,z) \implies \operatorname{divides}(\operatorname{lcm}(j,k),z). \tag{A.35}$$

Equations A.32 and A.35 hold from Lemmas 16 and 17; Equations A.34 and A.33 hold directly from Lemma 16. ■

Convex Hull

From Example 8, we denote by $\mathcal{C}_h \colon P \to P$ the convex hull of a set of points that produces another set of points.

Theorem 36 (Super Idempotent)

$$\forall p, q \in P \colon \mathcal{C}_h(p \cup q) = \mathcal{C}_h(p \cup \mathcal{C}_h(q))$$

□

Proofs are left to concrete definitions $\mathcal{C}_h$.

Theorem 37 (Idempotent)

$$\forall p \colon \mathcal{C}_h(p) = \mathcal{C}_h(\mathcal{C}_h(p))$$

□

PROOF Holds by substituting $\emptyset$ for p in Theorem 36

$$\mathcal{C}_h(p \cup \emptyset) = \mathcal{C}_h(\mathcal{C}_h(p \cup \emptyset) \cup \emptyset)$$

{From principles of logic: $x \cup \emptyset = x$}

$$\mathcal{C}_h(p) = \mathcal{C}_h(\mathcal{C}_h(p)).$$

■

Theorem 38 (Commutative)

$$\forall p, q \in P \colon \mathcal{C}_h(p \cup q) = \mathcal{C}_h(q \cup p)$$

□

PROOF Holds by union commutativity: $p \cup q = q \cup p$. ■

Theorem 39 (Associative)

$$\forall p, q, r \in P \colon \mathcal{C}_h(\mathcal{C}_h(p \cup q) \cup r) = \mathcal{C}_h(p \cup \mathcal{C}_h(q \cup r))$$

□

Proof

$$\begin{aligned}
\mathcal{C}_h(\mathcal{C}_h(p \cup q) \cup r) &= \mathcal{C}_h(p \cup \mathcal{C}_h(q \cup r)) \\
\{&\text{From Theorem 38}\} \\
\mathcal{C}_h(r \cup \mathcal{C}_h(p \cup q)) &= \\
\{&\text{From Theorem 36}\} \\
\mathcal{C}_h(r \cup p \cup q) &= \\
\{&\text{From union commutativity}\} \\
\mathcal{C}_h(p \cup q \cup r) &= \\
\{&\text{From Theorem 36}\} \\
\mathcal{C}_h(p \cup \mathcal{C}_h(q \cup r)) &
\end{aligned}$$

■

A.3 Mean Square Error

Lemma 18 *For all $K \subset \mathcal{A}$ and $j \in \mathcal{A} \setminus K$, where $S \xrightarrow{K} S'$,*

$$(\text{MSE}(S, K) \geq \text{MSE}(S', K)) \implies (\text{MSE}(S, K \cup \{j\}) \geq \text{MSE}(S', K \cup \{j\})) \tag{A.36}$$

□

Proof We will add the unchanged agent to the antecedent through a series of rewrites.

$$\begin{aligned}
\text{MSE}(S, K) &\geq \text{MSE}(S', K) \\
\text{MSE}(S, K) - \text{MSE}(S', K) &\geq 0 \\
\frac{1}{|K|}\sum_{k \in K}(S(k) - \text{AM}(S, K))^2 - \frac{1}{|K|}\sum_{k \in K}(S'(k) - \text{AM}(S', K))^2 &\geq \\
\frac{1}{|K|}\left(\sum_{k \in K}(S(k) - \text{AM}(S, K))^2 - \sum_{k \in K}(S'(k) - \text{AM}(S', K))^2\right) &\geq .
\end{aligned}$$

Since $|K| > 0$, $\frac{1}{|K|}$ can be removed. Further, since the transition is a local-global relation,

$\text{AM}(S, K) = \text{AM}(S', K)$. Continuing,

$$\sum_{k \in K} (S(k) - \text{AM}(S, K))^2 - \sum_{k \in K} (S'(k) - \text{AM}(S, K))^2 \geq 0$$

$$\sum_{k \in K} \left(S(k)^2 - 2 \cdot S(k) \cdot \text{AM}(S, K) - S'(k)^2 + 2 \cdot S'(k) \cdot \text{AM}(S, K) \right) \geq$$

$$\sum_{k \in K} \left(S(k)^2 - S'(k)^2 \right) - 2 \cdot \text{AM}(S, K) \sum_{k \in K} S(k) + 2 \cdot \text{AM}(S, K) \sum_{k \in K} S'(k) \geq .$$

Note that,

$$\sum_{k \in K} S(k) = |K| \cdot \text{AM}(S, K) \qquad \bigwedge$$

$$\sum_{k \in K} S'(k) = |K| \cdot \text{AM}(S', K) = |K| \cdot \text{AM}(S, K).$$

Hence,

$$\sum_{k \in K} \left(S(k)^2 - S'(k)^2 \right) - 2 \cdot \text{AM}(S, K) \cdot |K| \cdot \text{AM}(S, K) + 2 \cdot \text{AM}(S, K) \cdot |K| \cdot \text{AM}(S, K) \geq 0$$

$$\sum_{k \in K} \left(S(k)^2 - S'(k)^2 \right) \geq .$$

We now add the agent j to build the consequent. Note that,

$$S(j) = S'(j) \qquad \bigwedge$$

$$\text{AM}(S, K \cup \{j\}) = \text{AM}(S', K \cup \{j\}),$$

hence

$$\Bigg(\sum_{k \in K} \left(S(k)^2 - S'(k)^2 \right) + S(j)^2 - S'(j)^2 + \\ (|K|+1) \cdot \mathrm{AM}(S, K \cup \{j\})^2 - (|K|+1) \cdot \mathrm{AM}(S', K \cup \{j\})^2 - \\ 2 \cdot \mathrm{AM}(S, K \cup \{j\}) \cdot (|K|+1) \cdot \mathrm{AM}(S, K \cup \{j\}) + \\ 2 \cdot \mathrm{AM}(S', K \cup \{j\}) \cdot (|K|+1) \cdot \mathrm{AM}(S', K \cup \{j\}) \Bigg) \geq 0.$$

Note that,

$$\begin{aligned} & 2 \cdot \mathrm{AM}(S, K \cup \{j\}) \cdot (|K|+1) \cdot \mathrm{AM}(S, K \cup \{j\}) \\ = \; & 2 \cdot \mathrm{AM}(S, j \cup \{K\}) \cdot \sum_{k \in K \cup \{j\}} S(k). \end{aligned}$$

Likewise for S'.

$$\Bigg(\sum_{k \in K \cup \{j\}} S(k)^2 - 2\,\mathrm{AM}(S, K \cup \{j\}) \sum_{k \in K \cup \{j\}} S(k) + \mathrm{AM}(S, K \cup \{j\})^2 (|K|+1) - \\ \sum_{k \in K \cup \{j\}} S'(k)^2 - 2\,\mathrm{AM}(S', K \cup \{j\}) \sum_{k \in K \cup \{j\}} S'(k) + \mathrm{AM}(S', K \cup \{j\})^2 (|K|+1) \Bigg) \geq 0.$$

Replacing the definition of MSE

$$\begin{aligned} (|K|+1)\,\mathrm{MSE}(S, K \cup \{j\}) - (|K|+1)\,\mathrm{MSE}(S', K \cup \{j\}) &\geq 0 \\ (|K|+1)\,(\mathrm{MSE}(S, K \cup \{j\}) - \mathrm{MSE}(S', K \cup \{j\})) &\geq \\ \mathrm{MSE}(S, K \cup \{j\}) - \mathrm{MSE}(S', K \cup \{j\}) &\geq \\ \mathrm{MSE}(S, K \cup \{j\}) \geq \mathrm{MSE}(S', K \cup \{j\}). & \quad \blacksquare \end{aligned}$$

A.4 Permutations

Lemma 19 (Lemma 5)

$$\rho(S, S') \wedge \rho(S', S'') \implies \rho(S, S'')$$

□

Proof From the antecedent, we can assume there exist two functions, $(f_1, f_2) : \mathcal{A} \to \mathcal{A}$, such that both are bijections and

$$\forall k \in \mathcal{A} \colon S(k) = S'(f_1(k)) \wedge S'(k) = S''(f_2(k)).$$

We use the composition of these functions, $f_2 \circ f_1$, to instantiate h in the consequent. Thus, from Definition 20, we must show that this composition is injective, surjective, and permutes elements of S and S''. Throughout, let x, x', and x'' be specific instances of agents in $\mathcal{A}$.

Injective We are required to show

$$f_2(f_1(x)) = f_2(f_1(x')) \implies x = x'. \tag{A.37}$$

From the antecedent, for all j, k in $\mathcal{A}$, we know

$$f_1(j) = f_1(k) \implies j = k \tag{A.38}$$

$$f_2(j) = f_2(k) \implies j = k. \tag{A.39}$$

Let j and k be x and x', respectively, in Equation A.38, and $f_1(x)$ and $f_1(x')$, respectively, in A.39. Adding the antecedent from Equation A.37 to these assumptions, we have the following:

$$\begin{aligned} &f_2(f_1(x)) = f_2(f_1(x')) && \bigwedge \\ &f_2(f_1(x)) = f_2(f_1(x')) \implies f_1(x) = f_1(x') && \bigwedge \\ &f_1(x) = f_1(x') \implies x = x' \end{aligned}$$

which, by algebraic manipulation, implies that $x = x'$.

Surjective We are required to show that if f_1 and f_2 are surjective, then the composition of f_1 and f_2 is surjective as well.

$$\forall i \in \mathcal{A}, \exists j \in \mathcal{A} \colon f_1(j) = i \qquad \bigwedge \tag{A.40}$$

$$\forall i \in \mathcal{A}, \exists j \in \mathcal{A} \colon f_2(j) = i \tag{A.41}$$

$$\implies$$

$$\forall i \in \mathcal{A}, \exists j \in \mathcal{A} \colon f_2(f_1(j)) = i. \tag{A.42}$$

Let x'' be an instance of i in Equation A.42; use this value for i in Equation A.41. Let x' be an instance of j in Equation A.41; use this value for i in Equation A.40. Finally, let x be an instance of j in Equation A.40; use this value for j in Equation A.42. The implication follows:

$$\begin{aligned} f_1(x) = x' \wedge f_2(x') = x'' &\implies f_2(f_1(x)) = x'' \\ &\implies f_2(x') = x'' \\ &\implies x'' = x''. \end{aligned}$$

Permutes We are required to show

$$\forall i \in \mathcal{A} \colon S(i) = S'(f_1(i)) \qquad \bigwedge \tag{A.43}$$

$$\forall i \in \mathcal{A} \colon S'(i) = S''(f_2(i)) \tag{A.44}$$

$$\implies$$

$$\forall i \in \mathcal{A} \colon S(i) = S''(f_2(f_1(i))). \tag{A.45}$$

Let x be the instantiation of i in Equation A.45. Use x for i in Equation A.43 and

$f_1(x)$ in Equation A.44. The implication follows:

$$\begin{aligned} S(x) = S'(f_1(x)) \wedge S'(f_1(x)) = S''(f_2(f_1(x))) &\implies S(x) = S''(f_2(f_1(x))) \\ &\implies S(x) = S'(f_1(x)) \\ &\implies S(x) = S(x). \end{aligned}$$

■

Bibliography

[1] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag. 1.2.1.2, 9.1

[2] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. doi:10.1109/32.588521. 1.2.1.2, 9.2.1

[3] K. Sacha. Model-based implementation of real-time systems. In *International Conference on Computer Safety, Reliability, and Security*, pages 332–345, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-87698-4_28. 1.2.1.2

[4] J. Blech and A. Poetzsch-Heffter. A certifying code generation phase. *Electronic Notes in Theoretical Computer Science*, 190(4):65–82, 2007. doi:10.1016/j.entcs.2007.09.008. 1.2.1.2

[5] E. Denney and B. Fischer. Extending source code generators for evidence-based software certification. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 138–145, Washington, DC, USA, 2006. IEEE Computer Society. doi:10.1109/ISoLA.2006.76. 1.2.1.2

[6] K. Kennedy. *Caps: concurrent automatic programming system.* PhD thesis, Clemson University, Clemson, SC, USA, 2008. 1.2.1.2, 9.3

[7] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K.R. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for*

Technology Transfer, 7(3):212–232, 2005. doi:10.1007/s10009-004-0167-4. 1.2.1.2, 9.3.3

[8] J. Armstrong. *Programming Erlang—Software for a Concurrent World*. Cambridge University Press, New York, NY, USA, first edition, July 2007. doi:10.1017/S0956796809007163. 1.2.1.2, 9.3

[9] S. Demri, F. Laroussinie, and P. Schnoebelen. A parametric analysis of the state-explosion problem in model checking. *Journal of Computer and System Sciences*, 72(4):547–575, 2006. doi:10.1016/j.jcss.2005.11.003. 1.4.1

[10] J. Woodcock, P. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):1–36, 2009. doi:10.1145/1592434.1592436. 1.4.1, 9.3.4

[11] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Symposium on Principles of distributed computing*, pages 137–151, New York, NY, USA, 1987. ACM. doi:10.1145/41840.41852. 1.5

[12] I. Keidar, R. Khazan, N. Lynch, and A. Shvartsman. An inheritance-based technique for building simulation proofs incrementally. *Transactions on Software Engineering and Methodology*, 11(1):63–91, 2002. doi:10.1145/504087.504090. 1.5

[13] B. Jonsson. Compositional specification and verification of distributed systems. *Transactions on Programming Languages and Systems*, 16(2):259–303, 1994. doi:10.1145/174662.174665. 1.5

[14] B. Möller. Algebraic calculation of graph and sorting algorithms. In Dines Bjørner, Manfred Broy, and Igor Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 394–413. Springer Berlin/Heidelberg, 1993. doi:10.1007/BFb0039722. 1.5

[15] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. 2.1

[16] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. 2.1.2

[17] K.M. Chandy, B. Go, S. Mitra, C. Pilotto, and J. White. Verification of distributed systems with local-global predicates. *Formal Aspects of Computing*, pages 1–31, 2010. doi:10.1007/s00165-010-0150-7. 2.3.2, 3, 14

[18] J. Bard. *Morphogenesis : the cellular and molecular processes of developmental anatomy*, volume 23 of *Developmental and cell biology series*. Cambridge, 1990. 2.4

[19] L. Wolpert. *Principles of development*, volume 23. Oxford University Press, third edition, 2007. 2.4

[20] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, USA, 1999. 2.4

[21] J. Kennedy and R. Eberhart with Y. Shi. *Swarm Intelligence*. Morgan Kaufmann, 2001. 2.4

[22] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, 2000. doi:10.1145/332833.332842. 2.4

[23] A. Kondacs. Biologically-inspired self-assembly of two-dimensional shapes using global-to-local compilation. In *International Joint Conference on Artificial Intelligence*, pages 633–638, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers, Inc. 2.4

[24] E. Klavins, R. Ghrist, and D. Lipsky. Graph grammars for self assembling robotic systems. In *International Conference on Robotics and Automation*, volume 5, pages 5293–5300. IEEE, April 2004. doi:10.1109/ROBOT.2004.1302558. 2.4

[25] E. Klavins. Programmable self-assembly. *Control Systems Magazine, IEEE*, 27(4):43–56, August 2007. doi:10.1109/MCS.2007.384126. 2.4

[26] D. Coore. *The Growing Point Language*. Lambert Academic Publishing, 2010. 2.4

[27] R. Nagpal. *Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics*. PhD thesis, Massachusetts Institute of Technology, 2001. 2.4

[28] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *Intelligent Systems, IEEE*, 21(2):10–19, March 2006. doi:10.1109/MIS.2006.29. 2.4

[29] D. Yamins, S. Waydo, and N. Khaneja. Group control and kernels: the 1-d equigrouping problem. *IEEE Conference on Decision and Control*, 3:2460–2466, December 2004. 2.4

[30] D. Yamins. Towards a theory of "local to global" in distributed multi-agent systems (I). In *International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 183–190, New York, NY, USA, 2005. ACM. doi:10.1145/1082473.1082501. 2.4

[31] D. Yamins. Towards a theory of "local to global" in distributed multi-agent systems (II). In *International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 191–198, New York, NY, USA, 2005. ACM. doi:10.1145/1082473.1082502. 2.4

[32] D. Yamins. The emergence of global properties from local interactions: static properties and one-dimensional patterns. In *International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1122–1124, New York, NY, USA, 2006. ACM. doi:10.1145/1160633.1160837. 2.4

[33] D. Yamins. *A theory of local-to-global algorithms for one-dimensional spatial multi-agent systems*. PhD thesis, Harvard University, Cambridge, MA, USA, 2008. 2.4

[34] N. Minsky. Regularity-based trust in cyberspace. In Paddy Nixon and Sotirios Terzis, editors, *Trust Management*, volume 2692 of *Lecture Notes in Computer Science*, pages 1071–1072. Springer Berlin/Heidelberg, 2003. doi:10.1007/3-540-44875-6_2. 2.4

[35] W. Zhang, C. Serban, and N. Minsky. *Establishing Global Properties of Multi-Agent Systems Via Local Laws*, volume 4389/2007 of *Lecture Notes in Computer Science*, pages 170–183. Springer Berlin/Heidelberg, 2007. doi:10.1007/978-3-540-71103-2. 2.4

[36] X. Ao and N. Minsky. Flexible regulation of distributed coalitions. In Einar Snekkenes and Dieter Gollmann, editors, *European Symposium on Research in Computer Security*, volume 2808/2003 of *Lecture Notes in Computer Science*, pages 39–60. Springer Berlin/Heidelberg, 2003. doi:10.1007/978-3-540-39650-5_3. 2.4

[37] C. Serban, W. Zhang, and N. Minsky. A decentralized mechanism for application level monitoring of distributed systems. *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 1–10, November 2009. doi:10.4108/ICST.COLLABORATECOM2009.8336. 2.4

[38] N. Minsky. Reducing spam via trustworthy self regulation by email senders. In *MIT Span Conference*, 2010. 2.4

[39] C. Serban, Y. Chen, W. Zhang, and N. Minsky. The concept of decentralized and secure electronic marketplace. *Electronic Commerce Research*, 8:79–101, 2008. doi:10.1007/s10660-008-9014-0. 2.4

[40] K. Marzullo and L. Sabel. Efficient detection of a class of stable properties. *Distributed Computing*, 8(2):81–91, 1994. doi:10.1007/BF02280830. 2.4

[41] R. Atreya, N. Mittal, A. Kshemkalyani, V. Garg, and M. Singhal. Efficient detection of a locally stable predicate in a distributed system. *Journal of Parallel and Distributed Computing*, 67(4):369–385, 2007. doi:10.1016/j.jpdc.2006.12.004. 2.4

[42] P. Wegner, F. Arbab, D. Goldin, P. McBurney, M. Luck, and D. Robertson. The role of agent interaction in models of computing: Panelist reviews. *Electronic Notes in Theoretical Computer Science*, 141(5):181–198, 2005. Workshop on the Foundations of Interactive Computation. doi:10.1016/j.entcs.2005.05.022. 2.4

[43] T. Nipkow and L. Paulson. Proof pearl: Defining functions over finite sets. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 385–396. Springer Berlin / Heidelberg, 2005. doi:10.1007/11541868_25. 3, 3.2

[44] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999. doi:10.1017/S0956796899003500. 3, 3.2

[45] R. Dean. *Elements of Abstract Algebra*. Wiley, New York, second edition, 1966. 3

[46] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269, April 1967. 12

[47] L. Huang. Advanced dynamic programming in semiring and hypergraph frameworks. In *Advanced Dynamic Programming in Computational Linguistics: Theory, Algorithms and Applications—Tutorial notes*, pages 1–18, Manchester, UK, August 2008. Coling 2008 Organizing Committee. 12, 4.6

[48] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962. doi:10.1145/321105.321107. 4.6

[49] M. Yoeli. A note on a generalization of boolean matrix theory. *The American Mathematical Monthly*, 68(6):552–557, June–July 1961. 4.6

[50] G. Penn. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science*, 354(1):72–81, 2006. doi:10.1016/j.tcs.2005.11.008. 4.6

[51] M. Gondran and M. Minoux. *Graphs, dioids and semirings*, volume 41 of *Operations Research/Computer Science Interfaces Series*. Springer, New York, 2008. New models and algorithms. 4.6

[52] C. Huang and C. Lengauer. An incremental mechanical development of systolic solutions to the algebraic path problem. *Acta Informatica*, 27(2):97–124, 1989. doi:10.1007/BF00265150. 4.6

[53] G. Chen, B. Wang, and C. Lu. On the parallel computation of the algebraic path problem. *IEEE Transactions on Parallel and Distributed Systems*, 3:251–256, 1992. doi:10.1109/71.127265. 4.6

[54] G. Rote. A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion). *Computing*, 34:191–219, 1985. doi:10.1007/BF02253318. 4.6

[55] C. Djamégni, P. Quinton, S. Rajopadhye, and T. Risset. Derivation of systolic algorithms for the algebraic path problem by recurrence transformations. *Parallel Computing*, 26(11):1429–1445, 2000. doi:10.1016/S0167-8191(00)00039-9. 4.6

[56] H. Tsai, S. Horng, S. Tsai, T. Kao, and S. Lee. Solving an algebraic path problem and some related graph problems on a hyper-bus broadcast network. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1226–1235, December 1997. doi:10.1109/71.640014. 4.6

[57] E. Fink. A survey of sequential and systolic algorithms for the algebraic path problem. Technical Report CS-92-37, University of Waterloo, 1992. 4.6

[58] D. Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76, 1977. doi:10.1016/0304-3975(77)90056-1. 4.6

[59] R. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, 1981. doi:10.1145/322261.322272. 4.6

[60] M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002. 4.6

[61] H. Hofstee, A. Martin, and J. van de Snepscheut. Distributed sorting. *Science of Computer Programming*, 15:119–133, December 1990. doi:10.1016/0167-6423(90)90081-N. 5

[62] D. Knuth. *The art of computer programming, volume 3: sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, second edition, 1998. 5

[63] M. Mehyar. *Distributed averaging and efficient file sharing on peer-to-peer networks*. PhD thesis, California Institute of Technology, 2007. 6.1

[64] J. Fax and R. Murray. Information flow and cooperative control of vehicle formations. *IEEE Transactions on Automatic Control*, 49(9):1465–1476, September 2004. doi:10.1109/TAC.2004.834433. 6.1

[65] R. Olfati-Saber and J. Shamma. Consensus filters for sensor networks and distributed sensor fusion. In *IEEE Conference on Decision and Control*, pages 6698–6703, December 2005. doi:10.1109/CDC.2005.1583238. 6.1

[66] S. Kar, S. Aldosari, and J. Moura. Topology for distributed inference on graphs. *IEEE Transactions on Signal Processing*, 56(6):2609–2613, June 2008. doi:10.1109/TSP.2008.923536. 6.1

[67] C. Xu and F. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. 6.1

[68] M. Mehyar, D. Spanos, J. Pongsajapan, S. Low, and R. Murray. Asynchronous distributed averaging on communication networks. *IEEE/ACM Transactions on Networking*, 15(3):512–520, June 2007. doi:10.1109/TNET.2007.893226. 6.1

[69] R. Olfati-Saber and R. Murray. Consensus problems in networks of agents with switching topology and time-delays. *IEEE Transactions on Automatic Control*, 49(9):1520–1533, September 2004. doi:10.1109/TAC.2004.834113. 6.1

[70] S. Kar and J. Moura. Distributed consensus algorithms in sensor networks with imperfect communication: Link failures and channel noise. *IEEE Transactions on Signal Processing*, 57(1):355 –369, January 2009. doi:10.1109/TSP.2008.2007111. 6.1

[71] L. Xiao, S. Boyd, and S. Kim. Distributed average consensus with least-mean-square deviation. *Journal of Parallel and Distributed Computing*, 67(1):33–46, 2007. doi:10.1016/j.jpdc.2006.08.010. 6.1

[72] C. Pilotto, K.M. Chandy, and J. White. Consensus on asynchronous communication networks in presence of external input. In *49th IEEE Conference on Decision and Control*, pages 3838–3844, December 2010. doi:10.1109/CDC.2010.5717134. 6.1

[73] Z. Minghui and S. Martnez. Discrete-time dynamic average consensus. *Automatica*, 46(2):322 – 329, 2010. doi:10.1016/j.automatica.2009.10.021. 7

[74] D. Spanos, R. Olfati-Saber, and R. Murray. Dynamic consensus on mobile networks. In *World Congress of the International Federation of Automatic Control*, 2005. 7

[75] S. Maharaj and J. Bicarregui. On the verification of vdm specification and refinement with pvs. In *International conference on Automated software engineering*, page 280, Washington, DC, USA, 1997. IEEE Computer Society. 9.1

[76] E. de Jong, J. van de Pol, and J. Hooman. Refinement in requirements specification and analysis: A case study. In *International Conference and Workshop on the Engineering of Computer Based Systems*, pages 290–298, Washington, DC, USA, April 2000. IEEE Computer Society. doi:10.1109/ECBS.2000.839888. 9.1

[77] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. 9.1.5

[78] G. Holzmann, R. Joshi, and A. Groce. Swarm verification. In *International Conference on Automated Software Engineering*, pages 1–6, Washington, DC, USA, 2008. IEEE Computer Society. doi:10.1109/ASE.2008.9. 9.2

[79] G. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *International SPIN Workshop*, 1997. 9.2.4

[80] M. Casadei and M. Viroli. An experience on probabilistic model checking and stochastic simulation to design self-organizing systems. In *Congress on Evolutionary Computation*, pages 1538–1545, Piscataway, NJ, USA, 2009. IEEE Press. doi:10.1109/CEC.2009.4983125. 9.2.6

[81] M. Casadei and M. Viroli. Using probabilistic model checking and simulation for designing self-organizing systems. In *Symposium on Applied Computing*, pages 2103–2104, New York, NY, USA, 2009. ACM. doi:10.1145/1529282.1529747. 9.2.6

[82] M. Casadei, M. Viroli, and L. Gardelli. On the collective sort problem for distributed tuple spaces. *Science of Computer Programming*, 74(9):702–722, 2009. doi:10.1016/j.scico.2008.09.018. 9.2.6

[83] M. Casadei, L. Gardelli, and M. Viroli. Simulating emergent properties of coordination in maude: the collective sort case. *Electronic Notes in Theoretical Computer Science*, 175(2):59–80, 2007. doi:10.1016/j.entcs.2007.05.022. 9.2.6

[84] J.O. Blech and A. Poetzsch-Heffter. A certifying code generation phase. *Electronic Notes in Theoretical Computer Science*, 190(4):65–82, 2007. doi:10.1016/j.entcs.2007.09.008. 9.3

[85] E. Denney and B. Fischer. Extending source code generators for evidence-based software certification. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 138–145, Washington, DC, USA, November 2006. IEEE Computer Society. doi:10.1109/ISoLA.2006.76. 9.3

[86] K. Sacha. Model-based implementation of real-time systems. In *International Conference on Computer Safety, Reliability, and Security*, volume 5219 of *Lecture Notes in Computer Science*, pages 332–345. Springer-Verlag, September 2008. doi:10.1007/978-3-540-87698-4_28. 9.3

[87] J. Armstrong. A history of Erlang. In *ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM. doi:10.1145/1238844.1238850. 9.3

[88] C. Hewitt. scriptJ(TM) extension of Java(R): discretionary, adaptive concurrency for privacy-friendly, client-cloud computing. *Computing Research Repository*, abs/1008.2748, 2010. 9.3

[89] R. Khazan. Group membership: a novel approach and the first single-round algorithm. In *Symposium on Principles of Distributed Computing*, pages 347–356, New York, NY, USA, July 2004. ACM. doi:10.1145/1011767.1011819. 9.3.1

[90] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *International Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 292–312, Berlin, Heidelberg, November 1992. Springer-Verlag. doi:10.1007/3-540-56188-9_20. 9.3.1

[91] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, 1996. doi:10.1109/32.481515. 9.3.1

[92] Q. Huang, C. Julien, and G. Roman. Relying on safe distance to achieve strong partitionable group membership in ad hoc networks. *IEEE Transactions on Mobile Computing*, 3(2):192–205, 2004. doi:10.1109/TMC.2004.14. 9.3.1

[93] A. Jain and R. Shyamasundar. Failure detection and membership management in grid environments. In *International Workshop on Grid Computing*, pages 44–52, Washington, DC, USA, November 2004. IEEE Computer Society. doi:10.1109/GRID.2004.30. 9.3.1

[94] M. Barnett, K.R. Leino, and W. Schulte. The spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin / Heidelberg, 2005. doi:10.1007/978-3-540-30569-9_3. 9.3.3

[95] M. Barnett, B. Chang, R. DeLine, B. Jacobs, and R.K. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin/Heidelberg, 2006. doi:10.1007/11804192_17. 10.1.1

[96] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, Inc., Orlando, FL, USA, 1983. 10.1.2

[97] A. Svolos, C. Konstantopoulos, and C. Kaklamanis. Efficient binary morphological algorithms on a massively parallel processor. *Parallel and Distributed Processing Symposium, International*, 0:281, 2000. doi:10.1109/IPDPS.2000.845997. 10.1.2

[98] M. Tarek, C. Boutrous-Saab, and S. Rampacek. Verifying correctness of web services choreography. *Web Services, European Conference on*, 0:306–318, 2006. doi:10.1109/ECOWS.2006.38. 10.1.3

[99] H. Zhu and B. Yu. Algebraic specification of web services. *Quality Software, International Conference on*, 0:457–464, 2010. doi:10.1109/QSIC.2010.47. 10.1.3

[100] H. Herrlich. *Axiom of Choice*, volume 1876 of *Lecture Notes in Mathematics*. Springer Berlin / Heidelberg, 2006. doi:10.1007/11601562. A.2.1