Abstract State Machines Capture Parallel Algorithms

ANDREAS BLASS
University of Michigan
and
YURI GUREVICH
Microsoft Research

We give an axiomatic description of parallel, synchronous algorithms. Our main result is that every such algorithm can be simulated, step for step, by an abstract state machine with a background that provides for multisets.

Categories and Subject Descriptors: F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Parallel algorithm, abstract state machine, ASM thesis, postulates for parallel computation

1. INTRODUCTION

The Abstract State Machine Thesis [Gurevich 1995] asserts that every algorithm is behaviorally equivalent to an abstract state machine and in particular is simulated step for step by that machine. There is considerable empirical evidence for this thesis [ASM Web]. In Gurevich [2000], the thesis was proved for sequential algorithms. The purpose of this article is to extend the ideas and results of Gurevich [2000] to parallel algorithms.

The algorithms we consider have computations divided logically into a sequence of discrete steps. Within each step, many parallel subcomputations may take place, but they finish before the next step begins. We do not consider the

The work of the first author was partially supported by NSF grant DMS-0070723 and by a grant from Microsoft Research. Most of this paper was written during his visits to Microsoft Research.
Authors' addresses: A. Blass, Mathematics Department, University of Michigan, Ann Arbor, MI 48109-1109; email: ablass@umich.edu; Y. Gurevich, Microsoft Research, One Microsoft Way, Redmond, WA 98052; email: gurevich@microsoft.com

more general notion of distributed computation, where many agents proceed asynchronously, each at its own speed, and where communications between agents may provide the only logical ordering between their actions. In addition, we assume that parallelism is the only way for the amount of computation within a single step to be unbounded, that is, to increase with the input or the state rather than being fixed by the algorithm. Another way to say this is that we insist on a fixed bound for the amount of sequentiality in any one step (more precise formulations will be given later).

We shall give a general, axiomatic description of such parallel algorithms, analogous to the description of sequential algorithms in Gurevich [2000]. We hope to make it plausible that our description is broad enough to cover all algorithms of the sort described in the preceding paragraph. Then, we shall prove that every such algorithm can be simulated step-for-step by an abstract state machine (ASM) [Gurevich 1995, 1997] operating with a suitable background in the sense of Blass and Gurevich [2000]. Since abstract state machines are conceptually quite simple, and since the background we need can be kept simple as well, our main result provides a significant reduction of the general notion of parallel algorithm. In effect, it provides a simple normal form for such algorithms. By enriching the background somewhat, we can make individual algorithms simpler and more natural, at the cost of some loss of simplicity in the general notion.

This article is organized as follows. Section 2 reviews the axioms for sequential algorithms introduced in Gurevich [2000], checking which of them remain correct for parallel computation and which must be modified. Although we briefly review the material from Gurevich [2000] that we use in the present paper, a detailed understanding of Sections 2 and 10 presupposes knowledge of Gurevich [2000]. Section 3 is devoted to a special case of parallel computation, where there is, within any one step, no interaction between the various processes. Some of the issues distinguishing parallel from sequential computation are most easily understood in this context. In Sections 4 and 5 we describe what must be added to this simple context in order to describe general parallel algorithms. In particular, we go beyond Section 3 by allowing communication and interaction between processes, but not in a way that would introduce unbounded sequentiality. The framework developed here covers the partial updates described and studied in Gurevich et al. [2001] and Gurevich and Tillman [2001] and used in the specification language AsmL [Foundations of Software Engineering Group]. Section 6 summarizes material on backgrounds from Blass and Gurevich [2000] and Gurevich et al. [2001], indicating its relevance to the matters discussed in the preceding sections. We assemble all this material in Section 7 to describe parallel computation in general. To support the claim that our description is general, we show in Section 8 how several popular models of parallel computation fit the description. (We thank two of the referees for suggesting that this material be added to the earlier version of the article.) In Section 9, we review the definition of abstract state machines in the context of a background containing multisets. We also establish a normal form theorem for these abstract state machines. Part of the parallelism and thus part of the power of this model comes from allowing comprehension terms for multisets,

analogous to the comprehension terms for sets used in Blass et al. [1999]. In Section 10, we prove our main result, showing how the general parallel algorithms described in Section 7 can be simulated by abstract state machines. In particular, this shows that the partial updates of Foundations of Software Engineering Group, Gurevich et al. [2001] and Gurevich and Tillman [2001] can be eliminated in favor of ordinary updates in the presence of the multiset background. In Section 11, we extend some of our results to unbounded sequentiality by using abstract state machines with submachines. The subsequent sections contain additional remarks about parallel algorithms in general and about possible variations in the axioms. The study of one of these variations leads to an apparently new complexity class and a complete problem for it in Section 12.

Terminology 1.1. By "parallelism" we generally mean unbounded parallelism. That is, the number of parallel processes is not subject to an *a priori* bound given by the program alone but rather depends on the state. Bounded parallelism, exemplified by the "`do in parallel`" rules of ASMs, is really sequential computation and was already covered by the analysis in Gurevich [2000].

Terminology 1.2. The word "sequential" has two different meanings, both of which are somewhat relevant to this article, so we must be careful to distinguish them.

The phrase "sequential time" refers to what was described above as synchrony between the various processes. It means that in any run of an algorithm, there is an initial state, followed by a second state, followed by a third, and so on, the progression of states being determined by the algorithm. This concept is to be contrasted with distributed algorithms, where there may be no natural global clock, and where an attempt to speak of the second state, the third, and so on may lead to states that depend, in general, not only on the algorithm but also on the order in which different agents happen to execute their tasks.

The phrase "sequential algorithm" means an algorithm which not only operates in sequential time but also does not involve (unbounded) parallelism. There is a uniform bound, depending only on the algorithm and not on the state, for how much reading and writing can be done at any one step. These are the algorithms characterized in Gurevich [2000].

Thus, the algorithms considered in the present paper are sequential time algorithms but not necessarily sequential algorithms.

Terminology 1.3. Throughout this article, we use "bounded" to mean that a bound depends only on the algorithm, not on its input or its state. Thus, for example, the algorithms considered in this paper have, in each step, bounded sequentiality but not (necessarily) bounded parallelism.

Notice that this meaning of "bounded" disagrees with the set-theoretic terminology "bounded quantification," which means that the quantified variable is restricted to range over a (specified) set. When the states of an algorithm include a set-theoretic structure, as for example in Blass et al. [1999], it often happens that the sets used by the algorithm grow with the input.

In such cases, bounded quantifiers in the set-theoretic sense may fail to be bounded in our sense; that is, the ranges of the quantified variables may not be bounded.

2. SEQUENTIAL TIME AND ABSTRACT STATES

The treatment of sequential algorithms in Gurevich [2000] was based on three postulates:

(1) **Sequential Time:** An algorithm A is associated with a set $\mathcal{S}(A)$ of states, a subset $\mathcal{I}(A) \subseteq \mathcal{S}(A)$ of initial states, and a map $\tau_A : \mathcal{S}(A) \to \mathcal{S}(A)$ called the one-step transformation.

(2) **Abstract State:** All states of A are first-order structures with the same finite vocabulary, which we call the vocabulary of A. τ_A does not change the base set of a state. Both $\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphisms. Any isomorphism from a state X to a state Y is also an isomorphism from $\tau_A(X)$ to $\tau_A(Y)$.

(3) **Bounded Exploration:** There is a finite set T of terms in the vocabulary of A such that, whenever two states X and Y coincide on T, then $\Delta(A, X) = \Delta(A, Y)$.

In the last postulate, $\Delta(A, X)$, the *update set* produced by algorithm A in state X, is defined as the set of all triples $(f, \bar{a}, b)$ where f is a function symbol, $\bar{a}$ is a tuple of elements of X of the right length to serve as the argument tuple of f, b is the value of f at $\bar{a}$ in $\tau_A(X)$, but b is not the value of f at $\bar{a}$ in X. Thus $\Delta(A, X)$ can be regarded as all the information of the form $f(\bar{a}) = b$ that is not true in X but becomes true when τ_A is applied to X. It represents the changes in X produced by the algorithm A.

Recall our convention that "bounded" means that the bound depends only on the algorithm, not on the state. The name of the Bounded Exploration Postulate refers to the fact that what the algorithm does at any step depends only on the values of a fixed, finite set T of terms, which in turn depends only on the algorithm. In particular, the number of elements of the state that are involved in one computation step is bounded.

By a *pure run* of an algorithm, we mean a finite or infinite sequence of states in which the first member is an initial state and each of the other members is obtained by applying τ_A to the immediately preceding member. (The word "pure" refers to the fact that there are no external influences on the sequence of states, for example no users typing input.)

We shall need a modification of the notion of sequential algorithm, which is also of independent interest, namely a sequential algorithm with output. This is a process that works like a sequential algorithm but may also, at each of its steps, send some information to the outside world. For example, the algorithm may write a character string on a printer, or it may send data to some other algorithm. The information sent is, like every object involved in an algorithm's computation, a member of its state.

One way of modeling this situation would be to have a dynamic function symbol `output` to which the algorithm assigns values that are the objects to

be sent; it would be assumed that the outside world somehow monitors this dynamic symbol and extracts the appropriate values. It will be useful for our purposes to regard sending information as a separate operation from updating the state. In particular, we allow for the possibility of several elements being sent at the same step. In fact, we shall also need the possibility of sending the same element several times during one step, so in general we have not a set but a multiset of outputs at each step. (See Section 4 below for a discussion of multisets; for now, it will do no harm to think of sets instead.) We formalize this picture as follows.

Definition 2.1. A *sequential algorithm with output* is an algorithm satisfying the Sequential Time, Abstract State, and Bounded Sequentiality Postulates above with the following modifications.

—Add to the Sequential Time Postulate that there is a function σ_A assigning to each state X a multiset $\sigma_A(X)$ of members of the base set of X, called the multiset of outputs from X in algorithm A.
—Add to the Abstract State Postulate that σ_A respects isomorphisms in the sense that, if $i : X \to Y$ is an isomorphism between two states, then $i[\sigma_A(X)] = \sigma_A(Y)$.
—Add to the Bounded Sequentiality Postulate that, if two states X and Y coincide on T, then $\sigma_A(X) = \sigma_A(Y)$.

The proof of the sequential thesis in Gurevich [2000] can be adapted to show that any sequential algorithm with output is equivalent to a sequential abstract state machine with output. (See the proof of Theorem 10.1 for a discussion of this adaptation.) Here sequential ASMs with output are defined just like ordinary sequential ASMs except that the definition of rules has an extra clause, introducing rules of the form "$\texttt{Output}(t)$" where t is any term. The semantics of such a rule is that the value of t (in the current state) is put into the output multiset of the algorithm at this state.

In some situations, it may be convenient to have several sorts of output, for example writing on several printers or sending data to several destinations. This can be incorporated in the preceding picture by tagging each output element with a label indicating what sort of output it is. Alternatively, if the number of output functions depends only on the algorithm and not on the state, then we could extend the notion of sequential algorithm with output by having several output functions $\sigma_{A,l}$, and we could extend the notion of ASM with output by having several sorts of output rules, $\texttt{Output}_l(t)$. All the preceding comments apply as well in this extended context.

Remark 2.2. In the present article as well as in Gurevich [2000], the notion of initial state, though important for accurately modeling general notions of algorithms, plays no essential role in the technical development. We could simplify the postulates by omitting all mention of initial states. This simplification could be obtained equivalently by assuming that all states are initial. To use a smaller set of initial states merely means to confine one's attention to a subset of the runs.

For the first two axioms, the Sequential Time Postulate and the Abstract State Postulate, the justifications given in Gurevich [2000, Sections 3.1 to 3.3 and 4] do not depend on the fact that the algorithm is sequential. So these two axioms are equally valid for the parallel algorithms that are our concern in the present paper. We adopt them as part of our general description of parallel algorithms.

The Bounded Exploration Postulate, on the other hand, specifically describes sequential algorithms. It is false for most parallel algorithms. For example, consider an algorithm that takes as input a (simple, undirected) graph and transforms it to its complement, deleting all the edges present in the input and adding edges between those pairs of distinct vertices that were not adjacent in the input.

```
do forall x in Vertex
   do forall y in Vertex
      if x≠y then Adjacent(x,y):=¬Adjacent(x,y) endif
   enddo
enddo
```

It is certainly not true for this algorithm that agreement between two states on a fixed finite set entails agreement of the update sets. The update sets depend on the entire graphs, which may be very different despite any fixed finite amount of agreement. So the Bounded Exploration Postulate fails for such an algorithm.

It fails for an additional reason also. There are very few ground terms in the vocabulary of graphs; they all denote `true`, `false`, or `undef`. So the hypothesis in the Bounded Exploration Postulate, that X and Y agree over T, does not even guarantee any agreement at all between the graphs; it only guarantees that `true`, `false`, and `undef` agree.

Clearly, to describe parallel algorithms, we must drop the Bounded Exploration Postulate and assume, in its place, an axiom or axioms specifically designed for parallelism. The development of such axioms will be the goal of the next few sections.

One aspect of the Bounded Exploration Postulate, however, still makes sense in the parallel context. The transition from any state X to the next state $\tau_A(X)$ still consists of certain changes or updates, collected in the set $\Delta(A, X)$ defined above. It will be the task of our new axioms to explain where this set of updates comes from. The explanation will have to be new, but what is to be explained is still the update set $\Delta(A, X)$.

We adopt from Gurevich [2000] the following very strong notion of (behavioral) equivalence of algorithms.

Definition 2.3. Two algorithms are equivalent if they have the same states, the same initial states, and the same one-step transformation function.

We shall obtain, for every parallel algorithm (in the sense defined by our postulates) an abstract state machine that is equivalent to it in the sense defined here. It will follow of course that they are also equivalent in any of the weaker senses of equivalence that have been considered in the literature. By using a very strong notion of equivalence, we obtain a correspondingly strong theorem.

The following corollary is a trivial consequence of the definitions, but we mention it explicitly for emphasis.

COROLLARY 2.4. *Equivalent algorithms have the same pure runs.*

Remark 2.5. It is reasonable to ask about impure runs—runs that involve some intervention by the external world. The situation is quite simple as long as we assume that the environment intervenes only between steps of the algorithm. That is, we suppose that a run is a sequence of states in which the first state is initial and each subsequent state comes from its immediate predecessor either by the algorithm's transition function (as in a pure run) or by an action of the environment. Then, if two algorithms are equivalent, every run of either algorithm is also a run of the other, involving exactly the same interventions of the environment (at exactly the same steps).

The situation is much less clear if the environment can intervene during a step of the algorithm. Indeed, it is not even clear in this case what it would mean for runs of two equivalent algorithms to involve the same interventions of the environment, since equivalence of algorithms does not require any similarity between the internal structure (if any) of the algorithms' steps.

There is, however, a reasonably clear case, namely the one in which the environment assigns values to certain functions, called external functions, and these values do not change during any one step of the algorithm. In this case, it does no harm to pretend that the values of external functions were assigned at the beginning of each step of the algorithm, and this means in effect that the environment's actions occur between the algorithm's steps. So we are back in the easy situation discussed above.

Notice that this sort of action by the environment covers non-determinism, for we can regard non-deterministic choices as being given by external functions. A slightly more involved argument justifies the importing of new elements from the reserve; see Gurevich [1997].

More general interventions of an environment within a step, for example replies to an algorithm's calls of external routines, appear to be useful in modeling and therefore should be considered, but we do not deal with them in this paper. Our goal is simply to prove equivalence, as defined above, between arbitrary parallel algorithms and abstract state machines. Accordingly, we shall not treat environments except in the present remark and one appearance of non-determinism in Section 8.

3. A SIMPLIFIED PICTURE

In this section, we consider an oversimplified view of parallel algorithms, designed to remain close to the ideas of Gurevich [2000] while exhibiting some of the new phenomena involved with parallelism.

A key observation for our treatment is that a parallel computation consists of a number of processes running (not surprisingly) in parallel, and that, if we analyze the computation far enough, then each of these processes satisfies Bounded Exploration even though the whole computation does not.

For example, in the graph-complementing algorithm, each pair of distinct vertices (x, y) has an associated process that makes this pair adjacent in $\tau_A(X)$ if and only if they were not adjacent in X. That is, the process associated with (x, y) is responsible for one update in $\Delta(A, X)$, namely $(\texttt{Adjacent}, (x, y), \texttt{true})$ or $(\texttt{Adjacent}, (x, y), \texttt{false})$. And any single such process satisfies Bounded Exploration in a suitable sense.

Terminology 3.1. We have been using the word "process" in a rather unusual and specialized way, namely referring only to subprocesses so small that no (unbounded) parallelism remains. Processes in this sense are thus subject to the Bounded Exploration Postulate.

Our example can be viewed as having a tree structure (as its indentation was designed to suggest). The overall computation first splits into subcomputations indexed by a single vertex x. The subcomputation associated with x executes the following algorithm in which `me` is to be interpreted as denoting x.

```
do forall y in Vertex
   if me≠y then Adjacent(me,y):=¬Adjacent(me,y) endif
enddo.
```

This intermediate level subcomputation still includes parallelism, looking at all vertices y of the graph, so it does not satisfy Bounded Exploration. In standard terminology, "process" can refer to subcomputations like this and, typically, to much larger ones, usually not satisfying Bounded Exploration. To avoid conflict with this well established terminology, we shall henceforth use the word *proclet* to mean a process in the specialized sense above, satisfying Bounded Exploration in a suitable sense.[1]

We must be careful about the "suitable sense" in which proclets satisfy Bounded Exploration. Remember that the vocabulary of graphs is so poor that no ground term denotes a vertex. The bounded-exploration witness T for the proclet associated with (x, y) should consist of (names for) x, y, `true`, and `false`. (See the discussion in Gurevich [2000, Section 5.4].) But the vocabulary doesn't have names for x and y.

The solution to this difficulty begins with a closer look at the notion of state, as explained in Gurevich [2000, Section 3.3.1]. The crucial point is that the state should contain everything that is relevant to the computation (at the chosen level of abstraction). Applied to parallel algorithms, this point means that *the proclets should themselves be (represented by) elements of the state*. In our graph-complementing example, this means that the pairs of vertices (x, y) (or some equivalent system of labels for the proclets) should be elements of the state, along with the vertices themselves. In addition, the vocabulary should include names for static functions `first` and `second` mapping each (x, y) to x and to y, respectively.

To complete the solution to the "lack of sufficient ground terms" problem, we stipulate that a proclet should know which proclet it is. More precisely, this

[1]Etymology of "proclet": The suffix "let" means "small" as in "piglet" or "wavelet." Since "processlet" seems awkward, we shortened it to "proclet."

means that a proclet p operates on a structure that is like the actual state of the computation except that its vocabulary has an additional, nullary, static symbol `me` interpreted as denoting p. For example, the algorithm executed by any single proclet in our graph-complementing example is

```
if first(me)≠second(me) then
   Adjacent(first(me), second(me)):=                    (A1)
   ¬Adjacent(first(me), second(me)) endif.
```

Terminology 3.2. The state in which a proclet executes its computation will be called the *local state* of that proclet. When necessary for contrast, we use *global state* to mean the state of the entire computation. Thus, in the situation under discussion, the local state of a proclet p differs from the global state only by having the additional symbol `me` denoting p. In later sections, when we discuss more general parallel algorithms than in the present section, there will be additional symbols interpreted in the local states of proclets.

In many algorithms, each proclet accesses only a tiny part of the state. Our definition of local state, as including the whole global state, is convenient for our purposes because it provides maximum generality.

Remark 3.3. It may appear that, by including the whole global state in each proclet's local state, we have made inadequate provision for private memory. There would indeed be a problem if we were describing a situation where the proclets are antagonistic to each other and can gain an advantage by keeping information away from other proclets. Our proclets, however, are parts of a single overall computation. They are not antagonistic. If a proclet is not supposed to access certain information, then this can be assured by the algorithm it executes; it does not require a provision for privacy in the general model that we are developing.

In general, a state should include functions which, applied to a proclet, produce the various parameters needed by that proclet's calculation. This is another instance of the principle that the state should contain all the information relevant to the computation. In the example, `first` and `second` play this role, telling a proclet which instance of the adjacency relation it is to negate.

In our example of the graph-complementing algorithm, all the proclets execute the same algorithm. The only difference between them is that they have different values for the symbol `me`. It will be useful to notice that this situation is quite general. At first sight, it appears that one could easily have different algorithms for different proclets. Suppose, for example, that the graph-complementing algorithm were expanded to also make some unary predicate true of all vertices. That is, adjoin to it

```
do forall x in Vertex, P(x):=true enddo,
```

combining the two rules by means of `do in parallel`. Then some proclets execute the algorithm $\mathcal{A}1$ exhibited above to negate one adjacency relationship while other proclets execute

```
P(me):=true,                                            (A2)
```

a quite different-looking algorithm. Nevertheless, we can arrange for all proclets to execute the same algorithm if we remember, once again, that the state is supposed to include all information relevant to the computation. Here, this means that there should be unary predicates, say `Proclet1` and `Proclet2`, satisfied by the two sorts of proclets. Then all proclets execute the algorithm described by

```
if Proclet1(me) then A1
elseif Proclet2(me) then A2
endif.
```

It is clear that the same trick will reduce to a single algorithm any finite number of algorithms that may be written out in the overall algorithm for the whole computation. In our example, we took advantage of the fact that the labels for the two sorts of proclets were different; proclets executing $\mathcal{A}1$ were (labeled by) pairs of vertices while those executing $\mathcal{A}2$ were (labeled by) single vertices. In situations where several types of proclets are naturally labeled by the same objects (e.g., by pairs of vertices), an additional tag should be included in the labels to distinguish them.

What if algorithms for proclets are not written out in the overall algorithm but are produced as part of the computation? Then these algorithms will be produced in a certain form, perhaps as parse trees of programs in some programming language (or perhaps as strings or in some other format). In order for such a program to be executed (or even read) by a proclet under the constraint of Bounded Exploration, the program would have to be quite trivial. So in some sense, this situation cannot really arise. But even if we consider programs so trivial that a proclet could execute them, the "real" algorithm executed by the proclets would then be essentially an interpreter for that language. Each proclet reads its program and traverses its parse tree (perhaps after first creating the parse tree from another representation) and executes it according to the interpreter instructions. Thus, once again, what appeared to be many programs really amounts to just one.

We also adopt the convention that the set of proclets should be nameable in the vocabulary of the state. For now, this can be taken to mean a unary relation symbol `Proclet` defining the set of proclets or a Boolean expression `Proclet`(x) that is true just when the value of x is a proclet; later, when we have discussed backgrounds (Section 6), we shall be able to deal with this set more directly.

In the example above where we combined two algorithms $\mathcal{A}1$ and $\mathcal{A}2$ into one, the set of proclets would be the union of `Proclet1` and `Proclet2`, so `Proclet` would be definable as the disjunction of these two predicates.

The justification for requiring `Proclet` to be definable is, once again, the principle that states contain all the relevant information; surely the proclets involved in a computation are relevant to it.

The symbol `Proclet` (or the functions involved in defining `Proclet` if it isn't a single symbol) can be dynamic. That is, it is possible for proclets to be created or destroyed during the computation. We can still arrange, for the same reasons as above, that all proclets execute the same algorithm.

We point out that, although we regard algorithms as analyzable into proclets, we do not require that this analysis be unique. It is possible for an algorithm to be analyzable into subprocesses in more than one way; see Lamport [1997] for an example. Even if such non-uniqueness were to persist down to the level of proclets—if an algorithm were decomposed into proclets in more than one way—then any of these ways would lead to a simulation by an abstract state machine in our main result, Theorem 10.1.

In summary, this section has produced a picture of parallel computation very similar to the picture of sequential computation developed in Gurevich [2000]. The key difference is that, instead of a Bounded Exploration algorithm being executed once in the current state X, it is executed many times, once for each p satisfying `Proclet`, the execution by p being in the current state X augmented with p as the interpretation of `me`. The union of the update sets produced by all these proclets is the overall update set $\Delta(A, X)$.

4. A MORE REALISTIC PICTURE

What's wrong with the picture developed in the preceding section? Why did we call it oversimplified? The oversimplification is in the way we combined the results produced by the various proclets. We assumed that each proclet merely produces a set of updates and that the union of these sets is the overall update set $\Delta(A, X)$. But in fact there are very simple parallel algorithms that do not fit this description.

One example is given by the use of quantifiers in Gurevich [1995]. The evaluation of a Boolean term of the form $\forall x \in U\, \exists y \in V\, \varphi(x, y)$ can be viewed as a parallel process of the following sort. First, for each pair $\langle x, y\rangle \in U \times V$, the truth value of $\varphi(x, y)$ is computed by a proclet (or a larger process if φ is complicated). Then, for each $x \in U$, there is a proclet that collects the values of $\varphi(x, y)$ for all y and computes from them the value of $\exists y\, \varphi(x, y)$. Finally, the results from these proclets are collected, by another proclet, and used to compute the truth value of $\forall x\, \exists y\, \varphi(x, y)$. The important aspect of this example, for our present purposes, is that proclets do not merely produce updates but pass information to other proclets.

Similar situations arise in other familiar algorithmic operations. Suppose for instance that the algorithm uses a counter C, which proclets may increment by 1. If n proclets increment C (and no proclet does anything else to C), then the value of C in the next state $\tau_A(X)$ should be its value in the current state X plus n. As soon as $n > 1$, the actual update is not produced by any one of the proclets. And if the work of any proclet is regarded as producing the obvious update $(C, (), \mathtt{Val}_X(C) + 1)$ (where the $()$ in the middle means the empty tuple and where $\mathtt{Val}_X(C)$ is the value of C in the current state X), then the union of these update sets will consist of a single update that produces the wrong value for $\mathtt{Val}_{\tau_A(X)}(C)$.

Another example is given by updating a set S, where, in addition to (or instead of) replacing the entire set with an update like $S := T$, a proclet may put an element into S with an update like $S(a) :=$ `true` or take an element out of S. These situations are typical examples of the partial updates discussed

in Gurevich et al. [2001] and Gurevich and Tillman [2001] and used in AsmL [Foundations of Software Engineering Group].

What is really happening in such situations is that the individual proclets are producing something other than updates to be included in the final $\Delta(A, X)$. What they produce are pieces of information, like "increment C by 1," that need to be manipulated in some way to produce $\Delta(A, X)$. The required manipulation can be different in different situations. In our example of a counter, the required manipulation was just counting, but there are many more complicated situations. For example, if individual proclets can increment a counter by any number, not just by 1, then instead of counting we shall need addition. In the example of quantification, proclets must form the Boolean conjunction or disjunction of all the truth values passed to them by other proclets.

Can one give a general description of this manipulation? What common aspects are shared by counting, addition, conjunction, disjunction, and all the many other manipulations that could be imagined here? We claim that they can all be described in terms of operations on multisets.

A multiset M is just a set in which elements have multiplicities that are positive integers. A familiar example would be the set of zeros of a polynomial "counted with multiplicity." Formally, a multiset M can be defined as a function from an underlying set M_0, the domain of M, to the positive integers, the value of the function at an element x being the multiplicity of x as an element of M. We write $\mathrm{Mult}(x, M)$ for the multiplicity of an element x in a multiset M; formally, this is just $M(x)$ if $x \in M_0$ and 0 otherwise. We use double braces $\{\!\{ \ldots \}\!\}$ as a notation for multisets just as ordinary braces are traditionally used for sets (and we trust that our double braces are easily distinguished from nested ordinary braces, $\{\!\{ \ldots \}\!\} \neq \{\{\ldots\}\}$). Thus, $\{\!\{x_1, \ldots, x_n\}\!\}$ is the multiset M such that, for any a, the multiplicity $\mathrm{Mult}(a, M)$ is the number of subscripts i for which $x_i = a$, that is, the multiplicity is determined by how often a is listed in $\{\!\{x_1, \ldots, x_n\}\!\}$. Similarly, if r denotes a multiset, $t(x)$ is a term and $\varphi(x)$ is a Boolean-valued term, then

$$\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$$

denotes the multiset M such that, for every a,

$$\mathrm{Mult}(a, M) = \sum_{\substack{x \text{ such that} \\ \varphi(x) \text{ and } t(x)=a}} \mathrm{Mult}(x, r).$$

That is, M contains the values $t(x)$ for elements $x \in r$ satisfying $\varphi(x)$, and the multiplicities arise from the multiplicity of x as a member of r and from many different x's yielding the same $t(x)$. For example, if $r = \{\!\{0, 0, 1, 1, 1, 2, 2\}\!\}$, if φ is `true`, and if $t(0) = a$, $t(1) = b$, and $t(2) = a$, then $\mathrm{Mult}(a, M) = 4$ and $\mathrm{Mult}(b, M) = 3$; so $M = \{\!\{a, a, b, b, b, a, a\}\!\}$, obtained from r by replacing every element x with $t(x)$.

We also define binary sum (a sort of union) of two multisets by

$$\mathrm{Mult}(a, x \uplus y) = \mathrm{Mult}(a, x) + \mathrm{Mult}(a, y),$$

and we define the sum of a multiset of multisets by

$$\mathrm{Mult}(a, \biguplus M) = \sum_{x \in M_0} \mathrm{Mult}(a, x) \cdot \mathrm{Mult}(x, M).$$

In connection with the definition of $x \uplus y$, note that if a is in both x and y, then its occurrences in x and its occurrences in y are regarded as separate in the sense that they contribute independently to the sum $x \uplus y$. This is the case even if $x = y$, so $x \uplus x \neq x$ unless $x = \varnothing$. A similar remark applies to $\biguplus$.

The sum of all the multiplicities of elements in a multiset M is called the cardinality of M. If the elements of M are themselves numbers, then it is easy to define the sum of the members of M, namely the sum over all members $x \in M_0$ of x times its multiplicity in M. Similarly, one could define the product of a multiset of numbers.

We introduce two additional elementary operations on multisets. First, define TheUnique(x) for a multiset x to be y if y is the only element of x, with multiplicity 1; if x has cardinality $\neq 1$, or if x is not a multiset, then TheUnique(x) = `undef`. Also define AsSet(x) to be the multiset that has the same members as x but with multiplicity only 1.

Convention 4.1. In principle, sets and multisets are different sorts of entities and should be treated as distinct types. It is, however, possible to regard sets as a special case of multisets, and in this paper it is convenient to do so. We therefore adopt the technical convention that sets are identified with multisets in which the multiplicity of each member is 1. In particular, the empty set $\varnothing$ is identified with the empty multiset. With this convention, AsSet converts a multiset into a set by keeping the members but forgetting the multiplicities. The standard Boolean operations on sets can be extended naturally to multisets, but the only one we shall need here is the sum, generalizing union, as defined above.

In our example of incrementing a counter, the individual incrementing proclets can be regarded as contributing their pieces of information, "increment C by k," or simply the numbers k to a multiset, to which the operation of addition is to be applied. Finally, the sum σ so obtained is used to define the required update $(C, (), \mathtt{Val}_X(C) + \sigma)$.

More generally, the partial updates discussed in Gurevich et al. [2001] and Gurevich and Tillman [2001] and used in Foundations of Software Engineering Group, can be viewed as messages, sent to a proclet that implements the appropriate sort of integration of the partial updates to produce (ordinary, total) updates.

We have broken the manipulation of information into two stages: First, collect into a multiset all the pieces of information produced by the proclets. Second, perform some operation (for example addition) on the resulting multiset. The second stage can be regarded as having bounded exploration, provided we think of it as acting on the multiset, not on the individual elements in the multiset. That is, the only genuinely parallel operation is the collection of the many individual pieces of information into a single object, the multiset. We regard this observation as a conceptual simplification of parallelism.

It is remarkable how few basic concepts are needed for formalizing parallel, synchronous computation.

What an algorithm does with such multisets of information—how the pieces of information are combined to produce the updates—may (but need not) be given by a single proclet's computation, that is, a computation with bounded exploration, taking the multiset as input. In this situation, the functions used for that computation (at that level of abstraction), for example the function mapping a multiset to its cardinality or the function mapping a multiset of numbers to the sum of its elements, should be part of the state. This is an instance of the general principle that the state contains everything needed to determine how the algorithm proceeds. Ordinarily, the functions used here will be static, but there is no *a priori* reason why they could not be dynamic.

A slightly more complicated situation arises if (as will often be the case) the information produced by the proclets must be processed by a parallel computation rather than by a single proclet. This occurs in the example, considered above, of computing the truth value of $\forall x \in U\, \exists y \in V\, \varphi(x, y)$. For another example, consider many counters $C(i)$ (where C is a function symbol and i ranges over some subset of the state), each capable of being incremented by any of numerous proclets. Then these proclets produce pieces of information of the form "increment $C(i)$ by n," which must be appropriately collected and combined. A natural way to do this is to collect all the increment instructions for each individual i into a multiset, say $C\text{increments}(i)$, and then have, for each i, a proclet that sums the numbers in $C\text{increments}(i)$ and produces the appropriate update $(C, (i), \mathtt{Val}_X(C)(i) + n)$. We may refer to these as second-level proclets, since they take as their inputs the multisets of information pieces produced by the previous, first-level proclets. Each of the second-level proclets satisfies Bounded Exploration, and together they produce (as in Section 3) the required set of updates.

One can imagine third- and higher-level proclets, but for any algorithm the number of levels should be bounded. This boundedness is a result of our decision, described in the introduction, that any single step in our algorithms involves only bounded sequentiality.

Summarizing the present section, insofar as it goes beyond the earlier discussion, we have that, in addition to producing updates directly, proclets may produce pieces of information, which we call *messages*, to be sent to other proclets for use in their computations. These higher-level computations may then produce updates or further messages to yet higher-level proclets. But bounded sequentiality requires a bound, depending only on the algorithm and not on the state, for the height of this hierarchy of proclets. The update set $\Delta(A, X)$, leading to the next state, consists of all the updates produced by all the proclets.

Remark 4.2. A similar use of multisets occurred in Grädel and Gurevich [1998]. The logic developed in that article did not allow quantification over, for example, the natural numbers or the real numbers, yet certain basic operations, like addition, had to be performed on many numbers at a time. Multisets emerged as the natural domain for such operations.

5. PUSHING AND PULLING INFORMATION

In the examples considered in the preceding section, each proclet p sent information to only a finite number of proclets q (usually just one), each of which is "known" to p in the sense that the algorithm run by p specifies all the recipients q.

Some parallel algorithms, however, involve communication of a different sort between proclets. Suppose, for example, that there is a global memory consisting of some registers that many processes can access (for example in the PRAM model of parallel computation). It is certainly imaginable, indeed likely, that the number of processes accessing a particular global memory register is not bounded but increases with the size of the input. In this situation, we can regard each memory register as a proclet, but the program of this proclet could not specify all the readers that access this register. Indeed, since a proclet satisfies Bounded Exploration, its algorithm can specify only a bounded number of elements of the state and, in particular, only a bounded number of other proclets.

In situations like this, the transmission of information from the register to any one of the processes that access it must result from the algorithm executed by that process, not from the algorithm of the register. In other words, the communication is driven by an action of the recipient rather than by an action of the sender.

In order to make our description of parallel algorithms general, we must allow for both sender-driven and recipient-driven communication. We refer to the former as *pushing* information and the latter as *pulling* it.

Remark 5.1. What if we have an unbounded number of information sources and an unbounded number of information recipients, with each of the latter receiving information from each of the former? No source can specify all the recipients, and no recipient can specify all the sources, so neither pushing nor pulling suffices for this sort of many-to-many broadcasting. Nevertheless, the combination of pushing and pulling suffices. One approach is to add a bulletin board as an additional proclet. All the sources push their information to the bulletin board, where all this information is collected into a multiset, and then all the recipients pull this multiset from the bulletin board. Another approach is to add, for each pair ⟨source, recipient⟩ another proclet to serve as a communication channel between these two. The channel would pull information from its source and then push it to its recipient.

The remainder of this section describes in detail how information is transmitted between proclets within one step of the overall computation. Pushing and pulling differ in several ways, so we treat them separately.

5.1 Pushing

For a message to be pushed from a proclet p to another proclet q, it is necessary that the Bounded Exploration algorithm executed by p specify both the content of the message and the intended recipient q. Thus, each p can send messages to only a bounded number of other proclets in a single step. In contrast, there

need not be a bound on the number of messages received by a proclet q in one step; if, for example, q is the value of some term in the global state, then all the other proclets could push messages to it.

It is here that multisets are an essential part of the picture. We think of all the messages sent to q (in a particular step of the execution of the overall algorithm) as collected into a multiset, which we call $\texttt{Mailbox}(q)$. Being subject to Bounded Exploration, q cannot deal with the (possibly unboundedly many) elements of its mailbox individually, but it can deal with the entire mailbox regarded as a single element of its state. Recall in this connection the examples above, where a proclet associated with a counter took the multiset of incoming "increment" instructions and combined them (by means of a static function in the state) into the required update for this counter. Recall also the proclets involved in the evaluation of a quantified formula, which take a multiset of truth values of (instances of) a subformula and combine them (again by means of a static function) to produce the truth value of the larger formula.

In the multiset $\texttt{Mailbox}(q)$, multiplicities arise if several proclets send the same message to q and also if a single proclet sends a message to q several times (in one step). The example of the counter above shows that both sorts of multiplicities must be taken into account.

The function Mailbox introduced here is not part of the vocabulary of the algorithm. It is not something that persists from one state of the computation to the next, changing only when updated. Rather, it is entirely internal to a single step; there is no connection between Mailbox at one step and at the next. In each step of the algorithm, $\texttt{Mailbox}(p)$ contains just the messages sent to p during that step.

For any proclet p, $\texttt{Mailbox}(p)$ will function as an element of the local state in which p works (details about local states will be spelled out later), but it is not part of the global state of the overall algorithm. And it is only the global state that persists from one step of the algorithm to the next. This is not to say that a proclet cannot "remember" its mailbox from one step of the algorithm to the next. But in order to do so, it must store this mailbox in some location of the global state. This requirement is a consequence of the general principle that anything relevant to the future steps of the algorithm must be part of its (global) state resulting from the present step.

One can reasonably ask whether a Mailbox function as described above exists at all. After all, it appears that this function (at a particular step of the computation) depends on the computations of the proclets (for $\texttt{Mailbox}(p)$ contains the messages sent to p by the other proclets) while these computations depend in turn on Mailbox (for p can make use of $\texttt{Mailbox}(p)$ in its computation). In fact, it is easy to design an algorithm for which no such function can exist. Let there be just one proclet p, and let it execute the algorithm: If 0 is not in your mailbox, then send 0 as a message to yourself. So $\texttt{Mailbox}(p)$ should contain 0 if and only if it does not contain 0. Our postulates for parallel computation will exclude such pathology and will ensure that the Mailbox function exists; see Theorem 7.22.

In examples like those of the preceding section, the Mailbox function is easy to describe. In those examples, proclets were arranged in a hierarchy, with those

at the first level sending messages to those at the second level, which in turn send messages to proclets at the third level, and so on (for a bounded number of levels). Then for q at the first level, Mailbox(q) is empty, as no proclet sends messages to q. For proclets q at the second level, Mailbox(q) consists of the messages received from first-level proclets. And so forth. Our axioms will not explicitly refer to levels, but they will imply that a structure similar to this is present, ensuring the existence of Mailbox (as well as similar functions to be defined in the next subsection).

Terminology 5.2. When pushing a message, the sender produces the pair ⟨recipient, content⟩. When we need to be careful, we shall use the word *mailing* for this pair and *message* for the content alone.

5.2 Pulling

In pushing, all the "work" is done by the sender, who specifies both the recipient and the content of the mailing. The recipient can sit passively and just watch his mailbox fill up. Pulling is more complicated, because the recipient, who specifies the sender, obviously cannot also specify the content. Rather, each (potential) sender displays some information that he intends to make public, and a recipient specifies the sender whose display he wants to read.

We could imagine a sender displaying several sorts of information, perhaps with some distinguishing labels. Then a recipient would specify not only a sender but also a label, in order to read a particular one of the labeled pieces of information displayed by that particular sender. There is, however, no loss of generality in dispensing with labels and assuming that a sender displays only a single piece of information, at least as long as the states of the algorithm are sufficiently rich. The reason that this works is that, instead of displaying several pieces of data x with labels L, a sender could just display a set of ordered pairs $\langle x, L\rangle$. By reading this set and then extracting the appropriate component, a recipient can obtain the same information as by reading the information displayed with a particular label. Since it costs no generality and it simplifies the notation, we adopt the convention that every proclet displays at most one piece of information.

Each proclet p sets up, without having any particular recipient in mind, its display, which we call Display(p). The meaning of such a display is that this information is made available to any proclet that wants to pull it and that can name p. The vocabulary of a proclet's algorithm can contain a dynamic 0-ary function symbol myDisplay; a proclet sets up its display by assigning values to this symbol. If a proclet p assigns no value or assigns two different values to myDisplay, then Display(p) is undef at that step.

To pull information displayed by p, some other proclets q can use terms of the form Display(p) (or more precisely, Display(t) where t is a term with value p when evaluated in the local state of q). Thus, Display is used as a static unary function in the algorithms executed by individual proclets.

Remark 5.3. An earlier version of this work did not use Display functions but instead allowed explicit Pull commands in the programs executed by the

proclets. Such a command would pull information, from a specified proclet and store it in a specified location. The present set-up, using `Display` functions, is simpler and can easily simulate the old set-up.

Like `Mailbox`, the function symbols `myDisplay` and `Display` are not part of the vocabulary of the overall algorithm. Their meanings do not persist from one state of the algorithm to the next. Rather, they are used entirely within one step of the algorithm, and their values in two consecutive steps may be quite unrelated. In other words, the values of `Mailbox`, `myDisplay`, and `Display` are not given by the (global) state of an algorithm A but play an auxiliary role in describing the transition function τ_A from one state to the next. In the algorithm executed by a proclet p, `Display` acts like a static function; `myDisplay` acts like a dynamic function, whose value is `undef` unless and until p assigns a different value.

As with `Mailbox`, our axioms will imply that `Display` always has well-defined values; see Theorem 7.22.

6. BACKGROUNDS

This section is about possibly non-obvious elements of the states of a computation. "Obvious" is not a precise term, but what we mean is adequately suggested by the remark that, for an algorithm operating on a graph as input, the obvious elements would be the vertices of the graph, and the obvious functions would be the adjacency relation and the desired output. Non-obvious elements would include, for example, the Boolean values `true` and `false` as well as the element `undef` and the infinitely many reserve elements that are, according to the conventions of Gurevich [2000], present in all states.

What non-obvious elements should states include? That depends on the meaning of "should." As always, the state contains everything relevant to the future progress of the algorithm. So it would be reasonable to include in the state all of the sorts of things that algorithms often need, for example numbers, sets, maps, and sequences. On the other hand, we can interpret "should" as referring to just those elements that are needed for our present purpose, namely to simulate the algorithm with an abstract state machine. These needs are remarkably limited.

In this section, we first discuss a number of items that should be included in the state under the first interpretation of "should"—things that programmers expect their algorithms to have access to. Afterward, we briefly indicate the considerably shorter list of items that we actually need in this article. We also comment briefly on how some items can be obtained from others, that is, how the former can be regarded as syntactic sugar for constructs built from the latter.

Having adopted the Abstract State Postulate from Gurevich [2000], we are committed to using first-order structures as defined in Gurevich [2000], namely with a vocabulary consisting entirely of total function symbols. As explained in Gurevich [2000], this convention is based on the availability of the Boolean values `true` and `false` and the Boolean connectives, so that functions can represent relations; it also uses `undef` to represent partial functions by total ones.

Another commitment that we inherit from Gurevich [2000] is that the base set of the state should not change during a computation; this was justified on the basis of having an infinite supply of reserve elements that can be imported when, for example, an algorithm wants to add a new vertex to a graph. Thus, although the presence of these elements is not explicitly given by the Abstract State Postulate, it is implicit in the sense that the plausibility of the postulate depends on the availability of these elements.

There is another source of non-obvious elements, not needed in Gurevich [2000] but quite important for our present purposes. In fact, we already made use of these non-obvious elements in the preceding sections, when we assumed that the state contained tuples of elements (as in the labels of proclets and in the pieces of information produced by proclets) and multisets (collecting all the information produced by many proclets or the information displayed by one proclet). Here we are dealing with constructions that produce new elements (tuples or multisets) from old ones (the components of a tuple or the elements of a multiset). Such constructions can be iterated, producing tuples of multisets, multisets of tuples, and so forth. Of course, one can think of many other constructions of this sort, for example sets, maps, and sequences. A general approach to such constructions was developed in Blass and Gurevich [2000] under the name of "background classes"; a brief summary is included in Gurevich et al. [2001].

A background class K amounts to a way to associate with any set U a structure $K(U)$ (of a fixed vocabulary) having U as a subset of its base set. This construction $K(U)$ is required to satisfy some axioms, saying essentially that it respects embeddings (every embedding of sets $U \to V$ extends to an embedding of structures $K(U) \to K(V)$) and intersections; for details, see Blass and Gurevich [2000]. One can think of $K(U)$ as the result of adding to U various sorts of data structures built from the elements of U. For example, there is a background class for which $K(U)$ consists of U together with all (finite) sequences of elements of U, the vocabulary of $K(U)$ having symbols interpreted as the empty sequence, as the function sending each element of U to the corresponding sequence of length 1, and as the binary operation of concatenation. Numerous other examples of background classes are given in Blass and Gurevich [2000].

An algorithm with background K takes as inputs structures of some vocabulary (disjoint from the background vocabulary), but the initial state of the computation is richer, being obtained from the input structure I by enlarging I twice—first by adding the logic elements `true`, `false`, and `undef` (with the Boolean operations) and an infinite naked set of reserve elements and then by applying K to the result. This introduces new functions, namely those coming from the K construction; the original functions of I are extended to the larger base set by giving them their default values whenever any argument lies outside $|I|$. The computation then proceeds in this twice-enlarged structure. Thus for example, if K is the tuple background described above, then the computation could make use of tuples of elements of the input structure.

A realistic description of algorithms would involve quite a rich background, including numbers, sets, multisets, maps, sequences, and the like, since all these

things are generally taken to be available when designing algorithms. Very few of these things, however, will actually be required by our axiomatic description of parallel algorithms. In fact, we require just multisets and ordered pairs. The postulates will permit a very rich background but will not demand it.

Why do we choose postulates that demand only a little background? It makes our main result stronger in that it applies to all algorithms that have at least this little background, whether or not they use additional background beyond what the postulates require.

Why, then, do we require any background at all? Couldn't we get an even stronger theorem by not requiring any background? The difficulty with this suggestion is that, when we produce an ASM to simulate a given program, the ASM needs a certain amount of background—essentially multisets and ordered pairs. If the given algorithm did not have this background, then its states would not be the same as the states of the simulating ASM and it would not be equivalent to the ASM. As a result, we would have to replace our notion of equivalence with a weaker and more complicated notion of simulation. By requiring a certain minimum background in the given algorithm, whether or not that algorithm actually needs this background, we ensure that it and the simulating ASM are equivalent in the strong sense of Definition 2.3.

In addition and more importantly, it is reasonable to claim that all parallel algorithms make at least implicit use of multisets. Combining the activities of many processes into a single overall state transition seems to involve forming multisets of outputs of the individual processes, even if the algorithm is expressed in a language that does not make these multisets explicit. Thus, our background requirements merely make explicit what is already implicit in the idea of parallelism. They can be regarded as an instance of the general principle that everything needed by a computational process should be present in its state.

Remark 6.1. Since multisets are less familiar than sets, and since sets are widely used as a foundation for all of mathematics (and thus for theory of computation, as in Blass et al. [1999] for example), it is reasonable to ask whether sets could be used in place of multisets in our treatment of parallel algorithms. The answer is yes, there are at least two ways to do this, but each involves some costs.

Multiplicities arise mainly from the possibility of repetition of messages. For example, if several proclets request that the same counter be incremented, or even if one proclet makes several such requests concerning the same counter, then the multiplicities of the requests must be taken into account in order to increment the counter by the right total amount. One could avoid multiplicities by ensuring that the messages are all distinct. For example, we could require that proclets "sign" the messages they send and, if they send the same message several times, then the occurrences should be tagged. If proclet p sends message m, it would actually send, say, $\langle m, p, n\rangle$ where n is a suitable tag. (For example, if the proclet's algorithm is such that each message originated from a distinct, identifiable line in a program then n could be the number of that line.) Instead of signatures, one could arrange for the "operating system"

to add identifiers to messages; distinct identifiers could be new elements imported from the reserve. In addition to a suitable supply of tags (e.g., numbers or reserve elements—which would certainly be available in any realistic system), any tagging scheme requires some additional complication in the functions used to combine messages and produce updates. For example, consider a proclet responsible for updating a counter in response to "increment by k" commands. In the multiset view of the situation, this proclet could receive, as messages, the relevant numbers k, and it would apply to the multiset of these numbers a static function "sum" that sends any multiset of numbers to the sum of its members (with multiplicity of course). If our proclet receives the messages tagged, then it needs to apply a function that, given a set of tagged numbers, strips off the tags and adds the multiset of numbers so obtained. So this slightly more complicated and considerably less natural function would have to be built into our states. (We can't separate the operation into the obvious two parts, (1) strip off tags and (2) add, because the result of the first would be a multiset, just what we're trying to avoid. Also, it would not do to regard "strip off tags" as an operation to be performed in parallel by many proclets, one for each tagged message; for these proclets to communicate their results to the counter proclet, they'd have to send messages, which need tags.)

Another way to eliminate multisets in favor of sets is to invoke the fact that all (present day) mathematical entities can be defined in terms of sets. Specifically, a multiset can be viewed as a map from the underlying set to the natural numbers, sending each member of the multiset to its multiplicity. And maps can be viewed as sets of ordered pairs, while ordered pairs are themselves viewed as sets, say via Kuratowski's coding $\{\{x\}, \{x, y\}\}$. Of course, simple operations on multisets will look quite complicated after this coding.

As indicated above, the proof of our main theorem, Theorem 10.1, will require very little background, namely only ordered pairs and multisets. We regard this fact as important information, considerably simplifying the conceptual basis of parallel computation. Nevertheless, this simplification is based in part on Convention 4.1, identifying sets with certain multisets, and one might argue that it would be more honest to include sets as part of our background. The issue is to what extent such identifications or codings are to be permitted for the sake of simplifying the general framework, since they will usually complicate individual programs.[2]

We shall state and prove our results in a form that requires only multisets and ordered pairs in the background (though richer backgrounds are permitted). This makes our results stronger than they would be if a rich background were demanded. In the case of our main theorem, there is practically no cost for this strength. Had we included sets in our background, then `Proclet` would be a set rather than a multiset (with all multiplicities 1), but the ASM required in the proof would look exactly the same.

[2]There is an analogy with the possibility of expressing all Boolean connectives in terms of just one, "nand." This possibility can be used to simplify the general theory of connectives, but most of the propositional combinations occurring in practice become unpleasantly complicated when written in terms of nand.

In some of our other results, particularly in Section 11, there would be a greater cost. Some ASMs considered there naturally use a background that includes sets and natural numbers. We can get by with our standard background of multisets and ordered pairs, but at the cost of some awkward coding.

For readers who are interested in reducing everything to the standard background, we shall explain how to express everything we need in terms of this background. The explanations will be headed "*Reduction*." Readers who don't care about reducing to the standard background, who are willing to include sets, numbers, and some natural functions on them as part of the background because they are available in any realistic model of computation, are invited to skip these explanations.

Here's the first of the explanations, repeating what was said above about representing sets as multisets, and indicating how we could, if we were so inclined, reduce the standard background even further.

Reduction 6.2. For theoretical (but not for practical) purposes many sorts of background items can be expressed in terms of others. For example, using multisets, we can code sets as multisets in which each member has multiplicity 1. We can code natural numbers, by identifying n with the multiset whose only element is, say, `true` with multiplicity n. (Notice that, with reasonable operations on multisets, this identification would have the effect of representing numbers in unary notation. So it is definitely unsuited for real computation, but it can be convenient for some theoretical purposes. See Section 11.) We can code an ordered pair $\langle x, y \rangle$ as the multiset $\{\!\{x, y, y\}\!\}$.

If we wanted to be really stingy, we could even dispense with the propositional connectives, defining negation $\neg\varphi$ as the equation $\varphi = \texttt{false}$ and defining conjunction $\varphi \wedge \psi$ as the equation $\langle \varphi, \psi \rangle = \langle \texttt{true}, \texttt{true} \rangle$.

The only one of these reductions that we shall actually use in our main result is the one whereby sets are special multisets. We need sets so rarely (essentially just the set of proclets) and multisets so often that this reduction seems worthwhile.

Remark 6.3. Multisets are, as the name implies, a generalization of sets, the generalization being that multiplicities are permitted. They are also an abstraction from finite sequences, the abstraction being that the order of the components of a sequence is ignored to form a multiset. This second way of viewing multisets is relevant to parallel computation if one shifts the level of abstraction. Parallelism can be used to describe processes that are not intrinsically parallel but are actually sequential; parallelism is used to model the fact that the order of certain sequential steps is immaterial. When parallelism is used in this way as an abstraction of sequentiality at a lower level of abstraction, our multisets arise as sequences in which we have abstracted from the order of the components.

Another way of describing this connection between multisets and sequences is that sequences can be used as an implementation of multisets. In particular, the multisets of messages that we see in all parallel computation and that motivated us to work with multisets are often implemented sequentially as queues of messages.

7. THE GENERAL DESCRIPTION

In this section, we combine the points discussed in the preceding five sections, giving what we consider to be a quite general description of parallel computation. In the next section, we shall indicate how several familiar models of parallel computation fit this description.

7.1 Basic Postulates

To begin, we assume that a parallel algorithm satisfies the Sequential Time and Abstract State Postulates, which we repeat here for convenient reference.

—**Sequential Time:** An algorithm A is associated with a set $\mathcal{S}(A)$ of states, a subset $\mathcal{I}(A) \subseteq \mathcal{S}(A)$ of initial states, and a map $\tau_A : \mathcal{S}(A) \to \mathcal{S}(A)$ called the one-step transformation.

—**Abstract State:** All states of A are first-order structures with the same finite vocabulary, which we call the vocabulary of A. τ_A does not change the base set of a state. Both $\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphisms. Any isomorphism from a state X to a state Y is also an isomorphism from $\tau_A(X)$ to $\tau_A(Y)$.

In connection with the Sequential Time Postulate, recall Terminology 1.1 in the introduction. We deal exclusively with algorithms which, although perhaps not sequential algorithms, have sequential time. That is, they compute in discrete, well-defined steps, although the amount of reading and writing done in one step may vary from state to state in an unbounded manner.

7.2 Background Postulate

The next postulate is based on the discussion in Section 6. It summarizes several consequences of the general principle that a state of a computation must contain all the information relevant to the future progress of that computation. It also incorporates a few of the conventions about states that were assumed and used in Gurevich [2000].

—**Background:** Each state contains the following:
  —the elements `true` and `false`, the Boolean operations on them, `undef`, and the equality predicate,
  —all ordered pairs of elements of the state, with a binary function symbol for pairing $\langle x, y \rangle$ and unary functions `first` and `second` for extracting the components of a pair,
  —all finite multisets of elements of the state, with symbols for the empty multiset $\varnothing$, singletons $\{\!\{x\}\!\}$, binary sum $x \uplus y$, general sum $\biguplus x$, TheUnique, and AsSet, and
  —a variable-free term `Proclet` naming a finite set, also called `Proclet`.

Remark 7.1. In accordance with the Abstract State Postulate, isomorphic copies are to be permitted in the Background Postulate. That is, we do not require the state to contain actual pairs; an isomorphic copy of the pairing structure is good enough. Similarly for multisets.

Remark 7.2. Notice that the Background Postulate only imposes a lower limit on the background. An algorithm may very well use a far more extensive background than the axiom requires. It might, for example, use sequences, maps, natural numbers, and even infinite sets.

Remark 7.3. To some extent, the Background Postulate reflects properties inherent in the notion of parallel computation. For example, we consider multisets to be essential in this sense because parallel computation must implicitly involve mailboxes and these are multisets; see the discussion in Section 4. Other aspects of the Background Postulate reflect decisions that were somewhat arbitrary, because of the possibility of expressing some concepts in terms of others. We could have required finite sets to be available (as in Blass et al. [1999]), but we have chosen instead to regard them as a special case of multisets where the multiplicities of members are always 1. This choice was motivated mainly by the fact that we make so little use of sets (as opposed to multisets) in this paper; we need just one set, `Proclet`, in the remaining axioms and in the proof of our main theorem. Also, the "coding" of sets as special multisets seems quite natural.

On the other hand, we explicitly require ordered pairs to be present, though they could be represented as special multisets, identifying $\langle x, y \rangle$ with $\{\!\{x, y, y\}\!\}$ or, as is customary in set theory, with the set $\{\{x\}, \{x, y\}\}$. We chose not to rely on these codings since they do not seem so natural and since the definitions of `first` and `second` from either coding look a bit awkward.

Remark 7.4. There is also some arbitrariness in the choice of the specific functions to require, particularly in dealing with multisets. For example, the unordered pair is defined from our required operations by $\{\!\{x, y\}\!\} = \{\!\{x\}\!\} \uplus \{\!\{y\}\!\}$. We could equally well have required unordered pairs and then obtained binary sum as $x \uplus y = \biguplus \{\!\{x, y\}\!\}$. We would then not need the singleton operation, since $\{\!\{x\}\!\} = \mathrm{AsSet}(\{\!\{x, x\}\!\})$.

It may be useful to note, particularly for mathematical purposes, that it would suffice to use the following four operations as the basic ones: pairing $\{\!\{x, y\}\!\}$, sum $\biguplus x$, TheUnique, and AsSet. Indeed, we have already seen how these allow us to define singletons and binary sum. They also provide $\varnothing$ as $\biguplus(\{\!\{a\}\!\})$ where a is anything that is not a multiset, for example `true`.

These four operations and the comprehension construct $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ (which will be included in our definition of ASM terms in Section 9) are independent; none can be expressed in terms of the others. We briefly sketch the reasons. For this purpose, we need the notion of the *transitive closure* $TC(x)$ of a multiset x; this is the multiset whose members are x, all members of x, all their members, and so on, all counted with the obvious multiplicities. That is,

$$TC(x) = \biguplus_{n=0}^{\infty} U(n, x)$$

where

$$U(0, x) = \{\!\{x\}\!\} \quad \text{and} \quad U(n+1, x) = \biguplus U(n, x).$$

We also need the notion of the *hereditarily finite* multisets over a given set A of atoms; these are defined recursively as finite multisets whose members are either elements of A or hereditarily finite multisets over A.

The multisets x such that all multisets in $TC(x)$ have cardinality 0 or 1 form, together with the atoms, a universe closed under $\biguplus$, TheUnique, AsSet, and comprehension, but not under pairing. So pairing cannot be expressed in terms of the others. (Here and below, we include `undef` among the atoms, for the sake of closure under TheUnique.)

The multisets x such that all multisets in $TC(x)$ have cardinality 0, 1, or 2 form, together with the atoms, a universe closed under pairing, TheUnique, AsSet, and comprehension, but not under $\biguplus$. So $\biguplus$ cannot be expressed in terms of the others.

For a nonempty set A of atoms, the universe consisting of the hereditarily finite multisets over A (but not the elements of A) is closed under pairing, sum, AsSet, and comprehension but not under TheUnique. So TheUnique cannot be expressed in terms of the others.

For the independence of AsSet, consider the universe consisting of two atoms, a and `undef`, and all the hereditarily finite multisets over them. Call an element x of this universe *fat* if either it is `undef` or, for every membership-chain from a to x,

$$a = c_0 \in c_1 \in \cdots \in c_{l-1} \in c_l = x,$$

there is some i such that c_i is a member of c_{i+1} with multiplicity at least 2. One can show, by induction on terms built using pairing, $\biguplus$, TheUnique, and comprehension, that if t is such a term and if its free variables are interpreted as fat objects, then the value of t is also fat. Since $\{\!\{a, a\}\!\}$ is fat and $\mathrm{AsSet}(\{\!\{a, a\}\!\}) = \{\!\{a\}\!\}$ is not, it follows that no such term can match $\mathrm{AsSet}(x)$.

Finally, for any term $t(x, y)$ built from pairing, sum, TheUnique, and AsSet but without comprehension, there is a finite number n, computable from the syntactic structure of $t(x)$, such that, if x is interpreted as a set of atoms and if y is interpreted as an atom not in x, then the transitive closure of the value of $t(x, y)$ will contain at most n copies of y. Thus, taking x to be a set of more than n atoms, we see that pairing, sum, TheUnique, and AsSet cannot match the comprehension term $\{\!\{ y : z \in x : \mathtt{true} \}\!\}$.

Remark 7.5. The operations on multisets listed in the postulate, namely $\varnothing$, $\{\!\{x\}\!\}$, $x \uplus y$, $\biguplus x$, $\mathrm{TheUnique}(x)$, and $\mathrm{AsSet}(x)$ are intended to be a very small selection, adequate for our purposes and intuitively justified. Here "adequate" means that they suffice for our main theorem; "intuitively justified" means the following. Just as multisets are implicit in the very notion of parallel computation, in the form of the multisets $\mathtt{Mailbox}(p)$ of messages received by a proclet p, so these operations on multisets, with the possible exception of AsSet, will arise automatically in the context of messages. An absence of incoming messages and a single incoming message correspond respectively to $\varnothing$ and $\{\!\{x\}\!\}$, with TheUnique finding the message when there is only one. A proclet receiving a multiset x of messages from one source and y from another receives altogether $x \uplus y$; $\biguplus x$ arises similarly when there are many sources.

There are other functions associated with multisets, which seem quite natural but which we do not require to be present in the states of our algorithms because we have no need for them in this article. Others are omitted because we use them only slightly and can easily express them in terms of the basic operations we selected. Examples include Boolean-valued membership and numerical-valued multiplicity.

Remark 7.6. Since AsSet played an exceptional role in the preceding remark—not being obviously needed for a general description of parallelism—it may be worthwhile to point out that we make very little use of it. In fact, it is used in only three places in this article, and one of the three uses can be eliminated. Another use could be eliminated if we changed the definition of TheUnique to ignore multiplicities.

Specifically, we use AsSet near the end of Section 8.5 to convert a multiset of proclets into a set, as required by the Background Postulate; we use it again to define the displays in the proof of Theorem 10.1; and we use it a third time in Section 11.1 to produce a term $\mathcal{L}(p)$ for a set consisting of certain elements $u_1, \ldots, u_N$, each element occurring only once in the set regardless of how often it is listed among the u_i's.

The second use is in the context TheUnique(AsSet(x)), so we could dispense with AsSet if we redefined TheUnique to extract the unique element from a multiset even if that element occurs with multiplicity larger than 1.

The third use of AsSet can be eliminated as follows. First, note that we can easily define non-membership in multisets:

$$a \notin b \Leftrightarrow \{\!\{x : x \in b : x = a\}\!\} = \varnothing.$$

Now we can define, for each finite n, the n-ary set-forming operation $\{u_1, \ldots, u_n\}$ by using $\varnothing$ when $n = 0$ and then proceeding inductively with

$$\{u_1, \ldots, u_n, u_{n+1}\} = \{u_1, \ldots, u_n\} \uplus \{\!\{z : z \in \{\!\{u_{n+1}\}\!\} : z \notin \{u_1, \ldots, u_n\}\}\!\}.$$

Concerning the first use of AsSet, which seems not to be eliminable, see Remark 8.3.

Remark 7.7. It may seem strange that we require multisets in the background after having introduced in Blass et al. [1999] a system of parallel algorithms that uses only sets, not multisets. In accordance with the preceding discussions, we hold that multisets, though not explicit in Blass et al. [1999], are implicit there, just as in all parallel computation. What is special in the system of Blass et al. [1999] is that the "operating system" there quickly reduced the multisets (of messages) to ordinary sets by forgetting their multiplicities. But if we look closely, we find multisets implicit in parallel operations like `do forall` $x \in r$, $R(x)$. Here each $x \in r$ produces a set of updates, and all these sets together constitute *a priori* a multiset. The multiplicities are removed and the updates checked for clashes before any updates are executed.

Notice also that the set-up in Blass et al. [1999] took advantage of the fact that any reasonable construction, including multisets, can be modeled, however awkwardly, within set theory. If we consider, for example, several proclets incrementing the same counter, it is clear that a natural description involves a

multiset of updates. To model this with sets, as in Blass et al. [1999], instead of multisets, we would have to introduce some unnecessary and inelegant coding. See Remark 6.1 for additional comments about sets versus multisets.

It may be worth mentioning that a version of our main theorem would hold if we required of our background only that it be able to simulate (within a single step) multisets and the basic operations on them, rather than requiring that the multisets and their operations actually be part of the state. This version of the theorem would cover the computational system introduced in Blass et al. [1999]. To prove the theorem in this case, all uses of multisets in our proof would have to be replaced by the corresponding simulations.

Remark 7.8. The requirement, in the Abstract State Postulate, that the base set of the state cannot change during the computation was justified in Gurevich [2000] by the availability of an infinite reserve. Thus, we may regard the Abstract State Postulate as implicitly requiring the availability of such a reserve. We have chosen not to make this requirement explicit in the Background Postulate, simply because it will not be needed for our results.

Remark 7.9. It would do no harm to require `Proclet` to be a single, nullary symbol, but the added generality of allowing it to be a variable-free term makes our description conform better to intuition. The set of proclets should be definable in the current state, but it needn't be a basic function. It might, for instance, be defined as the union of two sets `Proclet1` and `Proclet2`, as in the example in Section 3. Note that we allow `Proclet` (or function symbols used in its definition) to be dynamic, so the set of proclets can change from one step of the computation to another; proclets can be created and destroyed.

7.3 Proclets

It remains to describe how each individual proclet works and how the proclets cooperate to produce the update set $\Delta(A, X)$ that leads from a (global) state X to the next state $\tau_A(X)$. The present subsection deals with the working of single proclets; the next subsection will describe their cooperation.

Although it is quite natural and logical to discuss first how a single proclet works and afterward how proclets interact, it leads to a rather unintuitive picture. The reason is that the local state in which a single proclet works should be determined by the actions (pushing messages and setting up displays) of other proclets. We therefore include in the present subsection some interaction-related definitions and remarks intended to make the discussion of single proclets more intuitive; the full import of these definitions and remarks will become explicit only in the next subsection when we consider interaction in detail.

We begin by recalling, from the discussion in Section 5, that the local state in which a proclet p works contains, in addition to the global state, two things produced by the action of (other) proclets, namely the distinguished element `Mailbox`(p) and the unary function `Display`. It will be convenient to have a name for all this proclet-produced information—not just the part relevant to

the computation of a particular proclet p, but all of it. That is the role of the following definition.

Definition 7.10. A *ken*[3] K of a state X consists of X together with two functions $\texttt{Mailbox}_K$ and $\texttt{Display}_K$, both having as their domain the set of proclets in X, such that the values of $\texttt{Mailbox}_K$ are multisets. (The values of $\texttt{Display}_K$ can be arbitrary elements of X.)

The next definition makes official what was already mentioned about the state used by a proclet.

Definition 7.11. Suppose K is a ken of a state X and suppose p is a proclet in X. Then the *local state* of p given K is the structure X plus:

—a static, nullary symbol `me`, interpreted as p,

—a static, nullary symbol `myMail`, interpreted as $\texttt{Mailbox}_K(p)$,

—a static, unary function symbol `Display`, interpreted as $\texttt{Display}_K$, and

—a dynamic, nullary symbol `myDisplay` initially interpreted as `undef`.

Remark 7.12. The intuition behind this definition is as follows. The local state in which a proclet p operates is like the global state X of the computation, with the following exceptions. First, the proclet knows which proclet it is; this is reflected by the presence of `me` in its vocabulary. Second, proclet p can use the multiset of its own received messages, $\texttt{Mailbox}(p)$, accessed via `myMail`, but it cannot use the corresponding multisets of the other proclets. The idea here is that for a message to be available to p it should be sent to p. The technical point is that, if proclets could look into each other's mailboxes, that would introduce a circularity into the computation and the passing of messages, possibly making Theorem 7.22 below incorrect. For details about this point, see the proof of Theorem 7.22. Third, a proclet p can access, via $\texttt{Display}(q)$ the information displayed by another proclet q. The access is arranged very simply, by allowing the algorithm executed by p to use `Display` as a static function symbol. Fourth, a proclet p has a dynamic symbol `myDisplay` to which it can assign a value that other proclets can access using `Display`.

The computation of a proclet takes place in its local state given a certain "correct" ken. The meaning of "correct" here depends on interaction between proclets and will therefore be specified only in the next subsection. The algorithm of the proclet, however, makes sense in its local state for any ken, not only the correct one. So, without relying on the notion of a correct ken, we can formulate the following postulate describing what each individual proclet's algorithm is to do. Note that the postulate does not refer to kens at all; they are not logically needed here. But we expect that it will be helpful to the intuition to think of the local state in the postulate as given by a ken.

—**Proclets:** Each element of `Proclet` is (or represents) a proclet that performs, at each step, a calculation of the following sort. The local state in which

[3]The ordinary meaning of "ken" is range of perception or knowledge. We use it for all the information that can be seen in the computation at this step.

proclet p works is the current global state X of the overall computation plus:
—a static, nullary symbol `me` denoting p,
—a static, nullary symbol `myMail` denoting some multiset,
—a static, unary function symbol `Display`, and
—a dynamic, nullary symbol `myDisplay`, with initial value `undef`.
All proclets execute the same algorithm, which is a sequential algorithm with output, in the sense of Definition 2.1. The outputs of this algorithm are ordered pairs ⟨addressee, content⟩ of elements of the state.

Several remarks are in order, to explain or justify aspects of this axiom.

Remark 7.13. As indicated in Section 5, the function symbols `me`, `myMail`, `myDisplay`, and `Display` are used for internal communication by proclets in one step and are not part of the global vocabulary. Their values do not persist from one state to the next. If, by accident, any of these symbols happen to be used in the global vocabulary, then they should be renamed (or other, new symbols should be used in the individual proclets' local states). Similarly, the symbol `Mailbox` should not be in the vocabulary, to avoid confusion with its use in our discussion.

Remark 7.14. The dynamic symbol `myDisplay` in the algorithm of a proclet will behave differently from the dynamic symbols in the vocabulary of the overall algorithm. Detailed information about this will be contained in the axioms in the next subsection, but we give an informal indication now to help the reader's intuition. The values of dynamic symbols of the overall vocabulary are changed only in the transition from one global state X to the next state $\tau_A(X)$. The value of `myDisplay`, in contrast, is used for inter-proclet communication within a step, that is, it is used for the computation of the update set $\Delta(A, X)$ that determines $\tau_A(X)$. Thus, `myDisplay` and also the (formally static) symbol `myMail` will acquire new values during a step, not at the completion of the step.

A brief explanation is in order for the apparent oxymoron of a new value for a static symbol `myMail`. The point is that the value of `myMail` in the state of a proclet p—the value of `Mailbox`(p), is determined by messages pushed to p by other proclets, not by updates executed by p. Thus, we may view this value as being empty at the start of a step and gradually growing as messages arrive, so it gets a new value. Yet, in the program executed by p there are no updates of `myMail`, so this program treats it as static.

A good intuitive picture of what happens within a step is that each proclet waits until it has received all the information from other proclets (either pushed by those proclets or pulled by itself) for that step and then executes its program, thereby possibly pushing information to other proclets and displaying information for them to pull. For this to work, it is obviously necessary to avoid deadlocks, where two or more proclets are waiting for information from each other; the next postulate will take care of this issue.

Remark 7.15. As explained earlier, we use "proclet" to mean subprocesses so small that they no longer involve unbounded parallelism; otherwise we would resolve them into lower-level processes. So it is reasonable to subject

these proclets to the Bounded Exploration Postulate, which is contained in our requirement that the proclets' algorithm be a sequential algorithm with output.

Remark 7.16. According to the Proclets Postulate, each proclet has available to it the whole global state of the algorithm. But nothing in the postulate requires a proclet to actually work with such a big state. In most realistic situations, each proclet will work with only a tiny part of the global state. We formulated the axiom to allow maximum freedom for the proclets' algorithm and thus to increase the domain of applicability of our main theorem.

Also, in many realistic algorithms, some proclets display information only to some, not all, of the other proclets. We need not build such access restrictions into our description because they can be incorporated into the algorithm executed by the proclets. If proclet p should not have access to the display of proclet q, then the proclets' algorithm can be so arranged that, when executed in any local state where `me` denotes p, it never uses `Display`(q).

Remark 7.17. We already discussed, in Section 3, the idea that all proclets should execute the same algorithm. To recapitulate: If there are several different algorithms to be executed by different sorts of proclets, then these can be combined into a single algorithm consisting of many cases, the cases being guarded by tests of which sort `me` belongs to.

Thus, the single algorithm mentioned in the Proclets Postulate will ordinarily be an `if`-chain: it can be given by a program of the form `if ... then ... elseif ... then ... elseif ... then ..... else ....` So each individual proclet would execute only one of the many clauses in such a chain.

If the algorithms to be executed by the proclets are produced by the computation itself (so there is not a fixed, finite set of them), then these programs produced by the computation are part of the state, and the proclets actually execute an interpreter on these. As noted earlier, this situation can arise only in trivial situations, because a proclet's algorithm is subject to Bounded Exploration.

7.4 Interaction Between Proclets

The remaining two postulates describe how proclets interact and how they produce the update set leading to the next state of the overall algorithm. They also specify the notion of "correct ken" alluded to above. Before stating these postulates, we give some motivation; more detailed comments and explanations are in the remarks following the postulates.

The essential idea, already described in Sections 4 and 5, is that proclets communicate by pushing and pulling information, that is, by sending messages to other proclets and by reading information displayed by other proclets. The picture in those earlier sections involved a hierarchy of proclets, in which information is transmitted from lower levels to higher ones. The height of the hierarchy is bounded, because of our assumption of bounded sequentiality. The next postulate, called Bounded Sequentiality, is intended to capture this idea in considerable generality. It does not require that the levels be explicitly definable

as part of the state, but it implies a hierarchy (see the proof of Theorem 7.22 for details).

The final postulate, called Updates, says what actually happens to the messages and updates produced by the individual proclets. The updates of global dynamic functions (i.e., all dynamic functions except `myDisplay`) go into the set $\Delta(A, X)$ of changes from the current state X to the next state $\tau_A(X)$. The messages go into the mailboxes of their addressees, still in the current step of the algorithm. An update of `myDisplay` by proclet p provides a value for $\texttt{Display}(p)$, which is used in other proclets' computations. If we think in terms of levels, this means that, in a single step of the algorithm, messages generated by proclets at the lowest level arrive in the mailboxes of proclets at the second level and thus influence the computations done by these proclets. Also, information displayed by proclets at the lowest level can be read by proclets at the next level, whose computations are thereby influenced. These computations may result, still in the same step, in messages sent to third-level proclets and displays read by third-level proclets, influencing their computations, and so forth (but for only a bounded number of levels). The Bounded Sequentiality Postulate will express this idea succinctly without explicit reference to levels.

This postulate will involve the relation "proclet q reads the display of proclet p," so we must give a definition of what this means. Since we don't assume that the proclets' algorithms are written in some traditional programming language, we cannot use syntactic criteria like "the program executed by proclets mentions $\texttt{Display}(t)$ for some term which, in the local state of q denotes p." Even if this were well-defined, it would often be wrong, as the occurrence of $\texttt{Display}(t)$ might be in a branch of a conditional rule and then might not actually be evaluated.

We use instead a semantic criterion, namely whether changing $\texttt{Display}(p)$ while leaving the rest of the ken unchanged would change the updates performed by q or the mailings it sends. This approach also has a flaw. It could happen, for some ken K, that there are proclets p_1 and p_2 with the following properties. If K_1 (resp. K_2) is exactly like K except for changing the value of $\texttt{Display}(p_1)$ (resp. $\texttt{Display}(p_2)$) then q computes the same updates and mailings in K_1 and in K_2 as it did in K, yet if K' is like K except for changing both $\texttt{Display}(p_1)$ and $\texttt{Display}(p_2)$, then q does something different. In this situation, we would say that, with respect to ken K, proclet q does not read the display of p_1 (for q's computation was unaffected by a change in this display) and does not read the display of p_2, yet somehow reads the combination of the two displays (for changing both displays alters q's computation).

This problem would not arise if we knew that q's computation is unaffected by changing only the value of $\texttt{Display}(p_1)$, regardless of what ken one begins with. Indeed, K' can be obtained from K_2 by such a change, and so q's computation would be the same in K' as in K_2, hence also as in K. Thus, there is a sensible notion of "q *might*, for some ken, read the display of p." That is, the notion of "reads for at least one ken" or "might read" is well-defined but there is no well-defined notion of "reads for a specific ken." Fortunately, it is precisely the notion of "might read" that we need, not the notion of "actually reads" with a specified ken. For details about this, see the proof of Theorem 7.22 below.

Definition 7.18. Let X be a (global) state and let p and q be proclets in X. Then q *pulls from* p if there are two kens for X, differing only in the values they give to `Display`(p), such that when q executes its algorithm in the local states determined by these kens, either it produces different updates (in the global state or in `myDisplay`) or it sends different mailings (where different multiplicities for the same mailing count as different mailings).

Definition 7.19. Let X be a (global) state. The *information flow digraph* of X is the directed graph that has the proclets of X as vertices and that has an edge from p to q if and only if either q pulls from p or, for some ken of X, p sends a message to q.

The intuition here is that an edge from p to q represents the possibility that information from p might (for some ken) flow to q.

With these definitions, we can state the next postulate.

—**Bounded Sequentiality:** For any (global) state X, there is a uniform bound B, depending only on the algorithm and not on the state, for the lengths of directed walks in the information flow digraph. In particular, this digraph is acyclic.

A walk in a digraph is a finite sequence of (not necessarily distinct) vertices such that there is an edge from each (except the last) to the next. The length of a walk is its length as a sequence, which is one more than the number of occurrences of edges in it.

For some (but not all) purposes, the Bounded Sequentiality Postulate can be weakened to the **Acyclicity Postulate** which requires that the information flow digraph be acyclic but does not impose any bound on the length of its walks. In particular, acyclicity suffices to prevent the sort of deadlock described earlier, where two or more proclets are waiting for information from each other.

Remark 7.20. The Bounded Sequentiality Postulate reflects our intention of describing a single step of the overall computation. Already in the case of sequential algorithms, a single step of an algorithm can involve a number of substeps; for example the evaluation of $g(f(c))$ involves the evaluation of $f(c)$ followed by the evaluation of g at this element. Furthermore a bounded number of steps, one after the other, can be regarded as a single step by an appropriate scaling. For example, given any abstract state machine, we can produce another such that two consecutive steps of the former are combined into one step of the latter. Similar speed-ups are well known for Turing machines and other computation models.

But to compress an unbounded number of steps into one (unbounded speed-up) would go beyond the concept of "step" considered in this article. To make precise the concept of bounded sequentiality, as described in the introduction, we require an *a priori* bound on the lengths of sequences of events within one step that must occur in a specified order. And walks in the information-flow digraph are such sequences of events. The earlier proclets in the walk must act before the later ones, because the messages produced by the former and

the information they display might influence the interpretation of `myMail` and `Display` in the computation of the latter.

A possible objection here concerns our definition of the graph as having an edge from p to q if, for some ken K, p pushes to q or q pulls from p, that is, if information from p *might under some circumstances* reach q. Why not put an edge from p to q only if p actually sends a message to q or q actually reads the display of p; that is, why not use only the actual "correct" ken rather than all possible ones? There are two reasons. One that we have already seen is that there is no clear notion of "q pulls from p with respect to a specific ken." The definition of "pulls from" works well only with respect to all kens at once. The second reason is that the acyclicity required by the Bounded Sequentiality Postulate is, as we shall see in the proof of Theorem 7.22 below, used to obtain a well-defined "correct" ken. Thus, the proposed weakening of the Bounded Exploration Postulate would refer to a ken whose existence is not yet ensured. For further discussion of this point, see Section 12.

Remark 7.21. It may seem that the Bounded Sequentiality Postulate could (and should) be weakened to allow the following sort of situation, while preserving its intuitive content. Suppose, for example, that there are two proclets p and q, such that p sends a message m_1 to q, and then q (using m_1) sends a message m_2 back to p, which then completes its computation using m_2. The information flow digraph then contains a directed cycle of length 2, which the postulate forbids, yet intuitively there is nothing wrong with this sort of computation. In such a situation, we prefer to distinguish, as distinct proclets, the two incarnations of p, namely one which, with empty mailbox, computes m_1 and sends it, and a second which, with mailbox containing m_2, completes the calculation.

One motivation for our preference is that each proclet now works with a single mailbox, rather than having its value of `myMail` change in the middle of the step. Another motivation is that it allows us to consider only the digraph, telling where information flows, rather than having to take into account the temporal ordering of pushing and pulling operations. In other words, our convention makes the Bounded Sequentiality Postulate considerably simpler than it would otherwise be.

This splitting of what might appear to be one proclet into two (or more) may become more intuitive if we recall that proclets were introduced simply as parts of the overall computation. There is a tendency to personify them, and then the splitting can seem excessively violent. But if we refrain from personification, the splitting becomes a reasonable convenience.

We must, however, address one potential problem with our convention. Suppose, in the example above, there were a third proclet r, sending a message m_3 to p. Since we regard the two incarnations of p as separate proclets, say p_1 and p_2, we must decide which one is to be the addressee for m_3. More precisely, this decision must be made in the program of the overall algorithm, telling all processes (in particular r) what messages to send and to whom. But is the necessary timing information (like "does p need m_3 for its computation of m_1?") available for use in the overall algorithm? We claim that it must be since we

deal with deterministic computation. Indeed, even if we regarded p_1 and p_2 as a single proclet whose mailbox changes (acquiring m_2 part way through the computation), it would be necessary to specify whether m_3 arrives before or after m_2 if this affects the computation. That specification suffices to tell us whether m_3 should be sent to p_1 or to p_2 when we regard these as separate proclets.

We are now in a position to establish, for each global state X, the existence of a unique "correct" ken. The conclusion of the following theorem spells out what we mean by "correct."

THEOREM 7.22. *For each state X, there exists a unique ken K with the following properties. For each proclet p,* $\texttt{Mailbox}_K(p)$ *consists exactly of the messages m sent to p by other proclets q executing their algorithm in the local state given by K. The multiplicity of m in* $\texttt{Mailbox}_K(p)$ *is the sum, over all proclets q, of the multiplicity with which q sends m to p. If the algorithm of a proclet p in the local state given by K produces a single update of* `myDisplay` *(possibly with some multiplicity), then the value given by this update is* $\texttt{Display}_K(p)$*; if it produces different updates or no update of* `myDisplay`*, then* $\texttt{Display}_K(p) = \texttt{undef}$.

Remark 7.23. For the proof of this theorem, the Acyclicity Postulate suffices in place of the Bounded Exploration Postulate.

Remark 7.24. The theorem can be viewed as a fixed-point description of the desired K. Given any ken K_0, we could let the proclets run in the local states given by K_0. The messages they send would determine a `Mailbox` function, collecting all messages sent to any proclet, and their updates of `myDisplay` would determine a `Display` function. These functions amount to another ken K_1. Intuitively, this means that "if the ken had been K_0 and the proclets had computed accordingly, then the ken should have been K_1." The theorem says that the process $K_0 \mapsto K_1$ has a unique fixed point. In other words, there is a unique *correct* ken K, where "correct" means that "if the ken had been K and the proclets had computed accordingly, then the ken should have been K."

We need a lemma for the proof of Theorem 7.22.

LEMMA 7.25. *Let p be a proclet in a global state X, and let K and K' be kens for X. Assume that* $\texttt{Mailbox}_K(p) = \texttt{Mailbox}_{K'}(p)$ *and that, for every q from which p pulls,* $\texttt{Display}_K(q) = \texttt{Display}_{K'}(q)$*. Then p produces the same updates (both global and of* `myDisplay`*) and the same mailings in the local states of p given by K and K'.*

PROOF. The local states of p given by K and K' have the same values for all functions of the global vocabulary (as K and K' are both kens for the same global state X), for `me` (trivially), and for `myMail` (as $\texttt{Mailbox}_K(p) = \texttt{Mailbox}_{K'}(p)$); they also agree about the initial value of `myDisplay` (namely `undef`). So the only difference between these two local states is in the possibly different values of $\texttt{Display}_K(q)$ and $\texttt{Display}_{K'}(q)$ for proclets q from which p does not pull.

According to Definition 7.18, the computation of p is unchanged if we alter its local state by changing $\texttt{Display}_K(q)$ for one proclet q from which p does not pull. By repeatedly applying this fact, we find that the computation of p

is also unchanged if we alter its local state by changing $\texttt{Display}_K(q)$ for any finite number of proclets q from which p does not pull. (Here we make essential use of the fact that "pulls from" was defined using all kens, not a specific ken.) Since there are only finitely many proclets altogether, the computation of p remains unchanged if we alter its local state by changing $\texttt{Display}_K(q)$ for any or all proclets q from which p does not pull. But, as observed above, this sort of alteration of the local state accounts for all of the difference between K and K' as far as p is concerned. □

Remark 7.26. Although real algorithms involve only finitely many proclets, as required in our Background Postulate, and although we used this finiteness in the preceding proof, it may be of interest for theoretical purposes to point out that the lemma remains true even if there are infinitely many proclets. The reason for this is based on the Bounded Exploration property of the proclets' algorithm. Using the notation of the proof of the lemma, we can argue as follows. It follows from Bounded Exploration, applied to the algorithm executed by p, that there is a finite set F of proclets such that p's computation under the ken K is unchanged if we change, in any way we like, the values of $\texttt{Display}_K(q)$ for all q not in F. In particular, we can change these values so as to coincide with those of $\texttt{Display}_{K'}(q)$. But then the resulting ken K'' differs from K' only in the displays of finitely many proclets, from each of which p does not pull. Then, changing K'' to K' one display at a time, as in the proof of the lemma, we find that p's computation is the same for K' as for K'' and therefore the same as for K.

We are now ready to prove the existence and uniqueness theorem for correct kens.

PROOF OF THEOREM 7.22. Fix a global state X, and consider the associated information-passing digraph. Because it is acyclic, we can partition the nodes into levels $L(k)$ indexed by positive integers k, so that if there is an edge from $p \in L(k)$ to $q \in L(l)$, then $k < l$. For example, the level of a node p could be defined as the length of the longest walk ending at p.

Consider now the conditions on K in the statement of the theorem, that is, the requirements defining correctness of a ken K. They can be equivalently reformulated as follows, making explicit the role of the levels.

(1) If $p \in L(k)$ then $\texttt{Mailbox}_K(p)$ consists of the messages (with multiplicities) sent to p by proclets in $\bigcup_{j<k} L(j)$, computing in their local states as given by K.
(2) If $p \in L(k)$ then $\texttt{Display}_K(p)$ is the value assigned to `myDisplay` by p, computing in its local state as given by K, provided there is a unique such value, and `undef` otherwise.

Intuitively, if we knew the correct ken at levels $< k$, then (1) would tell us how to obtain the mailboxes of proclets at level k, and then (2) would tell us how to obtain their displays. So the correct ken can be defined by recursion on k. More formally, the argument is as follows.

The computations of proclets mentioned in (1) are, according to the lemma, determined by the restrictions $\texttt{Mailbox}_K \restriction \bigcup_{j<k} L(j)$ and $\texttt{Display}_K \restriction \bigcup_{j<k} L(j)$, a combination that we abbreviate as $K \restriction \bigcup_{j<k} L(j)$. Indeed, if $q \in \bigcup_{j<k} L(j)$ then, thanks to the definitions of levels and of the information-flow digraph, all proclets from which q pulls are also in $\bigcup_{j<k} L(j)$.

Furthermore, the computation of p mentioned in item (2) is, again by the lemma, entirely determined by $\texttt{Mailbox}_K(p)$ and $\texttt{Display}_K \restriction \bigcup_{j<k} L(j)$. Combining these observations and remembering that p was an arbitrary element of $L(k)$, we find that the requirements (1) and (2) together specify exactly what $K \restriction L(k)$ should be, given $K \restriction \bigcup_{j<k} L(j)$.

Thus, these requirements constitute a recursive definition, by recursion on k, of $K \restriction L(k)$. Such a recursion, of course, has a unique solution, so the theorem is proved. □

Definition 7.27. For any global state X, the ken K given by Theorem 7.22 is called the *correct* ken for X, and the local states of proclets given by K are called the *correct* local states. We write simply `Mailbox` and `Display`, omitting the subscript K when it is the correct ken.

Note that, for the correct ken and local states, $\texttt{Mailbox}(p)$ consists of exactly (with multiplicity) the messages sent to p by other proclets executing their algorithms in their local states, and $\texttt{Display}(p)$ is the unique value assigned by p to `myDisplay` when executing its algorithm in its local state (or `undef` if there isn't a unique such value). In other words, inter-proclet communication works, for the correct ken, exactly as specified in Sections 4 and 5.

Having the correct ken and local states available, we can now easily finish the job of describing the algorithm's transition function. This is the role of our final postulate.

—**Updates:** The update set $\Delta(A, X)$ of the algorithm A in state X is the set of all updates of global dynamic functions produced by proclets in the correct local states for global state X.

The following definition is intended to formalize our claim that the postulates accurately describe synchronous, parallel algorithms whose individual steps have bounded sequentiality, that is, the class of algorithms described informally in the introduction.

Definition 7.28. A *parallel algorithm* is an algorithm satisfying the Sequential Time, Abstract State, Background, Proclets, Bounded Sequentiality, and Updates Postulates.

8. EXAMPLES

To support our claim that our postulates apply to general sequential-time algorithms with bounded sequentiality in each step, we discuss in this section a diverse collection of familiar approaches to parallelism. For each of these approaches, we briefly indicate how it fits our description.

8.1 Parallel Random Access Machines

We consider parallel random access machines (PRAMs), as described in Karp and Ramachandran [1990, page 873]: "A PRAM consists of several independent sequential processors, each with its own private memory, communicating with one another through a global memory. In one unit of time, each processor can read one global or local memory location, execute a single RAM operation, and write into one global or local memory location." As always, we take the states of an algorithm to contain all that is needed to determine the future progress of the computation. Thus, the state of a PRAM should include (in addition to what is required by the Background Postulate)

—all the memory locations, both global and local,
—all the values (usually integers) that can be stored in memory locations,
—all the processors, and
—all the possible RAM instructions for the processors as well as the programs of the processors.

There should be static functions describing the organization of the PRAM, including the function assigning to an address (which is a value) the corresponding memory location, the function assigning to each processor its accumulator register, the function assigning to every processor its program, the function assigning to every program its first instruction, the function assigning to every program and instruction the next instruction, and so on. There should also be static functions for whatever arithmetical (or other) operations are allowed as primitive RAM operations of the processors. The contents of the registers are given by a dynamic function. There is also a dynamic function assigning to each processor the RAM instruction that it is to execute next.

Further details depend on whether concurrent writing is allowed. (Concurrent reading makes no essential difference.)

In the case of exclusive writing (EREW or CREW PRAMs), where it never occurs that two processors write to the same memory location at the same step, we can take the proclets to be just the processors. The algorithm executed by any processor simply tells it to traverse its program and execute the instructions it finds.

The case of concurrent writing is more complicated, in that, in addition to the processors, global memory locations should also be considered as proclets, since they may have to do some computing to resolve write conflicts. Instead of writing directly to global memory, a processor p that wants to write a value v to a global memory location l sends either v or (in the "priority" model for resolving concurrent writes) the pair $\langle p, v\rangle$ to the proclet l. What l does then depends on the desired scheme for resolving write conflicts. In the "common" model, where all processors seeking to write to a memory location must agree on the value to be written, proclet l simply extracts the unique v from its mailbox and updates the content function at l to that value. (If the mailbox gives l more than one value to be written, then l refuses to write anything, and perhaps crashes the whole computation.) In the "priority" model, l applies a more complicated static function to its mailbox, picking out the pair $\langle p, v\rangle$ with the p of highest priority,

and it updates the content function at l to the corresponding v. The "arbitrary" model is somewhat different in that it is non-deterministic, arbitrarily choosing one of the candidates to be written into location l. As indicated in Remark 2.5, non-determinism can be modeled by an external function, which in this case would be a unary function assigning to each memory location l one member of its mailbox, that is, one of the conflicting writes for location l. The proclet l then executes the write chosen by the external function.

We remark that our description of exclusive-write PRAMs fits the simple picture described in Section 3, where proclets only perform updates and do not communicate with each other. For concurrent-write PRAMs, there is communication, as described in Section 4, but only of a very basic sort. Information is only pushed, not pulled, and the bound B in the Bounded Sequentiality Postulate can be taken to be 2. Processors are proclets at level 1, and global memory locations are proclets at level 2.

8.2 Circuits

We first consider uniform Boolean circuits of bounded fan-in, as described in Karp and Ramachandran [1990, pp. 897–898]. Any computation by such circuits can be regarded as consisting of two parts. The first part is the construction, from the length of the input, of the appropriate circuit. Uniformity means that this construction can be performed by a sequential algorithm (usually in log space). Since our interest in this article is in parallel algorithms, we shall ignore this part of the computation and concentrate on the second part, where the nodes of the circuit perform their computations.

We follow the usual and natural practice of regarding the depth of a circuit as a measure of the time required for (the second part of) the computation. Thus, we treat the computation of each level of a circuit as a separate step. This permits a very simple model of the computation. There is a proclet for each node. Static functions map each node to its immediate predecessors. Another static function assigns to each node the Boolean operation it is to perform. There is a dynamic function V assigning to each computation node the Boolean value that it computes (initially `undef` and updated just once during the computation), and assigning to each input and constant node its appropriate value. The algorithm executed by a proclet p begins by checking whether any of its immediate predecessors q have $V(q) = \texttt{undef}$. If so, p does nothing. If not, then p applies its Boolean operation to the $V(q)$'s and assigns this value to $V(p)$.

Notice that this description of circuits involves no communication between proclets within a step. It fits the simple description from Section 3.

There are a number of minor variations on this model. First, there is nothing special about *Boolean* values and operations; any other sort of values and appropriate operations on them could be used as well. Second, if the depths of the circuits are bounded, then we could regard the circuit's whole computation as a single step. Instead of (or in addition to) updating $V(p)$, proclet p would display this value, and the computation of any proclet would begin by pulling the displayed values of its predecessor nodes. Third, if the circuit model were non-uniform, then the first part of the computation, producing the

appropriate circuit from the input, would disappear, and the appropriate circuit would have to be given along with the input as part of the initial state. This would have no effect on the parallel part of the computation, the part modeled above.

A more serious modification of the model is needed if the circuits are allowed to have unbounded fan-in. In this situation, a proclet cannot "know" all of its immediate predecessors, that is, we cannot have static functions mapping each node to its immediate predecessors. If the fan-out is bounded, then there can be static functions assigning to each node p all the nodes q that will make use of the value computed by p. Then the algorithm executed by each proclet could push the value it computes to each of these other nodes.

Except in the case of circuits of bounded depth, Bounded Sequentiality requires that the computation proceed in several steps (rather than just one), most naturally one step for each level of the circuit. Since a proclet's immediate predecessors need not all be at the same level, some of the inputs for its Boolean operation may arrive before others. So we need a mechanism for storing them. A simple approach is to have a dynamic unary function `buffer` assigning to each proclet p the multiset (initially empty) of messages received so far. In addition, there should be a static function assigning to each p the number of messages it is to receive. At each computation step, a proclet p adds the newly received messages to its buffer,

```
buffer(me):=buffer(me)⊎myMail,
```

checks whether the new buffer has the total number of expected messages, and if so applies the appropriate Boolean operation and, in the next computation step, sends the result to its parent (or to output if p is the root). (The reason for waiting until the next computation step to forward the result of a node's computation to the parent node is simply to ensure Bounded Sequentiality—to make the "levels" of the circuit correspond to computation steps. Without the waiting, the whole circuit's computation would occur in a single step, and Bounded Sequentiality would be violated unless the depth of the circuit is bounded independently of the input.)

Still more modification is needed if both the fan-in and the fan-out can be unbounded. In this case, the natural model involves additional proclets, one for each edge in the circuit. The task of an edge proclet is to pull the value computed by its source node and push it to its target node. A node proclet proceeds almost as above. It waits until it has received the right number of inputs, then applies its Boolean operation to these, and in the next computation step displays the result (to be pulled by its outgoing edges).

8.3 Alternating Turing Machines

We use the description of alternating Turing machines given in Karp and Ramachandran [1990, page 901]; in particular, we assume for simplicity that an existential or universal configuration has just two successor configurations.

Before beginning our description of alternating Turing machines as parallel algorithms in our sense, we must comment on one difference between them and

the models considered in the preceding subsections. During the computation of an alternating Turing machine, new subcomputations are created, and these subcomputations will be elements of the state (proclets). We handle this situation by adopting the approach described in Gurevich [2000, Section 4.5]. Namely, we assume an infinite set of reserve elements, from which the environment selects one whenever a new element is needed. We refer to this as "importing" an element from the reserve.

As always, we include in our states everything relevant to the future progress of the computation. Specifically, the elements of our states include (in addition to what the Background Postulate requires)

—proclets, representing the subcomputations into which the main computation has branched,
—the symbols that can occur on the Turing tapes,
—the control states of the alternating Turing machine,
—the possible tape configurations, meaning what is written on the tape plus an indication of which cell is scanned by the read-write head, and
—infinitely many reserve elements.

The vocabulary includes (again in addition to what the Background Postulate requires) names for

—the dynamic unary relation `Proclet` that holds of exactly the proclets,
—the static nullary function `root` that is the main computation and is initially the only proclet,
—the dynamic function `active` assigning to each proclet the value `true` if it is computing and the value `false` if it has spawned subcomputations, and assigning `undef` to each non-proclet (initially, the only proclet `root` is active),
—for each tape symbol s and each element m of {left, right, stay} indicating a possible move of the read-write head, a static function from the set of tape configurations to itself, carrying out the corresponding Turing machine instruction "write s in the currently scanned cell and change which cell is scanned according to m,"
—a static function assigning to each tape configuration the symbol in the scanned cell,
—a dynamic function assigning to each proclet a tape configuration, initially assigning to the only proclet `root` the configuration in which the computation starts (ordinarily the input is written on the tape and the leftmost cell is scanned),
—a dynamic function assigning to each proclet its control state,
—two dynamic functions assigning to each inactive proclet its two immediate subcomputations and a dynamic function assigning to each proclet except `root` its parent (the initial values are all `undef` because `root` is the only proclet and it is active; the functions are dynamic only because the initial value `undef` can be changed to a "real" value; thereafter, the value no longer changes), and

—the dynamic function value assigning to each proclet its decision about acceptance or rejection, with true indicating acceptance, false indicating rejection, and undef indicating that the decision has not yet been made.

It is now easy to describe the algorithm followed by each proclet. We begin by considering active proclets. If the control state of such a proclet p is an ordinary state (i.e., not an accepting, rejecting, universal, or existential state), then p applies the appropriate update to its tape configuration and its control state, as determined, via the Turing machine instructions, from its current control state and the symbol written in the scanned square of its current tape configuration. If the control state is accepting or rejecting, then p updates value(p) to true or false accordingly. If the control state of p is universal or existential, then p imports two elements from reserve, makes them proclets (i.e., makes the predicate Proclet true of them), and initializes them as active, with configurations and control states as specified by the Turing machine instructions, with p as their parent, and with value initially undef; also, p updates its two subcomputation pointers (which were undef until now) to point to these new proclets, and p makes itself inactive. This completes the description of what an active proclet does. Before giving the description for inactive proclets, we note that when a proclet becomes inactive its control state is universal or existential. Inactive proclets will never change their control state, so we need to consider only these two cases.

An inactive proclet p with an existential control state looks at value(q) for its two immediate subcomputations q. If at least one is true then it sets value(p) to true. If both are false then it sets value(p) to false. Otherwise it does nothing.

An inactive proclet with a universal control state works the same way with the roles of true and false interchanged.

This completes the explanation of how alternating Turing machines fit our description of parallel algorithms. Notice that there is no communication between proclets, so the description in Section 3 suffices here.

Remark 8.1. It seems awkward to have entire tape configurations as single elements of the state. Contrast this with the standard ASM model of ordinary (sequential) Turing machines [Gurevich 2000, Example 4.3.1], where the content of the tape is given by a dynamic unary function from cells to symbols. The awkwardness seems, however, to be necessary for modeling alternating Turing machines at their natural level of abstraction; in other words, it seems to be inherent in the standard notion of alternating Turing machine. The point is that, when a computation enters an existential or universal state and spawns two subcomputations, it must pass (copies of) its tape configuration to these subcomputations. To formalize this without having the tape configurations as elements (and having pointers from computations to their configurations), we would have to include explicit instructions for how to copy the configuration and make the copies available to the subcomputation. Such instructions could certainly be given, but they are not part of the notion of alternating Turing machine; they are at a lower level of abstraction.

8.4 First-Order and Fixed-Point Logic

As already indicated with a simple example in Section 4, the evaluation of a first-order sentence in a given structure can be viewed as a parallel algorithm in our sense. We consider first the situation, as in that example, where one fixed sentence is to be evaluated in various structures. That is, the structure is given as input to the algorithm but the sentence is fixed. (Afterward, we shall consider the more general situation where the sentence is also part of the input.)

In the example, there was a proclet for each pair $\langle \varphi(\bar{x}), \bar{a} \rangle$, where $\varphi(\bar{x})$ is a subformula of the given sentence and $\bar{a}$ is a tuple of values for the free variables $\bar{x}$ of φ. In general, the situation is slightly more complicated for two reasons. First, the same subformula can occur several times in the given formula, and it will be convenient to treat those occurrences separately. So the first component of a proclet $\langle \varphi(\bar{x}), \bar{a} \rangle$ should be an occurrence of a subformula. Second, it is convenient for bookkeeping purposes to have all of a formula's free variables also free in its subformulas, but this is not generally the case. Accordingly, we define a variable v to be *pseudo-free* in an occurrence of a subformula φ of our given sentence if φ lies in the scope of a quantifier binding v. Since no variable is free in the whole sentence, every free variable of φ is necessarily pseudo-free. Intuitively, the definition of pseudo-free means that v could be free at the location of φ without destroying the fact that the whole sentence has no free variables. We also assume for simplicity that no variable is bound twice in the given sentence; it is easy to rename bound variables so as to satisfy this assumption. Alternatively, we could permit multiple binding of variables, at the cost of replacing variables in the following discussion with tagged variables, consisting of a variable together with an indication of which quantifier binds it.

Now we take the proclets to be the pairs $\langle \varphi(\bar{x}), \bar{a} \rangle$ where φ is an occurrence of a subformula in our given sentence and where $\bar{a}$ is a tuple of values for its pseudo-free variables $\bar{x}$. The initial state consists of the structure in which the formula is to be evaluated and the proclets, with static functions assigning to each proclet $\langle \varphi(\bar{x}), \bar{a} \rangle$ enough information to determine the main connective or quantifier in $\varphi(\bar{x})$ (or the information that $\varphi(\bar{x})$ is atomic and which atomic formula it is), the immediate superformula if any, and the elements $\bar{a}$. For atomic φ, the proclet simply evaluates the truth value by looking at the structure and pushes this truth value to its parent formula. If φ is a propositional combination of one or two subformulas, it gets their truth values from its mailbox, applies the appropriate Boolean operation, and either pushes the resulting truth value to its parent or, if φ is the whole sentence so there is no parent, makes the computed truth value the output of the whole computation. Similarly for quantified formulas, the appropriate Boolean operation now being one applied to the whole multiset of incoming messages; the two relevant operations, one for existential quantification and one for universal, are static functions (in accordance with the principle that the state contains everything relevant for the computation).

Remark 8.2. In the evaluation of a first-order sentence, unbounded parallelism arises only from quantification. Nevertheless, we have treated propositional combinations and quantifications alike, for the sake of simplicity in the description of the algorithm. We could sacrifice some simplicity for the

sake of having fewer proclets. In particular, it would suffice to have proclets $\langle \varphi(\bar{x}), \bar{a} \rangle$ only for those φ that are not used as an input to a propositional operation. The computation of such a proclet would in general involve a combination of independent (i.e., unnested) quantifier computations (as above) followed by propositional operations (also as above).

The preceding algorithm evaluates a fixed first-order sentence in one step, consisting of a number of phases. (Here and below, we use "phase" to refer, in an informal way, to the consecutive parts in a single step. Phases can be thought of as corresponding to the levels of the information-flow digraph or to the steps of the recursion in the proof of Theorem 7.22. In the present example, they also correspond to the depth of nesting of subformulas in a proclet's first component.) The bound B in the Bounded Sequentiality Postulate would be the depth of nesting of subformulas. (For the variant in the remark, B would be the depth of nesting of quantifiers.) For an algorithm that evaluates arbitrary first-order formulas instead of just a fixed one, the preceding approach would violate Bounded Sequentiality, though it would satisfy Acyclicity. To obtain a parallel algorithm in our sense, we should make the phases into separate steps. The modifications needed in the algorithm are fortunately rather slight. Instead of having each proclet p immediately push to its parent the truth value that it computes, let it store this truth value, say as $\texttt{value}(p)$ for a unary function `value` in the global vocabulary, and push it to its parent in the next computation step. Because of this delay, a proclet (for any non-atomic formula) should not begin its computation until it has checked that all the truth values it needs have been pushed to it. For a proclet computing a unary or binary Boolean operation, this means checking that its mailbox contains one or two (not necessarily distinct) elements, respectively. For a proclet computing a quantification, it means checking that its mailbox is nonempty; since all the subcomputations take the same time, if one has delivered a result then they all have.

Fixed-point logic can be handled similarly except for two things. First, the evaluation of even a single subformula involves unbounded sequentiality, so to fit our description it should be regarded as many steps, not as many phases in one step. Second, subformulas can involve predicate symbols that are not in the input sentence and are not interpreted in the structure where this sentence is to be evaluated. Namely, any fixed-point formula of the form $\mathrm{FP}_{P,\bar{x}}(\varphi(P, \bar{x}, \bar{y}))(\bar{t})$ has a subformula φ with the extra predicate P. For brevity, we call such extra predicates, which enter the formula by being bound by a fixed-point operator, *bound predicates*. Each of our proclets will have to have not only a formula and values for its pseudo-free variables (as in the discussion of first-order logic above) but also the values of any bound predicates occurring in its formula. These bound predicates will be given by unary functions mapping the proclet to the appropriate sets of tuples. Fortunately, tuples and sets (obtained from ordered pairs and multisets, respectively) are present in our states anyway in order to satisfy the Background Postulate.

Given these modifications, the proclets corresponding to atomic formulas, propositional combinations, and quantifications can compute just as in the

first-order case. (For atomic formulas involving bound predicates, those predicates, accessed via functions from the proclet, are used in the same way as the predicates of the given structure are used in the first-order situation.) It remains to describe how the proclet corresponding to a fixed-point formula operates. For definiteness, we work with the inflationary fixed-point operator. (See the next paragraph, or for a general treatment of fixed-point operators see [Dawar and Gurevich 2001].) Also, for brevity, we omit mention of free variables and values for them.

The proclet p for $\mathrm{IFP}_{P,\bar{x}}(\varphi(P,\bar{x}))(\bar{t})$ seeks to compute the successive iterates Φ^n of the inflationary operator defined by φ. Here $\Phi^0 = \varnothing$ and (in a somewhat simplified notation)

$$\Phi^{n+1} = \Phi^n \cup \{\bar{a} : \varphi(\Phi^n, \bar{a})\}.$$

Proclet p will keep track (in a location $\mathtt{temp}(p)$) of the most recently computed Φ^n, in order to compare it with the next Φ^{n+1} and so to determine whether the fixed point has been reached. The value of $\mathtt{temp}(p)$ is initially $\varnothing$, and p gives this (either by displaying it for pulling or by storing it in the global state, depending on how one wants to count steps) as the value of the bound predicate P to the proclets for the subformula φ with all possible values for $\bar{x}$. (If there were pseudo-free variables in φ or other bound predicates, they would be assigned the same values that they have for p.) When all these proclets finish their computations, they inform p by pushing to p pairs $\langle \bar{a}, v \rangle$ consisting of their value $\bar{a}$ for $\bar{x}$ and the computed truth value. Then p checks whether any $\bar{a}$ not in $\mathtt{temp}(p)$ (which at this stage is any $\bar{a}$ at all as $\mathtt{temp}(p)$ is empty) is associated with $v = \mathtt{true}$. If not, then the fixed point has been reached and p can evaluate it at $\bar{t}$ (which at this stage will certainly yield $\mathtt{false}$). Otherwise, p updates $\mathtt{temp}(p)$ by

$$\mathtt{temp}(p) := \mathtt{temp}(p) \uplus \{\!\{\mathtt{first}(x) : x \in \mathtt{myMail} : \mathtt{second}(x) = \mathtt{true}\}\!\}.$$

Then, p reactivates the proclets for φ with this new value of $\mathtt{temp}(p)$ as their value of P, and it repeats the same process until the fixed point is reached. (The update of $\mathtt{temp}(p)$ may produce a multiset rather than a set, but this makes no difference since any multiplicities here are not used.)

8.5 Abstract State Machines

We show that ASMs, which we shall use to simulate arbitrary parallel algorithms, are themselves parallel algorithms in our sense. Thus, our main theorem will provide a reduction from the class of parallel algorithms to a special subclass, not to some external and possibly stronger class. We refer to Section 9 below for the definition of ASMs used in this paper (very similar to the definition in Gurevich [1995]).

Consider an arbitrary ASM program Π, and assume that no variable is bound twice in Π; this assumption involves no loss of generality, since bound variables can be renamed. We intend to describe the operation of Π in a way that makes

it clear that all our postulates are satisfied. For expository purposes, we begin with a description that is incorrect in three ways but (we hope) makes the ideas clear. We shall point out the three difficulties when they arise in our description, and afterward we shall show how to correct them.

We assume that Π never produces inconsistent updates. (This is the first of the three difficulties.)

Let every occurrence of a term t or rule R in Π be represented by an element of the state and named by a (variable-free) term $\hat{t}$ or $\hat{R}$. This is easy to do; for example, enumerate all these occurrences in some arbitrary order and represent the nth one by $\{\!\{\varnothing, \ldots, \varnothing\}\!\}$ with n occurrences of $\varnothing$. Note that we are using t and R here to stand for occurrences of terms and rules. If the same term occurs several times in Π then it has several representatives $\hat{t}$. To simplify terminology, we will sometimes gloss over the distinction between terms (or rules) and their occurrences. Thus, we may write "t is a variable" when we really mean "t is an occurrence of a variable."

We take as proclets all pairs of the form $\langle \hat{t}, \bar{a} \rangle$ or $\langle \hat{R}, \bar{a} \rangle$, where t (or R) is an occurrence of a term (or rule) in Π and $\bar{a}$ is a tuple of values for all the pseudo-free variables of t (or R). (This is the second difficulty; since there are infinitely many possible values for a variable, we have infinitely many proclets, contrary to the finiteness requirement in the Background Postulate.) Here "pseudo-free" is defined as in Section 8.4.

Our algorithm will need to refer to the notion of the "parent" of a proclet. Here the parent of $\langle t, \bar{a} \rangle$ or $\langle R, \bar{a} \rangle$ is the proclet whose first component t' or R' is the term or rule having t or R as an immediate constituent and whose second component is either $\bar{a}$ if t' or R' doesn't bind a variable or, if it does bind a variable x, then $\bar{a}$ with the value for x omitted. Since Π is finite, the parent function, though perhaps complicated, is explicitly definable in the state of the ASM. We refrain from writing out the definition, but when we refer to "parents" we mean to use this definition.

Our algorithm will also refer to the notion of an "active" proclet. This notion is temporary in the same sense as `Mailbox` and `Display`; it does not persist from one state of the algorithm to the next. A proclet can "become" active in the middle of a step, when it pulls certain information from its parent or receives certain messages in its mailbox.

The task of a term-proclet $\langle \hat{t}, \bar{a} \rangle$, when it is active, will be to compute the value of t in the current state, using the values in $\bar{a}$ for its free variables, and push this value to its parent. More precisely, it should label the pushed value with its own identity, so that the parent knows where the value came from. So if the value computed by $\langle \hat{t}, \bar{a} \rangle$ is v then it should push $\langle v, \langle \hat{t}, \bar{a} \rangle \rangle$.

The task of a rule-proclet $\langle \hat{R}, \bar{a} \rangle$, when it is active, is to ensure the execution of the updates that rule R produces with values from $\bar{a}$ for its free variables. The rule-proclet may not (indeed often cannot, because of Bounded Exploration) execute all these updates on its own, but then it must activate enough of its children to ensure execution of the correct updates.

We now describe how proclets carry out their tasks.

An active term-proclet of the form $\langle \hat{x}, \bar{a} \rangle$ with x a variable reads the value of x from $\bar{a}$ and passes this to its parent.

An active term-proclet of the form $\langle \hat{t}, \bar{a} \rangle$ with t of the form $f(t_1, \ldots, t_n)$ does the following.

(1) Activate (by displaying an appropriate signal) the proclets $\langle \hat{t}_i, \bar{a} \rangle$ for $1 \leq i \leq n$.
(2) After receiving the values computed by these proclets, apply f to these values, and push the result to your parent.

(The third difficulty is visible here: Information must flow from this proclet to its children to activate the children, and then information must return from the children, in the form of the values they computed. So the information-flow digraph has cycles of length 2, contrary to the Bounded Sequentiality Postulate.)

An active term-proclet of the form $\langle \hat{t}, \bar{a} \rangle$ with t of the form $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ does the following.

(1) Activate the proclet $\langle \hat{r}, \bar{a} \rangle$.
(2) After receiving the value b that this child computes for r, display it as a signal activating all the proclets $\langle \widehat{t(x)}, \bar{a}^\frown c \rangle$ and $\langle \widehat{\varphi(x)}, \bar{a}^\frown c \rangle$ for $c \in b$. Also, remember b, in particular the multiplicities $\mathrm{Mult}(c, b)$ for future use.
(3) After receiving the values of t and φ for the various c's, collect the values of t for which the corresponding value of φ is `true` into a multiset, using the remembered b to get the multiplicities right. Push this multiset to your parent.

(The third difficulty is amplified here, and it will recur in the instructions for rule-proclets, but the essence of the difficulty remains the same: 2-cycles between a proclet and its children.)

An active rule-proclet of the form $\langle \hat{R}, \bar{a} \rangle$ with R an update rule, say $f(t_1, \ldots, t_n) := t_0$, does the following.

(1) Activate the proclets $\langle \hat{t}_i, \bar{a} \rangle$ for $0 \leq i \leq n$.
(2) After receiving the values b_i computed by these proclets, update $f(b_1, \ldots, b_n)$ to b_0.

An active rule-proclet of the form $\langle \hat{R}, \bar{a} \rangle$ with R a conditional rule, say `if` φ `then` R_0 `else` R_1 `endif`, does the following.

(1) Activate $\langle \hat{\varphi}, \bar{a} \rangle$.
(2) If the returned value is `true` then activate $\langle \hat{R}_0, \bar{a} \rangle$; if the returned value is `false` then activate $\langle \hat{R}_1, \bar{a} \rangle$.

An active rule-proclet of the form $\langle \hat{R}, \bar{a} \rangle$ with R a parallel rule, say `do forall` $x \in r$, $R_0(x)$ `enddo`, does the following.

(1) Activate $\langle \hat{r}, \bar{a} \rangle$.
(2) After receiving the value b that this child computes for r, display it as a signal activating all the proclets $\langle \widehat{R_0(x)}, \bar{a}^\frown c \rangle$ for $c \in b$.

This completes our first, triply incorrect description of the operation of Π as a parallel algorithm in our sense. It remains to correct the three difficulties.

First difficulty: We ignored the possibility of clashes: of updates that attempt to give two different values for the same location. Our notion of algorithm made no provision for this, but in ASMs there is a standard convention that, in case of clash, the transition function does nothing; the next state is the same as the current one. So the two models seem to disagree. Fortunately, every ASM is equivalent to one that cannot produce clashes. See Section 9.2 below for a proof. Thus, we can avoid the first difficulty by replacing the given ASM program Π by a clash-free equivalent Π' and then applying the description above to Π'.

Alternatively, we can avoid the normal form and work directly with the given ASM program Π, at the cost of some extra complexity in the description above. Add to the state a second copy of each rule-proclet corresponding to an update rule. The old rule-proclets are no longer responsible for getting their updates executed but instead collect their updates into a multiset, which they pass to their parents. In addition, each old update-rule-proclet pushes its update to the corresponding new proclet. Thus, the computation results eventually in the root proclet $\langle \Pi, \langle\rangle\rangle$ having the multiset of all the updates that would be executed were it not for possible clashes. This proclet can then check, by means of a static function, whether this multiset contains clashing updates. It displays the result of this check, to be pulled by the new copies of the update-rule-proclets. If there was a clash, these new proclets do nothing. If there was no clash then they execute the updates sent to them by the corresponding old update-rule-proclets.

Second difficulty: Infinitely many proclets. Of course there are only finitely many occurrences of terms and rules in Π, but any free variable has, *a priori*, infinitely many possible values, so we get infinitely many proclets. The solution to this is to redefine the notion of proclet so that there are only finitely many of them, yet they include all those proclets in the old sense that might possibly become active. Of course the Background Postulate demands that our new notion of proclet be given by a term. We therefore explain here how to produce such a term.

We first define a term $\mathtt{MDA}(p)$ whose value, for any proclet (in the old sense) p is the set of proclets that p **m**ight **d**irectly **a**ctivate. (More precisely, the term denotes a multiset containing these proclets with some multiplicities. At the end, we'll apply AsSet to get rid of the multiplicities.) The term is quite lengthy, but its construction is explained by the following clauses.

—If A is a variable v, then $\langle \hat{A}, \bar{a}\rangle = \varnothing$.

—If A is a term of the form $f(t_1, \ldots, t_n)$ then $\langle \hat{A}, \bar{a}\rangle$ is the sum of the singletons $\{\!\{\langle \hat{t}_i, \bar{a}\rangle\}\!\}$.

—If A is a comprehension term $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ then $\langle \hat{A}, \bar{a}\rangle$ is the sum of the singleton $\{\!\{\langle \hat{r}, \bar{a}\rangle\}\!\}$ and the two sets $\{\!\{\langle \hat{t}, \bar{a}^\frown c\rangle : c \in r : \mathtt{true}\}\!\}$ and $\{\!\{\langle \hat{\varphi}, \bar{a}^\frown c\rangle : c \in r : \mathtt{true}\}\!\}$.

—If A is an update rule $f(t_1, \ldots, t_n) := t_0$ then $\langle \hat{A}, \bar{a}\rangle$ is the sum of the singletons $\{\!\{\langle \hat{t}_i, \bar{a}\rangle\}\!\}$.

—If A is a conditional rule $\mathtt{if}\ \varphi\ \mathtt{then}\ R_0\ \mathtt{else}\ R_1\ \mathtt{endif}$ then $\langle \hat{A}, \bar{a}\rangle$ is the multiset (sum of three singletons) $\{\!\{\langle \hat{\varphi}, \bar{a}\rangle, \langle \hat{R}_0, \bar{a}\rangle, \langle \hat{R}_1, \bar{a}\rangle\}\!\}$.

—If A is a parallel rule `do forall` $x \in r$, $R_0(x)$ `enddo` then $\langle \hat{A}, \bar{a} \rangle$ is the sum of the singleton $\{\!\{\langle \hat{r}, \bar{a} \rangle\}\!\}$ and the multiset $\{\!\{\langle \hat{R}_0, \bar{a}^\frown c \rangle : c \in r : \mathtt{true}\}\!\}$.

Here $\bar{a}^\frown c$ means the sequence obtained from $\bar{a}$ by adjoining one more component c at the end. If sequences are coded in the obvious way by iterating the ordered pair operation, then $\bar{a}^\frown c = \langle \bar{a}, c \rangle$.

Next, we define another term $\mathtt{MA}(p)$ whose intended meaning is the set of proclets that p might activate, directly or indirectly. It is defined by induction on the first component (term or rule) of p. Notice that, since the program Π is finite, the whole recursion could be rewritten as an explicit definition, in which the recursive clauses are nested like the terms and rules in Π.

$$\mathtt{MA}(p) = \{\!\{p\}\!\} \uplus \bigl(\biguplus \{\!\{\mathtt{MA}(q) : q \in \mathtt{MDA}(p) : \mathtt{true}\}\!\}.$$

Finally, we define the new, finite set of proclets as $\mathrm{AsSet}(\mathtt{MA}(\langle \Pi, \langle\rangle \rangle))$.

Third difficulty: 2-cycles in the information-flow digraph. This problem was already discussed in Remark 7.21 and we adopt the solution proposed there. We replace each proclet by two or three proclets, each carrying out part of the computation of the old proclet. Specifically, where our description above of how a proclet performs its task used two or three numbered instructions, split the proclet into two or three proclets, each carrying out just one of the numbered steps. Also, where our description said to push a value to the parent, push it to the right incarnation of the parent, namely the incarnation corresponding to the instruction that receives the value.

Remark 8.3. Our correction of the second difficulty above required the use of AsSet to reduce the multiset $\mathtt{MA}(\langle \Pi, \langle\rangle \rangle)$ to a set, as required by the Background Postulate. Recall from Remark 7.5 that AsSet is the only one of our basic operations on multisets that lacks an obvious justification on the basis of being intuitively needed to deal with messages. One might therefore consider the possibility of avoiding the need for AsSet by allowing `Proclet` to be a multiset rather than a set. Unfortunately, this proposal leads to serious difficulties, both conceptual and technical.

If a proclet p is present with some multiplicity $m > 1$ in `Proclets`, should its actions, especially its pushing of messages, automatically be replicated m times, so that if the algorithm of p says to push a message to q then q will receive it m times? If so, then q, not knowing that these weren't m separate mailings from a single p, will act in response to all of them, producing a result quite different from what p intended. On the other hand, if q gets the message only once, then this reduction from m copies to one amounts to an application of AsSet, and an ASM simulation (in the proof of our main theorem) will need AsSet for this purpose.

Consider also our description above of how ASMs satisfy our postulates. In the absence of AsSet, a proclet p whose first component is a comprehension term $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ will activate proclets for $t(x)$, with various values of x, with multiplicities matching the multiplicity of x in r. At first sight, this looks good; it could even avoid the need for p to remember multiplicities (see part 2 of its instructions above). But now suppose that $t(x)$ is itself a comprehension

term. So the proclets p' activated by p (with multiplicities) will each activate further proclets p'', again with multiplicities. If p activates p' with multiplicity m and p' activates p'' with multiplicity m', then altogether p'' will be activated with multiplicity mm'. Now p'' is supposed to send to p' the value it computes. The mailbox of p' will contain this value mm' times. But the value computed by p'' should enter into the value computed by p' with multiplicity m', not mm', and there seems to be no way to extract this correct multiplicity from the mailbox of p'.

In summary, although we do not claim that it is impossible to avoid AsSet by allowing non-1 multiplicities in `Proclet`, we are convinced that the difficulties involved in doing so would outweigh any benefit from avoiding AsSet.

9. ABSTRACT STATE MACHINES

The main result to be proved in this article is that parallel algorithms, as defined by the postulates in Section 7, are equivalent to suitable abstract state machines (ASMs). In preparation for this result, we give in the present section a brief summary of the relevant definitions concerning ASMs. For more details, see Gurevich [1995]. Most of this material is a special case of what is in Gurevich [1995], the special case being defined by the presence, in all states, of the items required by the Background Postulate. The only deviations here from the definitions in Gurevich [1995] are that we use comprehension terms (see below) instead of quantifiers and that multisets serve (as sets did in Blass et al. [1999]) as the domains over which bound variables are allowed to range. We also provide a normal form for ASMs, which in particular allows us to replace any ASM with an equivalent one that never has conflicting updates. This fact is not needed for our main result, for there the ASM we construct automatically has no clashes, but it was used to simplify the discussion in Section 8.5 and we expect that it will be useful beyond the confines of this article.

9.1 Definition of ASMs

We begin with a vocabulary: a set of function symbols or names with specified arities, that includes all the items required by the Background Postulate. Some of the function symbols are declared to be relational; among these are the logic names (equality, the Boolean operations, `true`, and `false`). The values of the functions interpreting relational names in a structure are required to be in {`true`, `false`}. Also, some of the function symbols are declared to be static; among these are the logic names, `undef`, and the names required by the clauses of the Background Postulate concerning ordered pairs and multisets. Function symbols not declared to be static are called dynamic.

The collection of *terms* is defined recursively by:

—Every variable is a term.

—If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1 \ldots, t_n)$ is a term. If f is relational, then this is a Boolean-valued term.

—If $t(x)$, r, and $\varphi(x)$ are terms, if x is not free in r, and if $\varphi(x)$ is Boolean-valued, then $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ is a term.

Terms of the form $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ are called *comprehension terms*.

Free variables are defined in the obvious way, with $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ binding x. Semantically, when values have been assigned to the free variables, $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ is interpreted as the multiset in which the multiplicity of any object a is the sum, over all values of x for which the value of $t(x)$ is a and the value of $\varphi(x)$ is `true`, of the multiplicity of the value of x in the value of r. (If the value of r isn't a multiset, then this multiplicity is 0.) The remaining recursive clauses in the definition of the values of terms are just as in ordinary first-order logic. In the future, we shall omit "the value of" when the intended meaning is clear; thus we might write that the multiplicity of a in $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ is the sum, over all x for which $t(x) = a$ and $\varphi(x) =$ `true` of the multiplicity of x in r.

The collection of *rules* is defined recursively by:

—If f is an n-ary dynamic function symbol and $t_0, t_1, \ldots, t_n$ are terms, then

$$f(t_1, \ldots, t_n) := t_0$$

is a rule, called an *update rule*. It is required that, if f is relational, then t_0 is Boolean.

—If φ is a Boolean term and R_1 and R_2 are rules, then

$$\texttt{if } \varphi \texttt{ then } R_1 \texttt{ else } R_2 \texttt{ endif}$$

is a rule, called a *conditional rule*.

—If x is a variable, $R(x)$ is a rule, and r is a term in which x is not free, then

$$\texttt{do forall } x \in r,\ R(x) \texttt{ enddo}$$

is a rule, called a *parallel rule*.

Again, free variables are defined in the obvious way, with `do forall` $x \in r$, `R(x) enddo` binding x. In any structure, given values for the free variables, a rule is interpreted as a set of updates, changing the values of some dynamic functions at certain argument tuples. See Gurevich [1995] or the next subsection for details.

An *ASM program* is simply a rule with no free variables. It thus determines a set of updates in any structure of the appropriate vocabulary. An *ASM* consists of such a program together with a collection of states, which are among the structures for the appropriate vocabulary, and a subcollection of initial states, both collections being closed under isomorphisms. The transition function of the ASM applies to any state by executing the update set of the program; the collection of states is required to be closed under the transition function.

Since some of the ASM constructs used in Gurevich [1995] are not included in the present description, we indicate how they can be simulated in the present set-up. Bounded quantifiers[4] can be simulated because $\exists x \in r\ \varphi(x)$ is equivalent

[4]Recall that, despite the terminology, the range of the quantified variable in a bounded quantifier need not be (uniformly) bounded. The size of the set over which the variable ranges can increase with the input rather than being bounded by the algorithm alone. See Terminology 1.3.

to $\{\!\{x : x \in r : \varphi(x)\}\!\} \neq \varnothing$. A parallel combination of two rules, written

$$\texttt{do in parallel } R_0, R_1 \texttt{ enddo},$$

is equivalent to

```
do forall x ∈ {{true, false}}
   if x = true then R1 else R0 endif
enddo.
```

Repeated use of this construction gives larger `do in parallel` blocks. The empty block, also called `Skip`, is simulated by `do forall` $x \in \varnothing$, R `enddo` for any rule R. Finally, `if`-chains as in Remark 7.17 can be simulated by nested conditionals.

Notice that ASMs as defined here involve two forms of parallelism, namely the `do forall` construct and comprehension terms. Neither can be eliminated in favor of the other. Without `do forall`, the update set generated by any rule would be bounded. Without comprehension terms, there would be no way to compute the truth value of a formula like $(\forall x \in r)(\exists y \in r)\, P(x, y)$ in a single step (though it could be done in two steps).

9.2 Normal Form

A standard convention in the ASM literature, following Gurevich [1995], is that if a program produces two conflicting updates, i.e., two updates of the same location with different values, then no update at all is executed. This convention for handling clashes will play no role in the proof of our main result, Theorem 10.1, since the ASM constructed there will never produce clashes. In our verification in Section 8.5 that ASMs satisfy our postulates, we ignored clashes to simplify the discussion. We show here that this simplification causes no loss of generality. Every ASM is equivalent to one that never produces clashes.

The idea behind the construction of a clash-free ASM equivalent to a given one is to rewrite the program Π of the given ASM so that, instead of executing updates, it collects in a multiset the updates that it intends to execute. Then this multiset is checked for clashes. If a clash is found, nothing is done (thereby correctly simulating Π); otherwise, all the collected updates are executed.

To carry out this idea, we must represent updates by members of the state, so that they can be collected into multisets. For this purpose, we assume that we have associated to each dynamic function symbol f (in the vocabulary of Π) a term $\hat{f}$ with no free variables, to serve as a code for f. For example, we could list the function symbols in some order and then associate to the nth symbol the term $\{\!\{\varnothing, \ldots, \varnothing\}\!\}$ where there are n occurrences of $\varnothing$. We also agree to code ordered tuples of arbitrary length by means of nested ordered pairs. That is, we introduce the abbreviations

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle = \langle\langle x_1, \ldots, x_n\rangle, x_{n+1}\rangle$$

for all $n \geq 2$. The corresponding projection functions, extracting a component from a tuple, are given (for $n \geq 2$) by

$$P_1^n(x) = \texttt{first}^{n-1}(x) \quad \text{and} \quad P_k^n(x) = \texttt{second}(\texttt{first}^{n-k}(x)) \text{ for } k \geq 2.$$

Then $P_k^n(\langle x_1, \ldots, x_n\rangle) = x_k$. For the sake of uniformity, we identify a sequence of length 1 with its unique component; then the preceding equations are also correct for $n = 1$.

We shall use an $(n + 2)$-tuple of the form $\langle \hat{f}, \bar{a}, b\rangle$ to represent an update of the n-ary dynamic function f at argument n-tuple $\bar{a}$ to value b. With these conventions in place, we are ready to transform ASM programs into terms representing the multiset of updates that they perform (or, rather, that they would perform if no clashes occurred). The transformation proceeds by induction on rules, defining for each rule R a term U_R, with the same free variables as R, to represent the update multiset of R. (Since we have multisets in the background, we use them here, but the multiplicities really play no role; executing an update many times has the same effect on the state as executing it once.)

—If R is an update rule $f(t_1, \ldots, t_n) := t_0$ then U_R is

$$\{\!\{\langle \hat{f}, t_1, \ldots, t_n, t_0\rangle\}\!\}.$$

—If R is a conditional rule `if` φ `then` R_0 `else` R_1 `endif` then U_R is

$$\{\!\{x : x \in U_{R_0} : \varphi\}\!\} \uplus \{\!\{x : x \in U_{R_1} : \neg\varphi\}\!\}.$$

—If R is a parallel rule `do forall` $x \in r,\ R_0(x)$ `enddo` then U_R is

$$\biguplus \{\!\{U_{R_0(x)} : x \in r : \texttt{true}\}\!\}.$$

Our interest is, of course, in the update set U_Π corresponding to the whole program Π. To check it for clashes, we define a Boolean-valued term `Clash` as

$$(\exists x \in U_\Pi)(\exists y \in U_\Pi)\,[\texttt{first}(x) = \texttt{first}(y) \wedge \texttt{second}(x) \neq \texttt{second}(y)].$$

To understand this, remember that, under our coding of tuples, an update $\langle \hat{f}, \bar{a}, b\rangle$ is an ordered pair whose first component is $\langle \hat{f}, \bar{a}\rangle$ and whose second component is b. Remember also that we saw at the end of the preceding subsection that quantifiers like $\exists x \in U_\Pi$ are available in our ASMs.

We define Δ to be the term $\{\!\{x : x \in U_\Pi : \neg\texttt{Clash}\}\!\}$. It has the same value as U_Π if `Clash` is false, but it has value $\varnothing$ if `Clash` is true. So Δ denotes the set of updates that Π actually executes. It remains to write an ASM program to execute these updates, given Δ. The program has the form

$$\texttt{do forall } x \in \Delta,\ \texttt{Execute}(x) \texttt{ enddo}$$

where `Execute`(x) is an `if`-chain with one clause for each dynamic function symbol f. The clause for f, whose role is to execute updates of f, reads

$$\begin{aligned}&\texttt{(else)if } P_1^{n+2}(x) = \hat{f} \texttt{ then}\\ &\quad f\left(P_2^{n+2}(x), P_3^{n+2}(x), \ldots, P_{n+1}^{n+2}(x)\right) := P_{n+2}^{n+2}(x) \texttt{ endif}.\end{aligned}$$

Notice that the clash-free equivalent of Π that we have produced here involves just a single `do forall` applied to an `if`-chain of simple updates. The range of the variable in the `do forall` is, however, a very complicated multiset term. Most of the parallelism in Π may be buried in this term.

10. THE EQUIVALENCE THEOREM

The purpose of this section is to prove that parallel algorithms are equivalent to abstract state machines (ASMs) working with suitable backgrounds. Recall that we defined parallel algorithms to be algorithms that satisfy the postulates of Section 7.

Notice that Definition 2.3 of equivalence requires the simulating ASM to have the same states and thus to use the same background as the algorithm A being simulated. The Background Postulate thus ensures that a certain minimal background is available to the ASM whether A makes any real use of it or not.

THEOREM 10.1. *Every parallel algorithm is equivalent to an ASM working with the same background and with the language described in Section 9.*

PROOF. We begin by considering what each individual proclet does at any step of the given algorithm A. If p is a proclet in the current state X of A, then it executes an algorithm as described in the Proclets Postulate. So it works in a local state that differs from X in also having `me`, `myMail`, `myDisplay`, and `Display` (see Terminology 3.2). And it executes an algorithm that satisfies all the postulates of Gurevich [2000] with the following two modifications. First, an inessential point is that to satisfy the Abstract State Postulate, we must consider, simultaneously with state X and proclet p, all isomorphic copies of X with the corresponding proclets. ("Corresponding" means the image of p under an isomorphism.) We agree to consider all these states and proclets together, but we continue to use the notation X, p for simplicity.

The second modification is more significant but still minor. It is that proclets not only produce updates (as the algorithms in Gurevich [2000] did) but also send messages. That is, they are sequential algorithms with outputs, in the sense of Definition 2.1. Reading Gurevich [2000] with this point in mind, we find that our proclets execute algorithms that are just like the sequential ASMs of Gurevich except that they allow, among their rules, one more construct, namely `Output`(t). In the present situation, where the outputs are pairs ⟨addressee, content⟩, it is convenient to use, in place of `Output`$\langle t_1, t_0\rangle$, the notation

$$\texttt{Push}\ t_0\ \texttt{to}\ t_1\ \texttt{endpush},$$

where t_0 and t_1 are terms with no free variables. (Remember that all terms in Gurevich [2000] were ground terms.) The semantics of the new construct is, of course, that execution of this statement by p causes the value of t_0 to be sent as a message to the proclet denoted by t_1. (If the value of t_1 is not a proclet, then nothing happens.) If the algorithm executes several `Push` rules with the same values for t_0 and t_1, then this message is sent to this recipient with the corresponding multiplicity.

Notice that we are specializing the situation of Definition 2.1. There, outputs were simply produced and sent to an unspecified outside world. Here, when a proclet produces an output ⟨addressee, content⟩, this means that the content is sent as a message to the addressee.

It may be useful to briefly sketch the ideas from Gurevich [2000] that are used here and the modifications needed to accommodate messages. By enlarging the bounded exploration witness T given by the Proclets Postulate via the definition of "sequential algorithm with output," we can arrange that it is closed under subterms and contains `true` and `false`. Then all elements involved in any update produced by p (either as components of the location being updated or as the new value), all messages sent by p, and all the recipients of these messages must be among the values of the terms in T (as evaluated in the local state of p). The reason is that otherwise we could alter the state, without changing the values of terms in T, but so that the element in question is not in the base set of the new state. This would contradict the choice of T as a bounded exploration witness. (This argument is Lemma 6.2 of Gurevich [2000].) It follows that what the algorithm does in any single (local) state can be simulated by a single rule, a `do in parallel` combination of the updates made and the messages sent in that state (repeating `Push` rules as often as needed to achieve the right multiplicity of messages). The reason for this is that the necessary updates and messages can be written explicitly using terms from T. (See Corollary 6.6 of Gurevich [2000].) Furthermore, this rule, which *a priori* depends on the state, actually depends only on which terms from T have equal values in the state. The reason is that two states that satisfy the same equality relationships between terms from T have isomorphic copies that actually coincide on T. Applying the definition of T to the isomorphic copies, and then applying isomorphism invariance (as required by the Abstract State and Proclets Postulates) to transport the results back along the isomorphisms to the original states, we obtain the claimed result. (See Lemma 6.9 of Gurevich [2000].) Thus, even though there are infinitely many possible states, there are only finitely many rules needed to match the given algorithm in all of them. Combining these rules with an `if`-chain, using the equality relations between elements of T as guards, we get a single rule that matches the given algorithm in all states. (See Lemma 6.11 of Gurevich [2000].)

Let us write Π for this "sequential ASM with `Push`" describing the action of each individual proclet. Our remaining task is to transform Π into a genuine ASM program (without `Push`, no longer sequential, and using the vocabulary of the global state, not of the individual proclets) describing the action of all the proclets together. We begin by describing our general strategy for this task and then give the details.

A reasonable approach to simulating the work of the given algorithm A is suggested by the proof of Theorem 7.22, where we showed that the correct ken can be produced recursively, level by level, where the level of a proclet means the maximum number of nodes in a chain of its predecessors in the information-flow digraph of the Bounded Sequentiality Postulate. Unfortunately, nothing in our postulates requires that the level of a proclet can be described in the given vocabulary; in anthropomorphic terms, a proclet need not know what its level is. So the recursion suggested in the proof of Theorem 7.22, computing one level at a time, must be modified.

Here is a rough description of the modification; it will need to be corrected later, but it conveys the key idea. The computation proceeds in B phases, where

B is the bound on lengths of walks given by the Bounded Sequentiality Postulate. In the first phase, instead of having only the level-1 proclets execute Π, let all proclets execute Π, with empty mailboxes and with all displays equal to `undef`. For level-1 proclets, the empty mailbox is correct (i.e., it agrees with the correct ken), since a level-1 proclet cannot receive messages. Furthermore, level-1 proclets never pull information, so the displays don't matter for their computations. So level-1 proclets compute in the first phase exactly as they would compute with the correct ken; see Lemma 7.25. All higher-level proclets, however, are probably computing nonsense, since they have not yet received their messages and they may pull from incorrect displays. Since level-1 proclets execute their computations correctly, in particular, they send the correct messages and produce their correct displays. So at this point, all level-2 proclets receive their correct messages.

In the second phase, let every proclet execute Π with `myMail` containing the messages it received in phase 1 and with all displays as computed in phase 1. We claim that now proclets at levels 1 and 2 are doing the correct computations.

The justification of this claim depends on the fact that, in formulating the Bounded Sequentiality Postulate, we included in the digraph all edges along which information *might be* pushed or pulled. Thus, the first phase did not produce any messages to level-1 proclets, despite the fact that many higher-level proclets were computing with incorrect local states. Similarly, level-2 proclets received messages, in the first phase, only from level-1 proclets, and those messages were correct since the level-1 proclets were already using their correct local states. Furthermore, all level-1 proclets produced their correct displays, and these are the only displays from which level-2 proclets pull information. In view of Lemma 7.25, this establishes the claim.

In the third phase, let every proclet execute Π with `myMail` containing the messages it received in phase 2 and with all displays as computed in phase 2. Then an analogous proof establishes the claim that now all proclets at the first three levels are doing the correct computations. Continuing in this manner, we find that in phase B all proclets are doing the correct computations.

This completes the rough description of what our ASM does, and it shows that the correct ken is produced at the end. In fact, it shows that during phase B all proclets are already working with correct mailboxes and displays.

The rough description needs a minor correction to make sure the updates are correct. In phases 1 through $B - 1$, some proclets are still using incorrect kens, so they may produce incorrect updates of the global state. In fact, we want no global updates to be produced in these phases, because subsequent phases should work with the current global state, not an updated one. The solution to the problem is easy: just suppress production of updates until phase B.

This gives an ASM simulation of the given algorithm A, but it does not seem to be a step-for-step simulation. Each of the B phases seems to be a separate step of the simulating ASM, since later phases use the results of earlier ones. It is, however, well known (see Gurevich [2000, Theorem 7.1]) that ASMs can be speeded up, compressing several steps into one, without changing the

vocabulary. This idea will complete the proof, but we must check that it still works in the presence of comprehension terms. This completes the description of the general strategy; we now proceed to the details.

We begin by converting the "sequential ASM with Push" Π into two terms of the global vocabulary (or more precisely term schemas) $\mathtt{Outmail}(p, M, D)$ and $\mathtt{Dspl}(p, M, D)$ with the following intended meaning. Suppose X is a global state, p is a proclet in X, M is a multiset, and D is a function from proclets to elements of X. Consider executing Π once in the local state obtained from X by interpreting me as p, interpreting myMail as M, and interpreting Display as D (and, as always, using undef as the initial value of the dynamic symbol myDisplay). Then the value in X of $\mathtt{Outmail}(p, M, D)$ is to be the multiset of the resulting mailings: the multiset of all pairs $\langle q, m\rangle$ such that the execution of Π results in message m being sent to proclet q. The multiplicity of $\langle q, m\rangle$ in $\mathtt{Outmail}(p, M, D)$ is the multiplicity with which p sends m to q. The value in X of $\mathtt{Dspl}(p, M, D)$ is the multiset of all elements x such that the execution of Π results in an update giving myDisplay the value x. If several updates give myDisplay the same value x, then this is reflected in the multiplicity of x in $\mathtt{Dspl}(p, M, D)$.

We remark that, although p and M are elements of the state X, D serves as a place-holder for a function, which is of a higher type. Ultimately, D will be replaced by definitions of actual functions, so everything will make sense within X with no need for higher types. For now, however, it is convenient to use D, as several different functions will have to be substituted for it (namely the display functions at different phases).

The terms Outmail and Dspl will be constructed by induction on the program Π. This means that we actually define $\mathtt{Outmail}_R$ and $\mathtt{Dspl}_R$ for any "sequential ASM with Push" rule R, the desired Outmail and Dspl being $\mathtt{Outmail}_\Pi$ and (essentially) $\mathtt{Dspl}_\Pi$. The definitions of $\mathtt{Outmail}_R$ and $\mathtt{Dspl}_R$ are given by the following obvious inductions on R. As a notational convenience, if t is a term in the vocabulary of local states, then we write t' for the term obtained from t by replacing

—me by p,
—myMail by M,
—$\mathtt{Display}(u)$ by $D(u)$, and
—myDisplay by undef.

This notation also applies to formulas φ since these are a special case of terms. With these conventions, the definitions of $\mathtt{Outmail}_R$ and $\mathtt{Dspl}_R$ are as follows, by induction on the construction of the rules defining sequential ASMs with Push.

—If R is an update rule, then $\mathtt{Outmail}_R(p, M, D)$ is $\varnothing$ and $\mathtt{Dspl}_R(p, M, D)$ is also $\varnothing$ unless R has the form myDisplay $:= t$ in which case $\mathtt{Dspl}_R(p, M, D)$ is $\{\!\{t'\}\!\}$.
—If R is Push t_0 to t_1, then $\mathtt{Outmail}_R(p, M, D)$ is $\{\!\{\langle t_1', t_0'\rangle\}\!\}$ and $\mathtt{Dspl}_R(p, M, D)$ is $\varnothing$.

—If R is do in parallel $R_0, \ldots, R_k$ enddo, then $\mathtt{Outmail}_R(p, M, D)$ is the sum $\mathtt{Outmail}_{R_0}(p, M, D) \uplus \cdots \uplus \mathtt{Outmail}_{R_k}(p, M, D)$ and similarly $\mathtt{Dspl}_R(p, M, D)$ is the sum $\mathtt{Dspl}_{R_0}(p, M, D) \uplus \cdots \uplus \mathtt{Dspl}_{R_k}(p, M, D)$.

—If R is if φ then R_0 else R_1 endif, then $\mathtt{Outmail}_R(p, M, D)$ is

$$\{\!\{z : z \in \mathtt{Outmail}_{R_0}(p, M, D) : \varphi'\}\!\} \uplus \{\!\{z : z \in \mathtt{Outmail}_{R_1}(p, M, D) : \neg\varphi'\}\!\},$$

and similarly $\mathtt{Dspl}_R(p, M, D)$ is

$$\{\!\{z : z \in \mathtt{Dspl}_{R_0}(p, M, D) : \varphi'\}\!\} \uplus \{\!\{z : z \in \mathtt{Dspl}_{R_1}(p, M, D) : \neg\varphi'\}\!\}.$$

We define $\mathtt{Outmail}(p, M, D)$ to be $\mathtt{Outmail}_\Pi(p, M, D)$, but we define $\mathtt{Dspl}(p, M, D)$ to be TheUnique(AsSet($\mathtt{Dspl}_\Pi(p, M, D)$)). Notice that $\mathtt{Dspl}_\Pi(p, M, D)$ represents the multiset of values assigned to myDisplay by the execution of Π (in the local state described above). So if only one value is assigned to myDisplay (perhaps several times) then this is the value of $\mathtt{Dspl}(p, M, D)$; if either no value or two different values are assigned to myDisplay then the value of $\mathtt{Dspl}(p, M, D)$ is undef.

We next define terms $\mathtt{Mailbox}_k(p)$ and $\mathtt{Display}_k(p)$ in the vocabulary of the global state. These terms are intended to denote the values of $\mathtt{Mailbox}(p)$ and $\mathtt{Display}(p)$ after k phases of the simulation described in the general strategy above. The definition is by induction on k.

—$\mathtt{Mailbox}_0(p)$ is $\varnothing$.

—$\mathtt{Display}_0(p)$ is undef.

—$\mathtt{Mailbox}_{k+1}(p)$ is

$$\biguplus \{\!\{\{\!\{\mathtt{second}(z) : z \in \mathtt{Outmail}(q, \mathtt{Mailbox}_k(q), \mathtt{Display}_k) : \mathtt{first}(z) = p\}\!\} : q \in \mathtt{Proclet} : \mathtt{true}\}\!\}. \quad (1)$$

—$\mathtt{Display}_{k+1}(p)$ is $\mathtt{Dspl}(p, \mathtt{Mailbox}_k(p), \mathtt{Display}_k)$.

To see what is happening in this definition, suppose, as an induction hypothesis, that for every proclet p the values of $\mathtt{Mailbox}_k(p)$ and $\mathtt{Display}_k(p)$ are the mailbox and display of p after k phases of the simulation described above. Then in phase $k + 1$, each proclet p will execute Π in the local state where myMail denotes $\mathtt{Mailbox}_k(p)$ and Display denotes $\mathtt{Display}_k$ (and of course me denotes p and myDisplay is initially undef). It will thus send mailings as described by $\mathtt{Outmail}(p, \mathtt{Mailbox}_k(p), \mathtt{Display}_k)$ and change its display to $\mathtt{Dspl}(p, \mathtt{Mailbox}_k(p), \mathtt{Display}_k)$.

Since this is the case for every proclet, we find that the messages pushed to p by another proclet q are exactly the members (with multiplicity) of the multiset

$$\{\!\{\mathtt{second}(z) : z \in \mathtt{Outmail}(q, \mathtt{Mailbox}_k(q), \mathtt{Display}_k) : \mathtt{first}(z) = p\}\!\}.$$

By taking the union of these multisets over all q, we obtain the multiset of all messages sent to p during this phase, the multiplicity of any message m being the sum, over all proclets that push m to x, of the multiplicities with which they push it. Thus, we obtain exactly the mailbox of p after $k + 1$ phases, and this is how $\mathtt{Mailbox}_{k+1}(p)$ was defined.

Similarly, $\mathtt{Dspl}(p, \mathtt{Mailbox}_k(p), \mathtt{Display}_k)$ is the value given to myDisplay by proclet p during phase $k + 1$ (or undef if there isn't a unique such value). But

this is exactly the display of p after $k+1$ phases, and it is how $\texttt{Display}_{k+1}(p)$ is defined.

After $B-1$ phases, every proclet has its correct mailbox and display. In other words, $\texttt{Mailbox}_{B-1}$ and $\texttt{Display}_{B-1}$ define the correct ken. So in phase B, updates of the global state can be computed. (Messages and displays can also be computed, but they will just reproduce what was computed at the preceding step.) So the update set $\Delta(A, X)$ of the overall algorithm, which consists of all the updates of the global state produced by all the proclets operating in local states given by the correct ken, is the same as the update set produced by the ASM program

$$\texttt{do forall } p \in \texttt{Proclet } \Pi^*(p) \texttt{ enddo}.$$

Here $\Pi^*(p)$ is obtained from Π by replacing

—me with p,
—myMail with $\texttt{Mailbox}_{B-1}(p)$,
—Display with $\texttt{Display}_{B-1}$,
—every Push rule with Skip, and
—every update of myDisplay with Skip.

Thus, we have an ASM producing the same updates of the global state as the given algorithm. □

Remark 10.2. As required by our strong definition of equivalence, the states of the given algorithm and the simulating ASM in this theorem are identical. There is no coding involved (unless one considers the Abstract State Postulate to involve coding).

Remark 10.3. We emphasize that the style of computation used in the preceding proof, with all proclets executing their algorithm in every phase, often with incorrect mailboxes, is intended to serve only for proving the theorem, not as an actual implementation of anything. We nested terms $\texttt{Mailbox}_k$ to a depth of $B-1$ levels in order to treat the general case, an arbitrary algorithm satisfying our postulates. In practice, other approaches will usually be available and much more efficient. Here are some simple examples.

Suppose the proclets "know" their level in the information-flow digraph. For example, suppose there are predicates $\texttt{level}_k$ for $1 \le k \le B$, such that $\texttt{level}_k(p)$ is true if and only if p is a proclet of level k in the information-flow digraph. Then by imposing a global synchronization on the phases, we can have proclets at any level k compute only in phase k. Formula (1) for $\texttt{Mailbox}_{k+1}(p)$ would then be modified to read

$$\texttt{Mailbox}_k(p) \uplus \biguplus \{\!\{\{\!\{\texttt{second}(z) : z \in \texttt{Outmail}(q, \texttt{Mailbox}_k(q), \texttt{Display}_k) : \texttt{first}(z) = p\}\!\} : q \in \texttt{Proclet} : \texttt{level}_{k+1}(q)\}\!\}. \quad (2)$$

In this version of the algorithm, for each proclet q, there is only one place where anything of the form $\texttt{Outmail}(q, \dots)$ needs to be computed, namely in producing

$\mathtt{Mailbox}_{k+1}$ from $\mathtt{Mailbox}_k$ where $k+1$ is the level of q. In this sense, this version of the algorithm is more efficient than the one in the proof of Theorem 10.1.

A useful special case of this is when there are only two levels, and proclets know which level they are in. For example, the top level may consist of proclets controlling counters and the bottom level of proclets requesting increments of those counters.

Another possibility that occurs frequently in practice is that each proclet p knows which other proclets will send it messages and how many messages to expect from each one. Suppose, for example, that the state contains a term $\mathtt{Senders}(p)$ whose value for any proclet p is the multiset of proclets that will send messages to p, the multiplicity of q in $\mathtt{Senders}(p)$ being the number of messages that q sends to p. Then we can modify the algorithm so that each p acts only when it has received all the messages it expects. In this situation, in addition to modifying formula (1), we introduce additional terms $\mathtt{Received}_k(p)$ to keep track of the senders of the messages received by p at or before phase k. We let both $\mathtt{Mailbox}_0(p)$ and $\mathtt{Received}_0(p)$ be $\varnothing$. We define $\mathtt{Mailbox}_{k+1}(x)$ for $0 \le k < B-1$ to be

$$\mathtt{Mailbox}_k(p) \uplus \biguplus \{\!\{\{\!\{\mathtt{second}(z) : z \in \mathtt{Outmail}(q, \mathtt{Mailbox}_k(q), \mathtt{Display}_k) : \mathtt{first}(z) = p\}\!\} : q \in \mathtt{Proclet} : \mathtt{Received}_k(q) = \mathtt{Senders}(q) \neq \mathtt{Received}_{k-1}(q)\}\!\}, \quad (3)$$

where the clause about $\mathtt{Received}_{k-1}(q)$ is to be omitted when $k = 0$. Similarly, we define $\mathtt{Received}_{k+1}(p)$ to be

$$\mathtt{Received}_k(p) \uplus \biguplus \{\!\{\{\!\{q : z \in \mathtt{Outmail}(q, \mathtt{Mailbox}_k(q), \mathtt{Display}_k) : \mathtt{first}(z) = p\}\!\} : q \in \mathtt{Proclet} : \mathtt{Received}_k(q) = \mathtt{Senders}(q) \neq \mathtt{Received}_{k-1}(q)\}\!\}. \quad (4)$$

A similar construction is available if each p knows how many messages to expect altogether, regardless of their senders.

Remark 10.4. As noted earlier, algorithms that use and combine partial updates in the sense of Foundations of Software Engineering Group, Gurevich et al. [2001] and Gurevich and Tillman [2001] are covered by our postulates. The partial updates are viewed as messages to proclets that combine them into appropriate (ordinary) updates. Theorem 10.1 shows that these can be simulated by ASMs, which don't use partial updates. Thus, the theorem provides an elimination of partial updates.

11. SEVERAL STEPS FOR ONE

This section presents an alternative ASM simulation of parallel algorithms, which has some advantages and some disadvantages vis-à-vis the simulation in the proof of Theorem 10.1. The first advantage is that it avoids the deeply nested comprehension terms $\mathtt{Mailbox}_k$. The price for this is that the vocabulary is slightly increased and that the simulation is no longer step-for-step but

rather $2B$ steps for one, where B is the bound from the Bounded Sequentiality Postulate. The second advantage is that the same simulation can be applied to algorithms that do not satisfy Bounded Sequentiality but only Acyclicity. In this case, the simulation uses an unbounded number of steps for one step of the given algorithm. Finally, we can restore step-for-step simulation by introducing the notion of submachines. In the general situation, this use of submachines looks like just a brutal redefinition of the notion of step, but in many real situations it behaves quite reasonably.

The material in this section will not be used later in the paper and can therefore be skipped without loss of continuity.

11.1 Phases Become Steps

The ASM produced in the proof of Theorem 10.1 consisted of a single `do forall` rule, but a great deal more parallelism and a certain amount of sequentiality were hidden in its comprehension terms. Notice that the definition of $\texttt{Mailbox}_k$ involves $O(k)$ nesting of comprehension terms. Any comprehension term $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ involves parallelism in that computations are done for all $x \in r$. Nesting of comprehension terms involves sequentiality in that the inner terms must be evaluated as part of the evaluation of the outer ones. So the ASM in the proof really involves $O(B)$ sequential steps, each of which involves parallelism.

Of course, this is not surprising, since the intuitive algorithm that this ASM expresses consisted of B phases, in each of which all the proclets operate.

The algorithm can be rewritten as another, perhaps more natural ASM in which this sequentiality is made more explicit by using separate steps for the phases, rather than deeply nested comprehension terms. We present such a reformulation in the present subsection; in the next subsection we shall show that it allows us to analyze algorithms that do not satisfy the Bounded Sequentiality Postulate but only the weaker Acyclicity Postulate.

This version of the ASM would not be appropriate for proving the theorem, since it simulates the given algorithm not step for step but $2B$ steps for one. It also uses additional vocabulary `phase` (a number, with initial value 1), `mode` (a Boolean value, initially `true`), `sends` (a ternary number-valued function, initially constant with value 0), `Mailbox` (a unary multiset-valued function), `newdisplay` (a unary function), and `Display` (another unary function).

Furthermore, the most natural formulation of this algorithm uses some additional background. Specifically, it uses natural numbers and the construction

$$\{\!\{z : z \in r \texttt{ with multiplicity } \mu(z)\}\!\}.$$

where r denotes a set and $\mu(z)$ a natural number for each z; this new term denotes the multiset consisting of the elements z of r, each repeated with the corresponding multiplicity $\mu(z)$. For those interested in keeping the background minimal, we shall explain at the end of this section how this additional background can be reduced to the standard multiset background.

The purpose of adding `phase`, `mode`, `sends`, `Mailbox`, `newdisplay`, and `Display` to the vocabulary is as follows. We use `phase` to count phases; its initial value is 1 and it increases by 1 after each phase, until it reaches B and the global updates

are executed. The two values of mode correspond to the two parts of each phase, first computing the updates of myDisplay and the mailings and then updating the displays and putting the messages into the appropriate mailboxes. We let mode = true correspond to the computing part and mode = false to the updating and mailing part. In fact, for improved readability, we shall use computing and communicating as aliases for true and false respectively, in this context. The function sends serves to store the computed messages between the two parts of a phase; the value of sends(p, m, q) is the multiplicity with which proclet p is sending message m to proclet q. Similarly, newdisplay stores the displays that are produced in the computing part of a phase and will take effect in the subsequent communicating part. The functions Mailbox and Display have the obvious meanings; they are updated during the communicating part of a phase for use in the computing part of the next phase.

We need a little more notation to describe the ASM we have in mind. As in the proof of Theorem 10.1, let Π be the "ASM with Push" that describes the operation of each individual proclet. We describe two variants $\Pi_0(p)$ and $\Pi_1(p)$ of Π. These will be ordinary ASMs (without outputs, i.e., without Push). $\Pi_0(p)$ describes the action of proclet p executing Π; it uses the ternary function sends to store, by means of updates, information about the messages sent by Π, and it similarly uses newdisplay to store updates of the display of p. $\Pi_1(p)$ is similar except that it performs none of the global updates in Π; it only updates newdisplay and sends. To simplify the description of $\Pi_0(p)$ and $\Pi_1(p)$, we assume that Π has the form of an if-chain,

```
if φ1 then R1
elseif φ2 then R2
...
elseif φk then Rk
endif,
```

where each R_i is a do in parallel block of updates and send rules and each φ_i is a complete description of equalities and inequalities between the terms in T, which include all the terms occurring in the R_i. This normalization of Π involves no loss of generality, since the proof of Theorem 6.13 in Gurevich [2000] and its adaptation to produce Π in our proof of Theorem 10.1 above actually produce Π in the required form. We can further assume that no R_i contains syntactically distinct terms that denote, according to the corresponding φ_i, the same value. This is easy to arrange because φ_i specifies exactly which terms from T denote the same value, so we can replace each of these terms in R_i by a selected representative of its equivalence class (with respect to "equality according to φ_i"). (The point of this last normalization is that multiplicities of messages can be read off from the Push rules.) Finally, we assume that each of the rules R_i contains at most one update of myDisplay. This is automatically true for the ASM programs Π produced in our proof of Theorem 10.1.

Now obtain $\Pi_0(p)$ from Π by

—replacing the nullary function symbol me with the variable p,
—replacing myMail with Mailbox(p),

```
if phase < B and mode=computing then
   do in parallel
      do forall p ∈ Proclet, Π1(p)     //do the work
      do forall p ∈ Proclet, Display(p):=undef     //clean up
      mode:=communicating
elseif phase < B and mode=communicating then
   do in parallel
      do forall q ∈ Proclet
         Mailbox(q):=
               ⊎{{{m : m ∈ L(p) with multiplicity sends(p, m, q)}} :
               p ∈ Proclet : true}} //do the work
      do forall q ∈ Proclet
         Display(q):=newdisplay(q) //more work
      do forall p ∈ Proclet do forall m ∈ L(p) do forall q ∈ Proclet
         sends(p, m, q):=0 //clean up
      do forall p ∈ Proclet newdisplay(p):=undef     //clean more
      mode:=computing, phase:=phase+1
elseif phase = B and mode=computing then
   do in parallel
      do forall p ∈ Proclet, Π0(p)
      do forall p ∈ Proclet, Display(p):=undef
      mode:=communicating
elseif phase = B and mode=communicating then
   do forall p ∈ Proclet do forall m ∈ L(p) do forall q ∈ Proclet
      sends(p, m, q):=0
   do forall p ∈ Proclet newdisplay(p):=undef
   mode:=computing, phase:=1
```

Fig. 1.

—if a Push-rule Push t_0 to t_1 endpush occurs exactly n times in a block R_i, replacing these n occurrences with (one occurrence of) the update rule sends$(p, t_0, t_1) := n$, and

—replacing any update of the form myDisplay $:= t$ with newdisplay$(p) := t$.

And $\Pi_1(x)$ is obtained by these same replacements and in addition

—replacing every global update rule in Π with skip.

(Note that this last replacement applies only to the update rules in Π that update global dynamic functions, not to those introduced in Π_0 as replacements for Push rules and for updates of myDisplay.)

We write $\mathcal{L}(p)$ for the set term AsSet($\{\!\{u_1, \ldots, u_N\}\!\}$) where the u_i's are all the terms occurring in $\Pi_1(p)$. Recall that AsSet arranges that each member of $\mathcal{L}(p)$ has multiplicity 1 regardless how often it occurs among the u_i's. Notice that the u_i's include names for all the messages that p could send and for the recipients of these messages.

With these notations, the desired ASM is as in Figure 1; it should be clear (especially in view of the comments, marked //) that it describes the same computational process as in the proof of Theorem 10.1. To save some space, we have omitted enddo and endif, relying on indentation to indicate where rules end. (Each line l in the figure should be regarded as implicitly ending all current

do-rules that begin with equal or greater indentation than l. And of course the end of the program implicitly closes all current rules.)

11.2 Unbounded Sequentiality and Submachines

As mentioned earlier, this ASM has another use, beyond "unraveling" the comprehension terms and making the phases explicit. Consider replacing the Bounded Sequentiality Postulate with the Acyclicity Postulate, requiring only that the information-flow digraph is acyclic and finite, but not necessarily of bounded depth. That is, the maximum length of a walk in the digraph, though finite in each state, can grow with the state. Such an algorithm would not be a parallel algorithm in the sense of this article, but it seems sufficiently similar to parallel algorithms to be amenable to a similar analysis. Notice, in particular, that Theorem 7.22 still applies; it used only the acyclicity of the digraph. So the modified notion of algorithm still gives a well-defined correct ken and thus a well-defined set of updates.

The analysis given by the ASM construction in the proof of Theorem 10.1 does not carry over to this modified notion of algorithm. It depended crucially on the bound B to produce the terms $\texttt{Mailbox}_{B-1}(p)$. Removing the uniform bound B leaves us with no terms to describe the final mailboxes.

But the reformulated version of the algorithm can be adapted to the new situation by using the notion of *submachines* of ASMs. A submachine of an ASM is another ASM, operating on the state of the first ASM plus possibly some "scratch paper." One can think of submachines as agents, in the sense of Gurevich [1995, Section 6]; they occur naturally (but without the name "submachine") in the context of recursive ASMs Gurevich and Spielmann [1997]. The action of the submachine, though it may be many steps in its own right, counts as only a single step, like an intervention of the environment, for the larger machine. For details about submachines, see Börger and Schmid [2000], Gurevich et al. [2001], and Gurevich and Tillman [2001].

We can trivially modify the ASM in Figure 1 by delegating all its work to a submachine. Instead of using the counter `phase` to tell when the iteration should stop, let the submachine keep track of both the current and immediately preceding values of `Mailbox` and `Display`, and let it stop when neither of them changes from one submachine step to the next. The effect is that the unbounded number of steps of the submachine now counts as a single step of the ASM. Furthermore, this ASM uses the same background and vocabulary as the given algorithm, since `phase`, `mode`, `sends`, `newdisplay`, `Mailbox`, `Display`, and the storage of the previous phase's mailbox and display can all be counted as scratch paper of the submachine.

Thus, all algorithms that are like parallel algorithms, except for having only acyclic sequentiality instead of bounded sequentiality, are equivalent to ASMs with submachines.

This simulation and the associated step-counting seem very distorted, since all of the work is done by the submachine and counted as a single step. But in practice, the situation is rarely so bad. Typically, a submachine works much more locally. Even if a step involves several submachines, their presence does

not greatly distort the intuitive notion of step. A typical example would be an algorithm whose steps satisfy bounded sequentiality except that the elements of some list (of unbounded length) have some operation performed on them one at a time. The AsmL distribution [Foundations of Software Engineering Group] contains some other examples.

We should also mention the converse result: ASMs with submachines are algorithms that satisfy our postulates except that Bounded Sequentiality is weakened to Acyclicity.

The concept of submachine plays an important role in very interesting questions related to partial updates Gurevich and Tillman [2001] and to the design of the specification language AsmL (see Foundations of Software Engineering Group, and Gurevich et al. [2001]).

Reduction 11.1. As promised above, we indicate here how to define, in terms of multiset operations, the additional background constructions used in the ASM exhibited above. First, the natural number n can be coded as the multiset whose only member is `true` with multiplicity n. Thus 0 is identified with $\varnothing$ and $n+1$ with $n \uplus \{\!\{\texttt{true}\}\!\}$. Then the term $\{\!\{z : z \in r \texttt{ with multiplicity } \mu(z)\}\!\}$ can be defined, thanks to our coding of numbers, as

$$\biguplus \{\!\{\{\!\{z : v \in \mu(z) : \texttt{true}\}\!\} : z \in r : \texttt{true}\}\!\}.$$

12. ACTUAL OR POTENTIAL MAILBOXES

In this section, we further explore an objection, already briefly considered in Remark 7.20, to the Bounded Sequentiality Postulate. The objection concerns the use, in the definition of the information-flow digraph, of all possible kens. Why not use only the correct kens?

As indicated above, the correct ken is not well-defined without some acyclicity information, which proceeds from the Bounded Sequentiality Postulate. To put it another way, the correct ken and the computations of the proclets are defined in an apparently circular fashion (see the proof of Theorem 7.22 and Remark 7.24), which can be viewed as a fixed-point requirement. Any ken gives rise, via the proclets' computations, to certain messages and displays, which in turn form a ken, and correctness requires that the latter ken coincide with the former. We used the Bounded Sequentiality Postulate to ensure that this fixed-point requirement has a unique solution, the correct ken.

If one wants to avoid considering all possible kens and instead focus on the correct one (and require bounded sequentiality only for the digraph it defines), then the following seems to be a reasonable proposal. Require first that the fixed-point requirement has a unique solution—call this the Unique Kens Postulate—and then require Bounded Sequentiality only for this ken. The Unique Kens Postulate amounts to assuming, rather than proving, Theorem 7.22. Thus, it removes one of the justifications, in Remark 7.20, for the use of all possible kens. If we restrict our attention to algorithms whose proclets communicate only by pushing, not by pulling information, then the other justification in Remark 7.20 also disappears. So in this "no pulling" context, the combination of the Unique Kens Postulate and Bounded Sequentiality for the

information-flow digraph of the correct ken is a plausible proposal for avoiding the consideration of all kens together.

We shall show that this proposal does not work, by exhibiting some things that would be parallel algorithms under the proposed postulates but that do not intuitively work as parallel algorithms. In the rest of this section, we use only non-pulling algorithms, so a ken for a state X amounts to (X plus) a mailbox function.

Here is a simple example to show what can go wrong with the proposal above. Consider an algorithm with three proclets, called a, b, c (all named by nullary symbols so they can refer to each other). The algorithms they execute are as follows. If a's mailbox is empty it sends message 0 to b; otherwise it does nothing. If b's mailbox is empty, it sends messages 0 to a and 1 to c; otherwise it does nothing. If c's mailbox contains message 1 and nothing else, it sends message 0 to itself; otherwise it does nothing.

Temporarily ignoring proclet c, we find two fixed-points (in the sense of the requirements for a correct ken, see Remark 7.24) for the interaction of a and b, namely, one of their mailboxes contains message 0 and the other is empty. So these proclets alone would violate the Unique Kens Postulate.

On the other hand, consider proclet c by itself. If c has 1 in its mailbox, then there is no fixed-point for its algorithm, because of the circularity. c must send message 0 to itself, and must therefore have 0 in its mailbox, if and only if it does not have 0 in its mailbox. So again, the Unique Kens Postulate would be violated.

Finally, let us consider the complete system of three proclets. The coupling between the a, b pair and c, via b's possible message 1 to c, makes the Unique Kens Postulate true for the complete system. Indeed, one of the two possible fixed-points for the a, b subsystem, the one where a has a message in its mailbox and b doesn't, cannot be consistently extended to c, for in this situation c will have message 1 from b, and then any possible mailbox for c leads, as we saw, to a contradiction. So the a, b configuration must be that a's mailbox is empty while b's contains message 0. In this situation, b does not send any messages, so c does not have 1 in its mailbox, and therefore c has a unique consistent mailbox, namely $\varnothing$.

Summarizing, we have that this system satisfies the Unique Kens Postulate, with $\mathtt{Mailbox}(b) = \{\!\{0\}\!\}$ and $\mathtt{Mailbox}(a) = \mathtt{Mailbox}(c) = \varnothing$. Furthermore, the information-flow digraph associated to this ken has just a single edge, from a to b, so it satisfies Bounded Sequentiality (with $B = 2$).

Despite satisfying the proposed weakening of our Bounded Sequentiality Postulate, this system does not seem to be a reasonable parallel algorithm. The uniqueness of the fixed-point comes about not through any way of computing the fixed-point but by the after-the-fact observation that one of the two possible configurations of a, b leads to impossible behavior of c.

More specifically, the sort of ASM simulation used in the proof of Theorem 10.1 will not work with this system. Indeed, this simulation leads to an oscillation of period 2. After any odd number of phases, the mailboxes of a and b contain 0 and that of c contains 1; after any non-zero even number of phases, the mailboxes of a and b are empty and that of c contains 0.

The simulation never produces the (unique) fixed-point and (therefore) never stabilizes.

More generally, there seems to be no way to program this sort of situation, in the sense of finding the unique fixed-point, without just searching through all possible `Mailbox` functions to see which one works.

In the remainder of this section, we show that finding the unique `Mailbox` for a given algorithm (subject to the Unique Kens Postulate and Bounded Sequentiality for the actual kens) may be unacceptably difficult in a complexity-theoretic sense. For this purpose, we first describe the relevant complexity class.

Definition 12.1. A *find unique* problem is a problem of the form

—Instance: a binary string x such that there is exactly one y satisfying $R(x, y)$,
—Task: Find the y such that $R(x, y)$,

where R is a polynomial-time computable binary relation, holding only between certain binary strings of equal length.

Remark 12.2.[5] As an indication of the possible complexity of find unique problems, we mention that this class of problems includes integer factorization. An instance of this problem is an integer (in binary notation) and the problem is to list its prime factors (with multiplicities) in non-decreasing order. Although the factorization problem is not known to be hard, that is, to be unsolvable in polynomial time, it is widely believed to be hard—widely enough to serve as a basis for cryptographic schemes.

After suitable padding to make the inputs and outputs have the same length, the factorization problem fits the definition of a find unique problem. The reason is that the prime-testing algorithm of Agrawal et al. [2002] allows us to check any proposed factorization in polynomial time.

Notice that factorization has the special property, not required for find unique problems in general, that for *every* x there is a unique y satisfying $R(x, y)$. In other words, every x is a legitimate instance.

Remark 12.3. Find unique problems with the special property just mentioned, that every x is a legitimate instance, are closely related to the complexity class NP ∩ co-NP of decision problems. Specifically, each of these special find unique problems amounts to a combination of polynomially many decision problems, each of which is in NP ∩ co-NP. Using notation as in the definition of find unique problems, we can, on an n-bit input x, obtain the appropriate y from the answers to the n decision problems "Is the ith bit of the correct y equal to 1?" Each of these decision problems is in NP because it can be phrased as "Is there a binary string y of length n, with ith bit 1, and satisfying $R(x, y)$?" It is also in co-NP because the negated question can be phrased the same way with "ith bit 0."

The NP problems arising in this way have the "unique witness" property, namely they can be expressed in the form $\exists w\, P(z, w)$ where for each z there

[5]This remark was simplified on 1 August 2003 in light of the result of Agrawal et al. [2002].

is at most one witness w (of polynomially bounded length). Conversely, if a set and its complement are both NP with unique witnesses, then the decision problem for this set immediately reduces to a find unique problem for which all strings are legitimate inputs. Namely, if the set's NP form is $\exists w\, P(z, w)$ and its complement's NP form is $\exists w\, Q(z, w)$, both having unique witnesses, then the decision problem for this set reduces to the find unique problem "given x find the unique w such that $P(x, w) \vee Q(x, w)$."

If we look at general find unique problems, for which not every string is a legitimate input, then there is a similar connection with a modified version of NP∩co-NP. A problem in this modified complexity class is given by two NP sets, say A and B; a legitimate instance is an x belonging to exactly one of the two sets; and the question about instance x is which of the two sets does it belong to. (When every x is legitimate, A and B are complementary, and we have an NP ∩ co-NP problem.)

Remark 12.4. As another indication of the possible complexity of find unique problems, we mention that a fairly standard oracle argument produces an oracle relative to which not all find unique problems are solvable in polynomial time.

The relevance of the class of find unique problems to our discussion of unique mailboxes (i.e., unique kens, since we deal with algorithms that don't pull information) is that the latter provides a complete problem for the former.

Consider the following problem, which we call the mailbox problem.

—Instance: a finite first-order structure and an "ASM with `Push`" in the vocabulary of this structure augmented by `me` and `myMail`, such that there is a unique correct `Mailbox` function and such that the information-flow digraph for this `Mailbox` function has no walks of length greater than 3.

—Task: Find the unique correct `Mailbox` function.

After suitable coding by binary strings and suitable padding to make the lengths of the strings match, this problem becomes a find unique problem. This follows immediately from the observation that one can compute in polynomial time whether a given `Mailbox` function is correct for the given structure and algorithm.

THEOREM 12.5. *The mailbox problem is complete for the class of find unique problems.*

PROOF. Let an arbitrary find unique problem be given, and let R be the binary relation defining it. So the problem is, given x such that there is a unique y with $R(x, y)$, to find this y. Fix a Turing machine M that computes $R(x, y)$ in time $p(n)$ where n is the common length of x and y and p is a polynomial. (Remember that R holds only between binary strings of equal length.) We show how to produce, for any given binary string x, a structure and an algorithm, constituting an instance of the mailbox problem, such that, from their uniquely determined correct `Mailbox` function, we can easily compute the y such that $R(x, y)$. Assume for simplicity and without loss of generality that the alphabet

of the Turing machine M consists of only 0 and 1; 0 is used as the blank symbol. Let s be the number of control states of M.

Our structure will be built from small components, many of which are essentially like the components of the small 3-proclet example given above. Specifically, there will be components of the following sorts.

—binary switch: This consists of two proclets, each of which sends message 0 to the other if and only if its own mailbox is empty. So these are like the a and b in the earlier example. Each binary switch has two consistent (with the algorithm) configurations, namely one mailbox contains 0 and the other is empty.

—s-ary switch: This generalization of the binary switch consists of s proclets, each of which sends message 0 to all the others if its own mailbox is empty; otherwise, it sends nothing. It is trivial to check that an s-ary switch has exactly s consistent configurations, in each of which one proclet's mailbox is empty and all the other mailboxes contain 0. (Recall that s is the number of control states of M.)

—black hole: This consists of a single proclet, which sends message 0 to itself if its mailbox contains something but does not contain 0. When activated, by receiving a message other than 0, a black hole has no consistent configuration. In the earlier example, c was a black hole.

These components will be arranged in (essentially) a $p(n)$ by $p(n)$ array, intended to describe the computation of M on an input consisting of the given x together with some unspecified y.

We now give a detailed description of the structure associated to a given binary string x of length n. Interspersed with the description is an explanation, in square brackets, of the intended purpose of the various parts of the structure.

First, the structure contains a linearly ordered set I of $p(n)$ elements, the successor relation being named by a predicate symbol of the vocabulary. For convenience, we identify I with $\{1, 2, \ldots, p(n)\}$ (with the usual ordering).

Next, there is a $p(n)$ by $p(n)$ array, indexed by $I \times I$, each cell of which contains two binary switches. In each of these binary switches, the two proclets are labeled 0 and 1 (and of course also labeled by their row and column numbers and another label to distinguish one binary switch from the other). [In the cell at row i and column j, henceforth called (i, j), the configuration of the first switch indicates which symbol, 0 or 1, is written in cell number j of the Turing tape at step i of the computation; the symbol is the same as the label of the proclet whose mailbox is empty. The second switch indicates whether this cell, number j, is the scanned cell at time i.]

For each row, there is an s-ary switch, whose proclets are labeled by the control states of M (and of course by the row). [In the switch at row i, the label of the (unique) proclet whose mailbox is empty is the control state of the computation at step i.]

There is one black hole. [It will be activated whenever something "bad" is discovered in the alleged computation described by the switches.]

Each cell in the $p(n)$ by $p(n)$ array has an additional proclet, called its cell guard. [For an unscanned cell, the cell guard's job is to make sure the content doesn't change from one step to the next; if it does change, the guard activates the black hole. For a scanned cell, the cell guard reports the nearby action to the row guard (see below).]

Each row has an additional proclet, called the row guard. [The row guard makes sure that its row has exactly one scanned cell and that the action near it is as specified by the program of M. If either of these is not the case, then the row guard activates the black hole.]

The bracketed explanations above probably make it clear what algorithms are executed by the various proclets, but to be safe we add a few clarifying comments, before continuing to the last part of the specification. The two switches in each cell inform some nearby cell guards of their configurations. Specifically, in each switch the proclet with empty mailbox sends its label (including not only the 0 or 1 but also which cell and which of the two switches it inhabits) to the guards of that cell and of the three nearest cells in the previous row. That is, cell (i, j) reports to the guards of cells (i, j), $(i-1, j-1)$, $(i-1, j)$, and $(i-1, j+1)$ (omitting any of these if they don't exist because they're outside the array). The effect is that the cell guard at (i, j) knows, by looking into its mailbox, what the tape contents are at (i, j), $(i+1, j-1)$, $(i+1, j)$, and $(i+1, j+1)$ and which, if any, of these are scanned. If the guard finds that (i, j) is not scanned but $(i+1, j)$ has different content, then it activates the black hole by sending it a 1. If the guard finds that (i, j) is scanned, then it sends to the row guard for row i a complete description of what it found in cells (i, j), $(i+1, j-1)$, $(i+1, j)$, and $(i+1, j+1)$. Each s-ary switch, say at row i, sends a message to the corresponding row guard, telling what configuration it is in; all that is needed here is for the proclet in the switch with empty mailbox to send its label to the row guard. If the row guard's mailbox contains either more or fewer than one message from cell guards, it activates the black hole. If it finds exactly one such message, then it compares that message, the message it received from the s-ary switch in its row, and the Turing machine program of M; if there is a discrepancy then it activates the black hole.

The description so far ensures that the black hole will be activated, and so there will be no consistent configuration, if the switches fail to code a computation of M.

We also want to activate the black hole if the input is wrong or if the computation of M fails to accept. In detail, this means the following. If $1 \leq j \leq n$, so that cell $(1, j)$ of our array corresponds to the jth cell of the initial Turing tape and should therefore code the jth bit of x, if the code is wrong then then the black hole should be activated. That is, if the jth bit of x is 0 (resp. 1) and if the proclet labeled 1 (resp. 0) in the switch coding the tape content at $(1, j)$ has empty mailbox, then this proclet sends a message 1 to the black hole. Also, if $j > 2n$ and the proclet labeled 1 in the switch coding the tape content at $(1, j)$ has empty mailbox, then this proclet sends a message 1 to the black hole; this ensures that the computation encoded by the switches has 0's (serving as blanks) in all input bits past $2n$. Thus, the input must be x followed by some n-bit string y, followed by all 0's. Furthermore, if the first cell in the first row

is not scanned, the proclet indicating this sends 1 to the black hole; so the encoded computation must start with the leftmost cell scanned. Finally, if, in the s-ary switch in the last row, the proclet with empty mailbox is labeled by a non-accepting state of M, then this proclet sends 1 to the black hole.

It follows that the only consistent configurations of this structure and algorithm are those where the switches code a correct and accepting computation of M from an input of the form xy (followed by blanks), where x is the given string of length n and y is another binary string of length n. By our assumption on the input x, we know that there is exactly one such y and therefore exactly one such computation. That is, our structure and algorithm have exactly one consistent configuration and therefore constitute an instance of the mailbox problem. Furthermore, from the solution to this instance, the unique appropriate computation, one can read off the unique y such that $R(x, y)$, for this y is encoded by the mailboxes of the switches in positions $n + 1$ through $2n$ of the first row.

Thus, we have a reduction of the find unique problem represented by R to the mailbox problem. It remains to observe that the reduction is computable in polynomial time. That is, for any fixed Turing machine M and polynomial p, giving the relation R, we can in polynomial time convert an arbitrary binary string x to the structure and algorithms described above. □

In view of the theorem and the earlier remarks about the complexity of find unique problems, it appears that the combination of the Unique Kens Postulate and Bounded Sequentiality for the correct ken is not a reasonable way to describe parallel algorithms. The combination describes something more complicated—essentially a search through all possible kens.

To conclude this section, we comment on another proposal, which seems less radical than the combination considered above. Define a ken to be *correct* if it satisfies the conclusion of Theorem 7.22, without making (for the moment) any assumption about the existence or uniqueness of such a ken. Then consider the information-flow digraph for correct kens. That is, there is an edge from p to q if, in the computations resulting from some correct ken, either p pushes to q or q pulls from p. Then assume as a new postulate that the lengths of walks in this digraph are bounded.

To avoid the difficulties mentioned earlier with defining the notion of "pulls from" when particular kens are considered (rather than all kens together), let us consider this proposed new postulate in the context where there is no pulling, i.e., `Display` simply isn't in the proclets' vocabulary.

The proposed new postulate implies that there is at most one correct ken. Indeed, if there were two, then they would both satisfy the same recursion conditions, for recursion over the levels of the information-flow digraph for correct kens, just as in the proof of Theorem 7.22. But the postulate does not imply that there exists any correct ken at all. So something more must be said before we can adopt the Updates Postulate in this context. There seem to be three moderately plausible ways to proceed.

One is to add, as a further postulate, that there exists a correct ken. The combination of these two new postulates is easily seen to be equivalent to the

previously considered combination of Unique Kens and Bounded Sequentiality for the correct ken. So this approach runs into the problems described earlier in this section.

A second approach is to admit the possibility that there is no correct ken and to modify the Updates Postulate to take this possibility into account. It seems natural to suppose that, when there is no correct ken, no updates are performed: $\tau_A(X) = X$, the system crashes. But there are two problems with this idea. First, any specific convention of this sort seems inappropriate in a general axiomatic description of arbitrary parallel algorithms. Second, it requires a good deal of work by the "operating system" to detect the absence of a correct ken. Work of this sort is not otherwise required, and one might ask whether an operating system with such power couldn't just do the whole calculation by itself without using proclets.

Finally, one could assume that the information-flow digraph G for correct kens not only exists but is given as part of the state. Then the algorithms for proclets could be arranged so as to always communicate along edges of G. That is, a rule of the form `Push` t_0 `to` t_1 could be replaced by one that does the sending only if G has an edge from `me` to t_1, and a term of the form `Display`(t) could be replaced with one that has the same value when G has an edge from t to `me` but is `undef` otherwise. With these changes, all pushing and pulling for any ken whatsoever will be along edges of G. In other words, the information-flow digraph in our original sense, defined using all kens, becomes a subgraph of G. Thus, the assumed bound on the lengths of walks in G gives a bound on the lengths of walks in the information-flow digraph, and we have recovered the original Bounded Sequentiality Postulate.

13. WHAT ELSE IS THERE?

After the treatment of sequential algorithms in Gurevich [2000] and the treatment of parallel algorithms in the present article, it is natural to ask what other kinds of algorithms remain to be treated. An obvious answer is algorithms of the sorts that we have specifically excluded from consideration in the main part of this article. The first exclusion was distributed algorithms in which agents act asynchronously. The general analysis of such algorithms from (something resembling) first principles remains a major task for future research. The second exclusion was algorithms where single steps can involve unbounded sequentiality, like the submachine algorithms in Section 11.

We have also not mentioned quantum algorithms, but Grädel and Nowack [2003] have shown that they can be modeled by ASMs (with an external function providing random bits); in fact, quantum states correspond to states in the sense of the present paper (so there are a great many of them in any nontrivial quantum algorithm). We have also not mentioned real-time algorithms and other algorithms involving continuous variation (analog algorithms), even though experience indicates that such processes also admit ASM models.

Does our main theorem cover all algorithms except these excluded ones? That is, are all other algorithms parallel algorithms in the sense defined at the

end of Section 7? Do they satisfy all our postulates? For the rest of this section, as in most of this article, we work with algorithms that are synchronous (in the sense that they proceed in a discrete sequence of steps) and that do not involve unbounded sequentiality within single steps.

Since the algorithms are synchronous, it is reasonable, as explained in Gurevich [2000], to assume the Sequential Time and Abstract State Postulates. What else can one say? We shall argue that, except in the case of sequential algorithms covered in Gurevich [2000], such an algorithm must involve substantial parallelism. This argument, by itself, will not prove that such an algorithm is necessarily a parallel algorithm in the sense of Definition 7.28, since it does not sufficiently delineate what happens in and between the proclets. We comment afterward on where such a delineation could come from.

Consider an algorithm, subject to the Sequential Time and Abstract State Postulates, but not satisfying the Bounded Exploration Postulate and therefore not a sequential algorithm. We further assume that the change in the state from one computation step to the next can be regarded as the result of some number of atomic events. By atomic events, we mean things like reading the content of a given location in the state, or writing a new value into a location, or sending or receiving a single piece of information. One such event may be a necessary prerequisite for another. For example, the first event may be reading a certain value which serves as a component of the location where the second event is to read or write. The relation "necessary prerequisite for" is a strict partial ordering of the atomic events in any one step of the computation. Having prohibited unbounded sequentiality, we have an upper bound B on the lengths of chains in this partial order. (B depends only on the algorithm, not on the state.) On the other hand, the number of atomic events should not be bounded independently of the state, since the algorithm is assumed not to have bounded exploration.[6]

But a partially ordered set of size S with no chains of length greater than B must have an antichain (i.e., a set of pairwise incomparable elements) of size at least S/B. Thus, our partially ordered sets of events must have, as the state varies, arbitrarily large antichains. And what is an antichain in one of these partially ordered sets? It is exactly a set of atomic events none of which is a prerequisite for another. That is, it is a set of atomic events that could be executed in parallel. It is in this sense that an algorithm with bounded sequentiality but unbounded exploration must involve substantial parallelism.

The parallelism found by this argument differs in two respects from the parallelism of proclets discussed in the preceding sections. First, proclets can be bigger than atomic events. And we took advantage of this, for example in saying (in Remark 7.17) that several species of proclets executing different algorithms can be regarded as executing the same algorithm, namely an

[6]It is imaginable that the number of atomic events is bounded yet Bounded Exploration fails, since the atomic events might not be given by a fixed finite set of terms. In this situation, the Bounded Exploration postulate of Gurevich [2000] would not fulfill its intended purpose of expressing that the work done in one step is bounded. In the general context of Gurevich [2000], there was no measure of "work" separate from exploration. In our present context, atomic events can be used to measure work, and a discrepancy between this measure and bounded exploration would suggest a problem with the latter.

`if`-chain of the other algorithms. Clearly, such a chain is not an atomic event. To get from parallel atomic events to the axioms used in this article, one would need to assemble atomic events into larger proclets while preserving a certain degree of independence, so that proclets can be executed in parallel except for the temporal ordering imposed by the pushing and pulling of information.

Second, there is no uniformity in the atomic events. Our argument leading to many parallel atomic events made no use of the fact that the entire algorithm is described by a finite text. This finiteness should imply that, if many things are to occur simultaneously, they must be specified in a finite number of ways, given by the algorithm itself, with only some parameters varying. If this conclusion could be made rigorous, it would bring us quite close to the idea of proclets, all executing one of a few algorithms (reducible to one algorithm by the `if`-chain idea) but with a varying parameter `me`.

We summarize these admittedly imprecise arguments in the imprecise conclusion that, if an algorithm is to work in sequential time, with bounded sequentiality in each step, then either it is a sequential algorithm in the sense of Gurevich [2000], or it is a parallel algorithm in the sense of the present article, or it does something very strange, something quite different from the behavior of currently existing algorithms.

REFERENCES

AGRAWAL, M., KAYAL, N., AND SAXENA, N. 2002. PRIMES is in P, To appear, available at `http://www.cse.iitk.ac.in/news/primality_v3.pdf`.

ASM WEB. The ASM Michigan Web Page. Edited by James K. Huggins. URL: www.eecs.umich.edu/gasm.

BLASS, A. AND GUREVICH, Y. 2000. The Logic of Choice. *J. Symbolic Logic 65*, 3 (September), 1264–1310.

BLASS, A., GUREVICH, Y., AND SHELAH, S. 1999. Choiceless Polynomial Time. *Annals of Pure and Applied Logic 100*, 141–187.

BÖRGER, E. AND SCHMID, J. 2000. Composition and Submachine Concepts for Sequential ASMs. In *Computer Science Logic (Proceedings of CSL 2000)*, P. Clote and H. Schwichtenberg, Eds. LNCS, vol. 1862. Springer-Verlag, 41–60.

DAWAR, A. AND GUREVICH, Y. 2001. Fixed Point Logics. *Bull. Symbolic Logic 8*.

FOUNDATIONS OF SOFTWARE ENGINEERING GROUP, MICROSOFT RESEARCH. AsmL Web Page. URL: research.microsoft.com/foundations/#AsmL.

GRÄDEL, E. AND GUREVICH, Y. 1998. Metafinite Model Theory. *Information and Computation 140*, 1 (10 January), 26–81.

GRÄDEL, E. AND NOWACK, A. 2003. Quantum Computing and Abstract State Machines. In *Abstract State Machines—Advances in Theory and Applications*, vol. 2589 of LNCS, Springer-Verlag, 309–32.

GUREVICH, Y. 1995. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 9–36.

GUREVICH, Y. 1997. May 1997 Draft of the ASM Guide. Tech. Rep. CSE-TR-336-97, University of Michigan.

GUREVICH, Y. 2000. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Trans. Computational Logic 1*, 1 (July), 77–111.

GUREVICH, Y., SCHULTE, W., AND VEANES, M. 2001. Rich Sequential-Time ASMs. In *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, R. Moreno-Díaz and A. Quesada-Arencibia, Eds. Universidad de Las Palmas de Gran Canaria, Canary Islands, Spain, 291–293.

GUREVICH, Y. AND SPIELMANN, M. 1997. Recursive Abstract State Machines. *J. Univ. Comput. Sci. 3*, 4, 233–246.

GUREVICH, Y. AND TILLMAN, N. 2001. Partial Updates: Exploration. *J. Univ. Comput. Sci. 7*, 918–952.

KARP, R. M. AND RAMACHANDRAN, V. 1990. Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. A. Elsevier and M.I.T. Press, 869–941.

LAMPORT, L. 1997. Processes are in the Eye of the Beholder. *Theoret. Comp. Sci. 179*, 333–351.

Received November 2001; revised April 2002; accepted June 2002