ABSTRACT STATE MACHINES CAPTURE PARALLEL ALGORITHMS: CORRECTION AND EXTENSION

ANDREAS BLASS AND YURI GUREVICH

Abstract. We consider parallel algorithms working in sequential global time, like circuits or parallel random access machines. Parallel abstract state machines (parallel ASMs) are such parallel algorithms, and the parallel ASM thesis asserts that every parallel algorithm is behaviorally equivalent to a parallel ASM. In an earlier paper, we axiomatized parallel algorithms, proved the ASM thesis and proved that every parallel ASM satisfies the axioms. It turned out, however, that the latter proof was flawed. We are forced to liberalize our axioms and allow on-the-fly creation of new parallel components. We prove the parallel thesis for the new, larger class of parallel algorithms, and we check that parallel ASMs satisfy the new axioms.

Contents

1. Introduction

Sequential algorithms, like C programs or sequential abstract state machines (sequential ASMs), work in small steps, that is steps of bounded complexity. Parallel algorithms, like circuits or parallel random access machines (PRAMs) or parallel ASMs, work in wide shallow steps. The steps are wide in the sense that the algorithm has no fixed bound on the number of components executing in parallel during a single step. The steps are shallow in the sense that the algorithm has a fixed bound, independent of the input or state, on the number of

actions executed sequentially during a step. The sequential ASM thesis asserts that every sequential algorithm is behaviorally equivalent to a sequential ASM. In [4], the second author axiomatized sequential algorithms by means of three convincing postulates, proved the thesis and checked that every sequential ASM satisfies the postulates. In [1], we extended that line of investigation to the more challenging case of parallel algorithms. We axiomatized parallel algorithms by means of pretty convincing postulates. It was relatively easy to check that the usual parallel models, like circuits or PRAMs or even alternating Turing machines, satisfy the postulates. It was harder to prove the parallel thesis, and — somewhat surprisingly — it was harder to prove that parallel ASMs satisfy the postulates. It turned out that the latter proof was flawed. Analysis revealed the root of the problem: Our postulates allowed dynamic creation of proclets (that is parallel components) but only in the inter-step manner: the next state can have more proclets than the previous one, but the set of proclets does not change within a single step. That was not general enough. Circuits, PRAMs, alternating Turing machines did not require on-the-fly proclet creation but parallel ASMs did. Here we liberalize accordingly the postulates of [1] and thus expand the notion of parallel algorithms. We show that the main theorem, the ASM thesis for parallel algorithms [1, Theorem 10.1], remains true for the new, larger class of parallel algorithms. And we check that parallel ASMs satisfy the new postulates. We use the occasion to correct a couple of other, smaller errors in [1].

In Section 2, we describe the problems with the examples in Sections 8.4 and 8.5 of [1], we show how all but one of the problems can easily be corrected, and we sketch the idea needed to correct the one remaining problem. In Section 3 we develop this idea in detail. This involves modifying the postulates from [1] to allow proclets to create (or activate) additional proclets on the fly. In Section 4, we show that this modification solves the one remaining problem in [1, Section 8]. Finally, in Section 5, we show how to extend the proof of the ASM thesis for parallel algorithms, [1, Theorem 10.1], to the wider class of algorithms defined by our modified postulates.

To avoid excessive repetitions, we assume that the reader is acquainted with the content of [1]. We occasionally give references to particular passages in [1], and we occasionally give reminders about particular points there, but we try to keep these references and reminders minimal.

2. The Problems and How to Solve Them

The flaws that have been found in [1] concern two of the examples in Section 8 of that paper. In this section, we explain what these flaws are, we show how to easily correct all but one of them within the framework of [1], and we indicate why the one remaining flaw requires a liberalization of the framework. Specifically, we first remove the one error from Section 8.4 of [1] and then turn our attention to Section 8.5. The latter section contains two errors. One requires some adjustment of the algorithm proposed in [1], but the other is more serious, requiring the liberalization of the framework to allow intra-step creation of proclets.

Formally, all of these problems in Section 8 of [1] have the same underlying cause: the use of comprehension terms in the description of algorithms. Recall that comprehension terms, expressions of the form $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ denoting multisets, are part of the syntax of parallel ASMs, as introduced in Section 9 of [1]. They are not, however, terms in the ordinary sense of first-order logic, and that is the sense in which "term" is used in the postulates of [1, Section 7]. The errors in the examples in Sections 8.4 and 8.5 arise from treating comprehension terms as though they were terms in the sense of the postulates.

Inspection of the postulates reveals that the notion of term is used there once explicitly and once implicitly. The explicit use is in the last clause of the Background Postulate (page 600), where `Proclet` is required to be a variable-free term. The implicit use is in the Proclets Postulate (pages 605–606) which requires the proclets to execute a sequential algorithm with output. The definition of sequential algorithms with output (page 582) incorporates the Bounded Exploration Postulate from [4] ("Bounded Sequentiality" at this point in [1] is a typo), which requires a bounded exploration witness consisting of finitely many terms. These sequential algorithms with output are equivalent to sequential ASMs with `Output` rules, as explained on page 631 of [1]. These ASMs use ordinary terms (of the vocabulary described in the Proclets Postulate), not comprehension terms, in their updates, guards, and outputs, because the bounded exploration witness consists of ordinary terms. With these observations in mind, it is easy to see what is wrong in the examples of Sections 8.4 and 8.5.

We begin with Section 8.4, which dealt with first-order and fixed-point logic. The treatment of first-order logic is correct, but the explanation (on page 621) of a step of the induction leading to a fixed point used a comprehension term in the update of $\texttt{temp}(p)$. The context there (page 621) is that a proclet p is to calculate an inflationary

fixed point $\mathrm{IFP}_{P,\bar{x}}(\varphi(P,\bar{x}))(\bar{t})$ by calculating the successive iterates

$$\Phi^0 = \varnothing \quad \text{and} \quad \Phi^{n+1} = \Phi^n \cup \{\bar{a} : \varphi(\Phi^n, \bar{a})\}$$

at successive steps of the algorithm under construction. Other proclets are available to carry out the subsidiary calculations associated with φ and its subformulas. The step from Φ^n to Φ^{n+1}, as described in [1] involved a comprehension term, in which the proclet p converts its mailbox (the multiset of results from the subsidiary computations using Φ^n) into the desired Φ^{n+1}.

Fortunately, the comprehension term here becomes unnecessary if we modify slightly the format in which p stores (as $\mathtt{temp}(p)$ in the notation of [1]) the sets Φ^n and the activity of the subsidiary proclets that compute the value of φ.

Instead of taking $\mathtt{temp}(p)$ to be the set Φ^n itself, let $\mathtt{temp}(p)$ be the characteristic function of this set, regarded as a set of ordered pairs, i.e.,

$$\{\langle \bar{a}, \mathtt{true}\rangle : \bar{a} \in \Phi^n\} \cup \{\langle \bar{a}, \mathtt{false}\rangle : \bar{a} \in M^k - \Phi^n\},$$

where M is the base set of the structure in which the formula is to be evaluated and where k is the arity of P.

Also, when a subsidiary proclet has computed the truth value v of φ at certain arguments $\bar{a}$, using Φ^n as the interpretation of P, let it push to p the pair $\langle \bar{a}, v'\rangle$ where v' is the disjunction of v and the truth value of $\Phi^n(\bar{a})$.

Note that this v' is the truth value of $\bar{a} \in \Phi^{n+1}$, and so the ordered pair $\langle \bar{a}, v'\rangle$ is one of the pairs that p should have in $\mathtt{temp}(p)$ at the next step of the computation. Thus, the update to be performed by p is simply to give $\mathtt{temp}(p)$ the value of $\mathtt{myMail}$ (provided this is different from the previous value of $\mathtt{temp}(p)$). No comprehension term is needed.

Technically, we should mention another modification in the work of the subsidiary proclets for φ and its subformulas. These proclets use the current value of Φ^n, which they pull from p. With the new format for storing Φ^n in $\mathtt{temp}(p)$, the subsidiary proclets will, of course, have to take this format into account in their computations. This modification has no effect on the discussion in [1], because that discussion didn't include such fine details about the work of the subsidiary proclets.

We turn next to the first difficulty in Section 8.5, namely in the work of a term-proclet $\langle \hat{u}, \bar{a}\rangle$ where u is a comprehension term $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ and where $\bar{a}$ is a list of values for the pseudo-free variables of u. (In [1] we wrote t for what we now call u; the new notation avoids confusion with $t(x)$.) This term-proclet $\langle \hat{u}, \bar{a}\rangle$ has the task of computing the value of u when its pseudo-free variables are given the values $\bar{a}$, and [1] contains a three-part recipe for how to do this. The

first two parts of the recipe are correct: Activate the proclet $\langle \hat{r}, \bar{a} \rangle$, which will provide a value for r, say b. Then activate, for each $c \in b$, the proclets $\langle \widehat{t(x)}, \bar{a}^\frown c \rangle$ and $\langle \widehat{\varphi(x)}, \bar{a}^\frown c \rangle$, which will provide the values of $t(c)$ and $\varphi(c)$. The error is in the final step, where the right $t(c)$'s, namely those corresponding to c's for which $\varphi(c) = \texttt{true}$, are picked out, assembled into a set, and equipped with the correct multiplicities. As it stands, this requires the use of comprehension terms, which are unavailable to proclets.

The key idea for repairing the problem is to arrange for the proclet $\langle \hat{u}, \bar{a} \rangle$ to find the multiset it needs, the value of u, as its mailbox, rather than trying to assemble it. Carrying out this idea will require careful attention to what messages are sent to $\langle \hat{u}, \bar{a} \rangle$ by other proclets. The messages must be just the right $t(c)$'s, each with its right multiplicity. Arranging this will require some additional proclets and static functions, to do some of the work that was previously hidden in a comprehension term, for example the work of picking out the right $t(c)$'s. We postpone the implementation of this idea until Section 4, because the details depend on the repair of the other error in Section 8.5, which we discuss next.

The remaining, serious error is the definition of the "terms" $\texttt{MDA}(p)$ and $\texttt{MA}(p)$ on pages 624–625 of [1]. (There is also an obvious typographical error in the definition of `MDA` which, as it stands, never mentions `MDA`; the intent was to have $\texttt{MDA}(\langle \hat{A}, \bar{a} \rangle)$ in all six places where $\langle \hat{A}, \bar{a} \rangle$ occurs.) With this definition, $\texttt{MDA}(p)$ and $\texttt{MA}(p)$ are not really terms, in the sense required by the postulates for algorithms, because they involve comprehension terms. The intention behind the definitions of `MDA` and `MA` was, as explained in [1], to provide a finite set that contains all proclets that might be activated by the algorithm. There seems to be no way to achieve this intention without using comprehension terms. That is, terms in the correct, first-order sense do not enable us to name, in advance of executing a particular step of the algorithm, a set guaranteed to contain all proclets that might be needed during that step. In other words, an appropriate set of proclets, for a particular step in the computation, can be described only during the execution of this step, not before the step begins. This is the motivation for the following extension of the notion of algorithm.

Do not require the full set of proclets for any step to be given by a term `Proclet` of the algorithm's vocabulary. Instead, require some subset, called *primary proclets*, to be given by such a term, `PriProclet`. Allow primary proclets to activate (or create) new *secondary proclets* during the step. Furthermore, allow secondary proclets to activate

further proclets (which we still call secondary, not tertiary), etc. All the proclets, primary as well as secondary, participate in the activities of proclets described in [1] — pushing and receiving messages, setting up and pulling displays, and updating the state. But only the primary proclets are specified as part of the algorithm's state. The secondary ones are temporary, losing their proclet status at the end of the step. In the terminology of [1, Section 7], the notion of secondary proclet is given by the ken, not the state.

In somewhat more detail, our conventions for the activation of secondary proclets are as follows. To activate new proclets, a proclet p, primary or secondary, must mark for activation a finite subset s of the state. The secondary proclets thereby activated are not, however, the members of s but rather the ordered pairs $\langle q, p\rangle$ where $q \in s$. Thus, a secondary proclet $\langle q, p\rangle$ "knows" its creator p in the sense that it can refer to p by a term in its local state, namely the term `second(me)`.

This liberalization of the notion of algorithm requires changes in several of the postulates and definitions of [1]; the next section spells these changes out.

3. Postulates

The purpose of this section is to modify the definition of algorithms from [1] by allowing proclets to activate other proclets.

Remark 1. The terminology "activate a proclet" was already used in [1], but with a different meaning than here.

In [1] each state of an algorithm determined a set of proclets. The notion of "active" proclet was not part of the state but rather a convenient and intuitive way for us to informally describe certain aspects of a ken, and it could have different meanings in the context of different algorithms. In particular, an inactive proclet was still a proclet and could, despite the terminology, engage in certain basic activities, for example pulling information to see whether it should become active.

In the present paper, each state of an algorithm will determine a set of *primary* proclets. Activation produces additional, *secondary* proclets, which would not be proclets and could not engage in any activities at all if they were not activated. Activation will be a formal notion, an essential ingredient in our postulates, not merely a convenient abbreviation for certain aspects of kens.

The distinction between the two notions of activation may be clarified by noting that the present notion (but not that in [1]) of activating a proclet could as well be called "procletizing an element of the state". □

In broad terms, our modification of the notion of parallel algorithm is to allow any proclet p to mark for activation, during the course of a step, some finite subset $\texttt{Act}(p)$ of the state X. The effect of this activation is that the elements $\langle q, p\rangle$ for $q \in \texttt{Act}(p)$ become proclets in their own right.

For this to make sense, a few observations are in order. First, as in [1], we regard subsets of X as multisets of elements of X in which all the multiplicities are 1. The Background Postulate of [1] ensures (and will continue to ensure even after we modify it below) that finite multisets of elements of a state X are themselves elements of X. In particular, $\texttt{Act}(p)$ is an element of X.

Second, what does it mean for p to "mark" a set? It means to assign that set as the value of a certain dynamic, nullary function symbol `myAct` in its local state. (This `myAct` is not part of the global state of the whole algorithm but of the local state of the individual proclet.) This is exactly like the way proclets produced their displays, $\texttt{Display}(p)$, by setting a value for `myDisplay` in [1]. We adopt the conventions that the initial value of `myAct`, at the beginning of any step, is the empty set and that, if a proclet assigns to `myAct` a value in X that is not a set, then nothing is thereby activated.

Third, not all proclets can arise from this activation process; there must be some proclets already available at the beginning of a step to get the activation process started. The set of these primary proclets is to be given by the value of a term `PriProclet` in the state. We could require that there is always just one primary proclet; if more are wanted, they could be activated by the primary one at the beginning of the step. This theoretical simplification, however, seems to bring no real benefit, and would impose a cost: As was shown in Sections 8.1–8.4 of [1], most known types of parallel algorithms do not require intra-step activation of proclets. If we insisted on starting each step with only a single proclet, we would be forced to include intra-step activation even in algorithms that otherwise have no need for it.

After this broad outline of how we intend to modify the notion of parallel algorithm from [1], we turn to the details of revising the postulates in [1, Section 7] to accommodate this picture of activation of proclets.

The **Sequential Time Postulate**, which was taken unchanged from [4] to [1], requires no changes here.

The **Abstract State Postulate**, also unchanged from [4] to [1], could remain unchanged again, but we take this opportunity to incorporate a small improvement that was first pointed out in [2], namely

that the sets $\mathcal{S}(A)$ of states and $\mathcal{I}(A)$ of initial states must be non-empty.

In the **Background Postulate** of [1], the last item required a variable-free term `Proclet` naming, in each state, a finite set, also called `Proclet`. Modify this by changing `Proclet` to `PriProclet`. This change, though only notational at the current point in the postulates, reflects the fact that only the set of primary proclets is given with the state; secondary proclets are activated during the computation steps. The notation `Proclet` will be used later to refer to the whole set of proclets, primary and secondary.

Remark 2. As in [1], the Background Postulate ensures the availability of (among other things) the operation AsSet, defined as taking any x to the multiset that has the same members as x has but with multiplicity only 1. In [1], we used this operation only when the argument is a multiset, so that it simply removes the multiplicities. In the present paper, there will be an additional use of AsSet, one in which the argument need not be a multiset. If x is not a multiset, then it has no members, and so the definition gives $\mathrm{AsSet}(x) = \varnothing$. It follows that, using AsSet and equality, we can test for sethood; x is a set if and only if $x = \mathrm{AsSet}(x)$. □

In [1, Definition 7.10], a *ken* K of a state X was defined to consist of X together with two functions $\texttt{Mailbox}_K$ and $\texttt{Display}_K$, such that each has the set of proclets as its domain and such that the values of $\texttt{Mailbox}_K$ are multisets. This definition needs significant changes to accommodate activation of new proclets.

The most obvious change is that the ken must include information about which proclets activate which other proclets. Activation takes place entirely within a step. It does not persist across a step boundary from one state to the next; any proclets that should remain proclets for the next step should be added to `PriProclet`. Thus, activation information is not part of the state. It is temporary information to be changed and used by the algorithm within a step, just like the mailboxes and displays of [1]; such information resides in the ken. So a ken K should include a unary function $\texttt{Act}_K$. The intended meaning of $\texttt{Act}_K(p)$ is the set marked for activation, as `myAct`, by the proclet p, but as far as the definition of "ken" is concerned, $\texttt{Act}_K$ is just some function from proclets to sets in X. The intended meaning will be formalized later in the notion of "correct ken". In these respects, $\texttt{Act}_K$ and `myAct` behave just like $\texttt{Display}_K$ and `myDisplay` in [1].

A more complicated change in the notion of ken arises from the fact that the functions $\texttt{Mailbox}_K$, $\texttt{Display}_K$, and $\texttt{Act}_K$ have as their

domain the set of proclets. In [1], that set was given with the state, but now it depends on the ken via the activations described by $\texttt{Act}_K$. This interdependence between kens and the associated sets of proclets accounts for the greater complexity of the following definition compared to the corresponding Definition 7.10 in [1].

Definition 3. A *ken* of a state X consists of X together with three functions, $\texttt{Mailbox}_K$, $\texttt{Display}_K$, and $\texttt{Act}_K$, with a common domain $\texttt{Proclet}_K \subseteq X$ and with values in X, subject to the following requirements.

- The values of $\texttt{Mailbox}_K$ are multisets.
- The values of $\texttt{Act}_K$ are sets.
- $\texttt{PriProclet}_X \subseteq \texttt{Proclet}_K$.
- If $p \in \texttt{Proclet}_K$ and $q \in \texttt{Act}_K(p)$, then $\langle q, p\rangle \in \texttt{Proclet}_K$.
- $\texttt{Proclet}_K$ is the smallest set satisfying the preceding two requirements.

The elements of $\texttt{Proclet}_K$ are called the *proclets* of the ken K.

As usual in recursive definitions, the detailed meaning of the last requirement is that, for all subsets Z of the state X, if $\texttt{PriProclet}_X \subseteq Z \subseteq \texttt{Proclet}_K$ and if, for each $p \in Z$ and each $q \in \texttt{Act}_K(p)$, we have $\langle q, p\rangle \in Z$, then $Z = \texttt{Proclet}_K$.

Notice that the closure condition on $\texttt{Proclet}_K$ incorporates our convention that, if q is in the set $\texttt{Act}(p)$ then it is not q itself but the pair $\langle q, p\rangle$ that becomes a proclet.

Remark 4. This convention allows the proclets $\langle q, p\rangle$ activated by p to "know" which proclet activated them, i.e., they can refer to p in their local states (see Definition 6 below) by means of the term $\texttt{second}(\texttt{me})$. This sort of knowledge seems intuitively reasonable, and it also serves two technical purposes.

First, it ensures that two proclets will not both activate the same secondary proclet for different purposes.

Second, it provides a way for a proclet p to pass information to the proclets it activates, namely by displaying it. Sending messages would not suffice for this purpose, since the number of messages sent by a proclet during a step will be bounded (because the proclets are sequential algorithms with output) while the number of proclets activated by p need not be bounded. So displaying and pulling are the only ways for p to convey information to all the proclets it activates. For this to work, all these proclets must know p, in order to pull the information.

If comprehension terms were available, then this second purpose would not require our convention of automatically indicating the activator in every secondary proclet. Indeed, if such indications were wanted, then instead of marking a set s for activation, the proclet p could mark $\{\!\{\langle q, p\rangle : q \in s : \mathtt{true}\}\!\}$, i.e., p could attach the activator tags on its own. Indeed, p could similarly convey any bounded amount of additional information to the proclets it creates, by building this information into the proclets themselves. But, since comprehension terms are not available to the proclets, this approach will not work. And in any case, it would not achieve the first purpose indicated above. □

Remark 5. One can imagine a more powerful sort of activation, where the tag added to a secondary proclet is not necessarily its activator but some other element of the state chosen by the activator. In addition to `myAct`, there would be another dynamic, nullary function `myTag` (initially `undef`). When a proclet updates `myAct` to a set s and `myTag` to e, the effect is to activate secondary proclets $\langle q, e\rangle$ for $q \in s$.

In this system, it would be possible for several proclets to activate the same secondary proclet, so it would be up to the algorithm to prevent unwanted clashes — for example by always including the activator p as a component of e.

This system provides a powerful means of communication from a proclet p to the proclets q that it activates. It avoids the need for p to display information for these q's since it can build the information into the proclets themselves.

Indeed, this sort of tagging could, with some awkwardness, replace displaying and pulling as a means of communication. One strategy for doing this is as follows. Let all the proclets x that want to pull from p instead send p a message, say of the form $\langle x, \mathrm{pull}\rangle$. Instead of displaying an entity d, p activates new auxiliary proclets $\langle q, \langle d, p\rangle\rangle$ for all q in its mailbox. That is, it updates `myAct` to `myMail` and updates `myTag` to $\langle d, \mathtt{me}\rangle$. Each new proclet $\langle q, \langle d, p\rangle\rangle$ checks whether its first component q is of the form $\langle x, \mathrm{pull}\rangle$. If so, it sends its second component $\langle d, p\rangle$ to x, which interprets this message as meaning that d is the display of p. A careful presentation of this strategy would have to prevent conflicts between the messages used here, to simulate displays, and any messages that the proclets use for other purposes; we refrain from looking at these details. □

In [1, Definition 7.11], we defined the local state of a proclet, given a ken K for a state X. To accommodate activation, we expand the local states of [1] to include a dynamic nullary symbol `myAct` whereby

a proclet indicates the set it wants to activate. The new definition, replacing Definition 7.11 of [1], therefore reads as follows.

Definition 6. Suppose K is a ken of a state X and suppose $p \in \texttt{Proclet}_K$. An *initial local state* for p in X is the structure X plus:

- a static, nullary symbol `me`, interpreted as p,
- a static, nullary symbol `myMail`, interpreted as some multiset in X,
- a static, unary symbol `Display`, interpreted as some unary function $X \to X$,
- a dynamic, nullary symbol `myDisplay`, interpreted as `undef`, and
- a dynamic, nullary symbol `myAct`, interpreted as $\varnothing$.

The *initial local state of p given by K* is the initial local state for p in X where

- `myMail` is interpreted as $\texttt{Mailbox}_K(p)$ and
- `Display` is interpreted as $\texttt{Display}_K$ extended[1] by $\texttt{Display}(x) =$ `undef` for $x \notin \texttt{Proclet}_K$.

Remark 7. The initial local state of p is the only local state of p that we shall work with. In any step of the overall algorithm, a proclet will execute the proclet algorithm only once, in the initial state. The result of this execution can include, in addition to output (messages to other proclets) and updates to the global state (to be executed at the end of the overall algorithm's step), updates to the dynamic symbols `myDisplay` and `myAct`. These updates can be regarded as producing a new non-initial local state, but no computation will be done in that state.

We can therefore, when discussing the state in which a proclet computes, speak of its local state, omitting the word "initial". In fact, in [1], we didn't even introduce "initial" in this context. We have done so here in order to emphasize that these states use the initial default values for `myDisplay` and `myAct`, even in the case of a proclet p and a ken K for which $\texttt{Display}_K(p)$ and $\texttt{Act}_K(p)$ have values different from these defaults. The similarity between the names `Display` and `myDisplay` does not entail any connection between the value of the former (in some ken) and the initial value of the latter. Only in the case of correct kens (defined later) is the syntactic similarity reflected in a semantical connection, and that connection does not involve the initial value of `myDisplay` but the final value, after execution of the

[1]This extension should also have been in the corresponding Definition 7.11 of [1].

proclet algorithm (also defined below). The same comments apply to `Act` and `myAct`. □

As in [1], we occasionally refer to the states of the entire algorithm as *global* states, to distinguish them from the local states of proclets.

Remark 8. There is an analogy between `myAct` and `myDisplay`. Both provide ways for a proclet to make a contribution to the overall ken, specifically to $\texttt{Act}_K$ and $\texttt{Display}_K$ respectively, at least when we deal (as we soon will) with correct kens. But there are two differences. The lesser of the two is that $\texttt{Act}_K$ is required to take sets as values, whereas the dynamic symbol `myAct` could, in principle, denote any element of the state. We shall (in the definition of "correct" ken) adopt the convention that if a proclet p gives `myAct` a value that is not a set, then the resulting value of $\texttt{Act}_K(p)$ should be $\varnothing$.

The second difference is that $\texttt{Display}_K$ is part of the initial local state given by K but $\texttt{Act}_K$ is not. That is, a proclet has access to what other proclets have displayed but not to the activations performed by other proclets. Why?

Allowing proclets to know what other proclets activate would introduce a sneaky means of communication. A proclet p could activate another proclet q, not so that q would participate in the computation but rather so that other proclets, seeing that q is activated, would be able to infer some information that p wants to transfer to them. If we are not careful, such communication could make the computation process circular, i.e., there might be no correct ken. For example, p might activate q if and only if it has no incoming message, while r might send a message to p if and only if q is activated. If there are no other proclets that might send a message to p, then no correct ken is possible; the message-sending and activating specified by the proclets' programs are circular and contradictory.

This particular example could be rendered harmless by acknowledging that p is sending information to r and putting an edge from p to r in the information flow digraph. That edge, together with the one arising from the message r might send to p, would constitute a cycle in that digraph, contrary to the Bounded Sequentiality Postulate. So the example would be excluded by this postulate.

But the general problem cannot be removed so easily. If proclets could know, in general, about each other's activations, then the information flow digraph should have edges from every proclet to every proclet — a monstrous contradiction to Bounded Sequentiality. To avoid such a disaster, we would have to perform an analysis of just which

proclets really do (in some ken) get information from which other proclets. Such an analysis would be essentially the same as the analysis leading to the definition of "pulls from", Definition 7.18 in [1].

There is a simpler way to get activation information to those proclets that might need it: If other proclets should find out what p has activated, let p incorporate $\texttt{Act}(p)$ into its display for the other proclets to read. In this way, the analysis mentioned above is subsumed by the analysis leading to "pulls from", there is no need to include sneaky transmission of information in the information flow digraph, and all communication between proclets still fits the push and pull paradigms of [1]. □

The Proclets Postulate of [1] needs one evident addition and some reorganization. The addition is that the initial local state in which a proclet operates should contain the dynamic, nullary symbol `myAct`. The reorganization arises from the following considerations along with a desire to stay conceptually close to the picture in [1]. The Proclets Postulate of [1, Section 7.3] was written so as to refer only to states, not to kens. Thus, for example, it says that `myMail` should denote (in the local state of a proclet p) some multiset, not that this multiset should be $\texttt{Mail}_K(p)$ for a specific ken K. Kens appeared in the postulates of [1] only in the subsequent Section 7.4, which dealt with interaction between proclets. In our present context, we must give up either this ken-independence of the Proclets Postulate or the mention of proclets in the postulate. The reason is, of course, that the notion of proclet now depends on the ken and not merely on the state. We choose the second option. That is, we retain the general structure of the Proclets Postulate, referring only to the state, and we therefore postpone any mention of the proclets themselves in postulates.

Because the postulate is not about proclets but only about their algorithm, we rename it as the **Proclet Algorithm Postulate**: The algorithm A determines a single sequential algorithm with output, called the *proclet algorithm*, in the vocabulary of the global algorithm plus the static nullary symbols `me` and `myMail`, the static unary symbol `Display`, and the dynamic nullary symbols `myDisplay` and `myAct`. The outputs of the proclet algorithm are ordered pairs $\langle \text{addressee}, \text{content} \rangle$ of elements of the state.

During any step of the (overall) algorithm, each element p in the current state X is to be regarded as potentially executing the proclet algorithm for one step in a state consisting of X, p as the denotation of `me`, some multiset as the denotation of `myMail`, some unary function as the denotation of `Display`, `undef` as the initial value of `myDisplay`,

and $\varnothing$ as the initial value of `myAct`. "Potentially" here refers to the fact that, once we define the correct ken K, only the elements of $\texttt{Proclet}_K$ will actually run the proclet algorithm. Also, once we define the correct ken K, the initial local state in which a proclet p operates will be the initial local state given by K as in Definition 6.

We emphasize that, in each step of the global algorithm, the proclet algorithm is to be executed for only one step by each proclet. Accordingly, it will be very convenient to use the following abbreviation.

Definition 9. Let K be a ken of a state X, and let $p \in \texttt{Proclet}_K$. We abbreviate "one step of the proclet algorithm is executed in the initial local state of p given by K" as "p fires in K." Here "one step is executed" means that the transition function is applied and outputs are produced once; there is no iteration as in runs of an algorithm.

The remarks following the Proclet Algorithm Postulate in [1, Section 7.3] apply here with the obvious additions, saying that `myAct` behaves like `myDisplay`, being updated during (rather than at the end of) a step and not retaining its value past the end of a step.

Section 7.4 of [1], about the interaction between proclets, needs substantial modification, especially because the very notion of proclet now depends on a sort of interaction, namely activation.

The first modification here, in Definition 7.18 of "q pulls from p", is rather minor. The idea behind the definition was that q pulls from p in state X if there are two kens for X, differing only in the values of $\texttt{Display}(p)$, such that q behaves differently in its initial local states given by these kens. We do not change this idea, but the detailed meaning of "behaves differently", namely producing different updates or sending different mailings, must now be extended to include activating different elements. Formally, the required change in Definition 7.18 is that "different updates (in the global state or in `myDisplay`)" becomes "different updates (in the global state, in `myDisplay`, or in `myAct`)".

In addition, we explicitly require the two kens in the definition to agree that p and q are proclets. In fact, this was already implicit. The definition involves $\texttt{Display}(p)$, which is defined only when p is a proclet. It also involves firing q, i.e., executing q in appropriate initial local states; these states involve $\texttt{Mailbox}(q)$, which is defined only if q is a proclet.

Definition 10. Let p and q be elements of a state X. Then q *pulls from* p if there are two kens K and K' such that

- both p and q are in $\texttt{Proclet}_K \cap \texttt{Proclet}_{K'}$,
- K and K' differ only in the values of $\texttt{Display}(p)$, and

- when q fires in K and in K', the results differ either in updates (of the global state or `myDisplay` or `myAct`) or in mailings (counting multiplicities).

The information flow digraph of [1, Definition 7.19] should be modified so that if p activates q then there is an edge from p to q. Intuitively, something has flowed from p to q, perhaps "activeness" or "proclethood"; it is not obvious whether this consitutes information. But the following example shows that, whether or not we call it information, it must be included in the digraph.

Example 11. Suppose there is just one primary proclet p, which activates another proclet q if and only if p's mailbox is empty (and p does nothing else). Suppose further that q's computation is just to send a message to p. So the information flow digraph has an edge from q to p, representing the possibility of a mailing, and we claim that it should also have an edge from p to q, representing the possibility of activation. By including this second edge, we introduce a cycle into the digraph, so that the Bounded Sequentiality Postulate is violated and the instructions we gave for these two proclets do not constitute an algorithm. Without the second edge, the instructions would satisfy the postulate. Our claim is that the former outcome, "not an algorithm," is correct. The reason is that these instructions cannot be consistently executed. p must activate q if and only if p's mailbox is empty, which happens if and only if q is not activated. (Once we define correctness of kens, we can say that the circularity prevents the existence of a correct ken.)

Taking into account the preceding observations, the dependence of the notion of proclet on the ken, and our desire to remain as close to [1] as these considerations permit, we are led to the following definition of the information flow digraph, replacing [1, Definition 7.19].

Definition 12. Let X be a (global) state. Its *information flow digraph* has as vertices all those elements $p \in X$ that belong to $\texttt{Proclet}_K$ for at least one ken K. There is an edge from p to q if at least one of the following conditions is satisfied.

- q pulls from p.
- There is a ken K, for which both p and q are proclets, such that, when p fires in K, it sends a message to q.
- q is of the form $\langle r, p\rangle$ and there is a ken K, for which p is a proclet, such that, when p fires in K, it updates `myAct` to a set containing r.

Remark 13. The three clauses defining the edges of the information flow digraph correspond to information flow by pulling, pushing, and

activating respectively. In all three clauses, as well as in the definition of vertices, we have included everything that *might* be involved for *some* reasonable ken. □

Remark 14. We explain briefly why our formulation of the third clause in Definition 12, the one about activation, is preferable to two plausible-looking alternatives. The first alternative is to replace the part about p updating `myAct` to a set containing r with the simpler requirement $r \in \mathtt{Act}_K(p)$. The second is to require not only p but also q to be a proclet of K. (Our definition requires q to be a proclet of some ken, in order to be a vertex of our digraph, but it need not be a proclet of the same ken K mentioned in the third clause.) Readers for whom our definition is obviously better than these alternatives are invited to skip the rest of this remark.

The defect in both alternatives arises from the fact that, in arbitrary kens K (as opposed to the correct kens, which will be defined later), no connection is required between the function $\mathtt{Act}_K$ and the results of the proclets' computations of `myAct` in their local states given by K.

On the one hand, this means that we could have $r \in \mathtt{Act}_K(p)$ even when r is not at all the sort of thing that p might activate under the proclet algorithm, indeed, even if the proclet algorithm is such that `myAct` can never be updated. As a result, the first proposed alternative could put into our digraph a great many activation edges that have nothing to do with any activation that the algorithm would ever perform. Indeed, we would have edges from each vertex p to all vertices of the form $\langle r, p\rangle$, for arbitrary r in the state. In this situation, the Bounded Sequentiality Axiom would be excessively difficult to satisfy.

On the other hand, if a proclet p, firing in K, activates $\langle r, p\rangle$ by updating `myAct` to a set that contains r, there is no guarantee that $r \in \mathtt{Act}_K(p)$, and therefore there is no guarantee that $\langle r, p\rangle$ is a proclet of K. As a result, the second proposed alternative would miss many possible activations.

Both alternative formulations would be admissible if we were working only with correct kens, but we are not, and for good reason. The existence and uniqueness of correct kens (Theorem 22 below) depends on the Bounded Sequentiality Postulate, which in turn is formulated in terms of the information flow digraph. For more information about the need to work with all kens rather than only correct ones, see Sections 7.4 and 12 of [1]. □

With the revised definition of the information flow digraph, we can retain the **Bounded Sequentiality Postulate** from [1]: There is a uniform bound B, depending only on the algorithm and not on the

state, for the lengths of all directed walks in the information flow digraphs of global states.

Definition 15. Let X be a state and p a vertex of its information flow digraph. The *level* of p in this digraph is the length of the longest walk in the digraph that ends at p. By *length* here we mean (as in [1]) the number of vertices in the walk, not the number of edges, so levels begin with 1, not 0.

Clearly, the level of p always exists and is no greater than the uniform bound B given by the Bounded Sequentiality Postulate. It also follows immediately from the definition that, if the information flow digraph has en edge from p to q then the level of p is strictly smaller than that of q.

Our next task is to prove the analog, in our new situation, of Theorem 7.22 of [1], which asserts the existence of a unique correct ken for each state. We begin with a definition of correctness, just like that in [1] except that it takes activation into account, both explicitly in a clause relating `Act` to `myAct` and implicitly by using the new notion of ken.

Definition 16. Let X be a global state and K a ken for X. Then K is a *correct* ken for X if the following three conditions are satisfied.

- For each $p \in \mathtt{Proclet}_K$, the members of $\mathtt{Mailbox}_K(p)$ are the messages sent to p by the proclets $q \in \mathtt{Proclet}_K$ when each such q fires in K. (The multiplicity of a message m in $\mathtt{Mailbox}_K(p)$ is the sum, over all $q \in \mathtt{Proclet}_K$, of the multiplicity with which q sends m to p, i.e., the multiplicity of $\langle p, m\rangle$ in the output multiset of q's execution of the proclet algorithm).
- For each $p \in \mathtt{Proclet}_K$, the value of $\mathtt{Display}_K(p)$ is the value that `myDisplay` obtains when p fires in K.
- For each $p \in \mathtt{Proclet}_K$, the value of $\mathtt{Act}_K(p)$ is the value that `myAct` obtains when p fires in K, provided this value is a set. Otherwise, $\mathtt{Act}_K(p) = \varnothing$.

For technical purposes, it is useful to have a name for the following consequence of correctness.

Definition 17. Let X be a global state and K a ken for X. Then K is a *plausible* ken for X if, whenever $p \in \mathtt{Proclet}_K$ and $q \in \mathtt{Act}_K(p)$, then $\langle q, p\rangle$ is at a higher level than p in the information flow digraph for X.

In connection with this definition, notice first that from $p \in \mathtt{Proclet}_K$ and $q \in \mathtt{Act}_K(p)$ it follows that $\langle q, p\rangle$ is a proclet of K and is therefore a vertex of the information flow digraph.

Notice also that, if K satisfies the third requirement in the definition of "correct", then it is plausible. Indeed, if we assume the third requirement and also assume $p \in \texttt{Proclet}_K$ and $q \in \texttt{Act}_K(p)$, then p firing in K produces a value of `myAct` that contains q. That provides an edge in the information flow digraph from p to $\langle q, p\rangle$ and thus ensures that $\langle q, p\rangle$ is at a higher level than p.

Before proving that every state has a unique correct ken, we record some preliminary information that will be needed in that proof.

Definition 18. Let K be a ken for a state X, and let l be a positive integer. $H(K, l)$ is defined to be the ken that is the same as K except that $\texttt{Display}_{H(K,l)}(p) = \texttt{undef}$ for those $p \in \texttt{Proclet}_K$ that have level $\geq l$ in the information flow digraph for X. We refer to $H(K, l)$ as the result of *hiding the displays* of K from level l up.

Notice that $\texttt{Act}_{H(K,l)} = \texttt{Act}_K$ and therefore $\texttt{Proclet}_{H(K,l)} = \texttt{Proclet}_K$. In particular, in passing from K to $H(K, l)$, we need not adjust the domains of the functions.

Lemma 19. *Let K be a ken for a state X, p a proclet of K, and l its level in the information flow digraph of X. Consider the one-step executions of the proclet algorithm by p in two initial local states, the one given by K and the other by $H(K, l)$. These two executions produce the same updates (to the global state and to* `myDisplay` *and* `myAct`*) and the same output multiset of messages.*

Proof. Inspection of the definitions of initial local states and hiding reveals that the two initial local states mentioned in the lemma differ only in the values of $\texttt{Display}(q)$ when $q \in \texttt{Proclet}_K = \texttt{Proclet}_{H(K,l)}$ and q has level $\geq l$. Of these q's, only finitely many are relevant to the updates and outputs produced when p executes the proclet algorithm, because this algorithm, being a sequential algorithm with output, satisfies the Bounded Exploration Postulate. So we can connect K to $H(K, l)$ by a finite sequence of kens, in which

- at the first step we pass from K to a ken in which $\texttt{Display}(q)$ has been changed to `undef` for all the irrelevant q's of level $\geq l$, so the new ken differs only finitely from $H(K, l)$, and
- at each subsequent step, we change the value of $\texttt{Display}(q)$ for only one $q \in \texttt{Proclet}_K$.

We already know that at the first step in this sequence the updates and outputs of p are unchanged. It remains to check that the same is true at all subsequent steps in our sequence of kens. So consider, for the rest of this proof, a particular pair of consecutive kens in the rest of our

sequence; suppose, toward a contradiction, that they do not agree as to p's computation; and let q be the unique element where their `Display` functions differ.

Because all the kens in the sequence have the same `Act` function, they all have the same proclets as K. In particular, our q is a proclet for both of the consecutive kens under consideration. This and the assumed disagreement as to p's computation imply that p pulls from q and so there is an edge from q to p in the information flow digraph. This is absurd, as q has level $\geq l$ and p has level l. □

Lemma 20. *Let X be a state, K and K' two plausible kens for it, and l a positive integer. Assume that, for all $q \in \texttt{Proclet}_K \cap \texttt{Proclet}_{K'}$,*

- *if q has level $\leq l$ then $\texttt{Mailbox}_K(q) = \texttt{Mailbox}_{K'}(q)$,*
- *if q has level $< l$ then $\texttt{Display}_K(q) = \texttt{Display}_{K'}(q)$, and*
- *if q has level $< l$ then $\texttt{Act}_K(q) = \texttt{Act}_{K'}(q)$.*

Let $p \in \texttt{Proclet}_K$ have level l. Then p is also in $\texttt{Proclet}_{K'}$, and the executions of p in its initial local states given by K and by K' produce the same updates and the same outputs.

Proof. We proceed by induction on l and observe that our assumptions about K and K' imply that the same assumptions also hold if l is replaced by any smaller l'. So, by induction hypothesis, any proclet of K with level $< l$ is also a proclet of K'. Since p is a proclet of K, it is either in $\texttt{PriProclet}_X$ or activated by some other $q \in \texttt{Proclet}_K$ (i.e., $p = \langle r, q\rangle$ for some $r \in \texttt{Act}_K(q)$). In the former case, we immediately have that p is also a proclet of K', because `PriProclet` is given by the state, not the ken. In the latter case, the assumption of plausibility implies that q has level $< l$. So $q \in \texttt{Proclet}_{K'}$ and, by hypothesis, $\texttt{Act}_K(q) = \texttt{Act}_{K'}(q)$. In particular, q activates p in K', and so $p \in \texttt{Proclet}_{K'}$, as desired.

To prove that p produces the same updates and outputs whether it fires in K or in K', we may, thanks to Lemma 19, work with the kens $H(K, l)$ and $H(K', l)$ obtained by hiding displays from level l up. But we claim that the initial local states of p given by these kens coincide. Clearly, proving the claim would suffice to complete the proof of the lemma.

To prove the claim, notice first that the only effect of a ken on the initial local state of p is to provide the interpretations of `myMail` and `Display`. Since p is at level l, the first assumption of the lemma ensures that K and K' and therefore also $H(K, l)$ and $H(K', l)$ agree as to $\texttt{Mailbox}(p)$, which provides the value of `myMail` for p's initial local state. As for `Display`, the second assumption of the lemma ensures

that all of K, K', $H(K,l)$, and $H(K',l)$ agree as to the displays of those elements of level $< l$ that are proclets with respect to both K and K'. But the induction hypothesis implies, as in the first paragraph of this proof, that an element of level $< l$ is a proclet for all or none of K, K', $H(K,l)$, and $H(K',l)$. So, for q of level $< l$, $\texttt{Display}(q)$ will have the same value for all of K, K', $H(K,l)$, and $H(K',l)$, either by the second assumption of the lemma or because of the convention, in the definition of initial local states, that `Display` of a non-proclet is always `undef`. The same convention combined with the definition of hiding ensures that $\texttt{Display}(q)$ has the same value, namely `undef`, for $H(K,l)$ and $H(K',l)$ whenever q has level $\geq l$. (Notice that such a q might be a proclet for just one of K and K'. In that case, $\texttt{Display}_{H(K,l)}(q)$ and $\texttt{Display}_{H(K',l)}(q)$ would be `undef` for different reasons — once because of hiding and once because of the convention concerning `Display` of non-proclets. The need to deal with this situation blocks the easier argument of [1, Lemma 7.25] and motivated the notion of hiding.) This completes the proof that the initial local states of p given by $H(K,l)$ and $H(K',l)$ are identical, as required. □

Remark 21. The hypothesis of the lemma may seem a bit awkward, because it concerns `Mailbox` up to and including level l but `Display` and `Act` only strictly below level l. We claim, however, that this situation is semantically natural even if syntactically awkward. The reason is that the parts of the ken covered by this hypothesis at stage l are exactly the parts relevant to the computations done by elements p at level l. Indeed, the information relevant to the computation of such a p consists of (1) whether p is a proclet and therefore should be doing a computation at all, (2) the mailbox of p, and (3) the displays that p reads. Here (1) involves `Proclet` at level l (namely at p), which is determined by `Act` at strictly lower levels (because of plausibility); (2) involves `Mailbox` at level l; and (3) involves `Display` at lower levels.

Another way to view the situation is that the hypothesis covers exactly the information that is determined, in a correct ken, by the activity of proclets at strictly lower levels than l. As a special case, for $l = 1$, we have just the information that is determined by the state without any need for computation by proclets, namely the proclets at level 1 (determined by `PriProclet` as plausibility prevents any activation of proclets of level 1) and their mailboxes (empty as nothing can send messages to them).

The general structure of this hypothesis, referring to `Mailbox` (and implicitly to `Proclet`) for one level higher than `Display` and `Act`, will recur in subsequent arguments. □

Theorem 22. *For every state X, there is a unique correct ken.*

Proof. We first prove uniqueness. Suppose both K and K' are correct kens for the state X. We prove, by induction on l, that:

(1) $\texttt{Proclet}_K$ and $\texttt{Proclet}_{K'}$ have the same elements of levels $\leq l$. So for levels $\leq l$ we can speak of proclets without specifying which of the two kens we mean.
(2) $\texttt{Mailbox}_K$ and $\texttt{Mailbox}_{K'}$ agree on all proclets of level $\leq l$.
(3) $\texttt{Display}_K$ and $\texttt{Display}_{K'}$ agree on all proclets of level $< l$.
(4) $\texttt{Act}_K$ and $\texttt{Act}_{K'}$ agree on all proclets of level $< l$.

For the base case, $l = 1$, items (3) and (4) are vacuous. Item (1) follows from the fact that a proclet at level 1 cannot be activated by another proclet in a correct (or just plausible) ken. So at level 1 the proclets for any plausible ken are just those in $\texttt{PriProclet}_X$. Item (2) at level 1 follows from the fact that messages always go from lower to higher levels (because they produce edges) in the information flow digraph. In particular, no message ever goes to a proclet of level 1. Since K and K' are correct, it follows that $\texttt{Mailbox}_K = \texttt{Mailbox}_{K'} = \varnothing$.

For the induction step, assume the assertions hold for a certain l. As noted in item (1), it makes sense to speak of proclets at level l without specifying K or K'. For each such proclet p, we can apply Lemma 20 to conclude that its updates and outputs are the same whether K or K' provides its initial local state. By correctness, it follows that $\texttt{Display}_K(p) = \texttt{Display}_{K'}(p)$ and $\texttt{Act}_K(p) = \texttt{Act}_{K'}(p)$. Thus, we have items (3) and (4) for $l + 1$ in place of l.

Furthermore, the mail sent by p is the same for K as for K'. Since this holds for all proclets p at level l and also for all proclets at lower levels (by the same argument), and since proclets at level $l + 1$ can receive mail only from proclets at levels $\leq l$ (because message-sending produces edges in the information flow digraph), we conclude that every proclet at level $l + 1$ receives the same messages, with the same multiplicity, whether the proclets are firing K or in K'. Another application of correctness gives us item (2) for proclets at level $l + 1$. Since we already had this item for proclets of lower level, we now have item (2) with $l + 1$ in place of l.

Finally, since proclets can be activated only by proclets of lower level, and since $\texttt{PriProclet}$ depends only on the state, not the ken, the proclets at level $l+1$ are determined by $\texttt{Act}(q)$ for proclets of levels $< l + 1$. Thus, item (1) for $l + 1$ follows from item (4) for $l + 1$.

This completes the induction and thus, if we take l to be the number of levels, the proof of uniqueness. To prove existence, we construct

the desired K by induction on levels l. After stage l, we will have constructed a ken $K(l)$, intended to have the following properties. Its proclets are the proclets of the correct ken up to and including level l, and its `Mailbox` function agrees with that of the correct ken on these proclets. Furthermore, its `Display` and `Act` functions agree with those of the correct ken on proclets at levels $< l$. On proclets of level l, its `Display` and `Act` functions have the default values `undef` and $\varnothing$, respectively.

For $l = 1$, let $\texttt{Mailbox}_{K(1)}$, $\texttt{Display}_{K(1)}$, and $\texttt{Act}_{K(1)}$ be the constant functions with domain $\texttt{PriProclet}_X$ and with values $\varnothing$, `undef`, and $\varnothing$, respectively. This is clearly a plausible ken, with $\texttt{Proclet}_{K(1)} = \texttt{PriProclet}_X$.

For the induction step, suppose $K(l)$ is already defined. Let each proclet $p \in \texttt{Proclet}_{K(l)}$ of level $\leq l$ fire in $K(l)$. Temporarily define $\texttt{Display}_{K(l+1)}(p)$ to be the resulting value of `myDisplay` for each such p and to be `undef` for all other elements p of the state. Similarly, temporarily define $\texttt{Act}_{K(l+1)}(p)$ to be the resulting value of `myAct` for each such p where this value is a set, and to be $\varnothing$ for all other elements p of the state. (We have defined these functions on too large a domain and will correct for this later; this was the only reason for saying "temporarily".) Use this temporary $\texttt{Act}_{K(l+1)}$ to define $\texttt{Proclet}_{K(l+1)}$, as in the definition of ken. Then restrict $\texttt{Display}_{K(l+1)}$ and $\texttt{Act}_{K(l+1)}$ to $\texttt{Proclet}_{K(l+1)}$ as required in the definition of ken. Note that, by restricting $\texttt{Act}_{K(l+1)}$ we have not changed what $\texttt{Proclet}_{K(l+1)}$ should be, since the characterization of `Proclet` in terms of `Act` (in the last three clauses in the definition of ken) uses `Act` only applied to elements of `Proclet`.

To complete the induction step, we must define $\texttt{Mailbox}_{K(l+1)}(p)$ for all elements $p \in \texttt{Proclet}_{K(l+1)}$. Define it to be the multiset of messages sent to p by the elements q of $\texttt{Proclet}_{K(l)}$ firing as in the preceding paragraph, multiplicities being summed over q. This completes the definition of $K(l)$.

In the inductive step, if $q \in \texttt{Act}_{K(l+1)}(p)$, with p and therefore also q in $\texttt{Proclet}_{K(l+1)}$, then, by definition of $\texttt{Act}_{K(l+1)}$, the ken $K(l)$ witnesses that the information flow digraph has an edge from p to $\langle q, p\rangle$. Thus, $K(l+1)$ is plausible. Since $K(1)$ is vacuously plausible, all $\texttt{Act}_{K(1)}(p)$ being empty, we conclude that $K(l)$ is plausible for all l.

We shall show that the sequence of kens $K(l)$ gradually stabilizes in the following sense. For each l, the kens $K(l)$ and $K(l+1)$ agree as to `Proclet` and `Mailbox` up to and including level l of the information

flow digraph, and they agree as to `Display` and `Act` at all levels strictly below l. The proof is by induction on l.

For the basis of the induction, $l = 1$, notice that the assertions of agreement concerning `Display` and `Act` are vacuous. Concerning `Proclet`, the definition says that it contains only elements of `PriProclet` when the ken is $K(1)$, but when the ken is $K(2)$ it can contain also secondary $\langle r, p\rangle$, where $r \in \mathtt{Act}_{K(2)}(p)$. As $K(2)$ is plausible, such secondary proclets cannot be at level 1. Thus, $K(1)$ and $K(2)$ have the same proclets at level 1. These proclets have, by definition, empty mailboxes under $K(1)$. Under $K(2)$, their mailboxes contain the messages sent to them by proclets q firing in $K(1)$. Any such message would cause an edge from q to p in the information flow digraph, which is impossible since p is at level 1. This completes the verification of the claimed stabilization between $K(1)$ and $K(2)$ and thus the basis of our induction.

For the induction step, suppose we have the desired stabilization between $K(l)$ and $K(l+1)$. To prove the stabilization between $K(l+1)$ and $K(l+2)$, compare the definitions of these two kens. The former fires the proclets of $K(l)$ at levels $\leq l$ in their initial local states given by $K(l)$; the latter fires the proclets of $K(l+1)$ at levels $\leq l+1$ in their initial local states given by $K(l+1)$. By induction hypothesis, these two sets of firings involve the same proclets p at levels $\leq l$ (though there may be different proclets at higher levels). Furthermore, for each such proclet p, its two computations yield the same updates and outputs, by Lemma 20 since the kens are plausible. This means, in particular, that, in the definitions of $K(l+1)$ and $K(l+2)$, the temporary versions of `Display` and `Act` agree up to and including level l.

We claim next that that `Proclet` is the same, for $K(l+1)$ and $K(l+2)$, up to and including level $l+1$. Suppose, toward a contradiction, that some q of level $\leq l+1$ is a proclet in one of the kens $K(l+1)$ and $K(l+2)$ but not in the other. Consider such a q of minimum possible level in the information flow digraph. Clearly, q cannot be in $\mathtt{PriProclet}_X$, for then it would be a proclet of both kens. So it is of the form $q = \langle r, p\rangle$ where, for one of the kens p is a proclet and $r \in \mathtt{Act}(p)$, while for the other ken either p is not a proclet or $r \notin \mathtt{Act}(p)$. As the kens are plausible, p is of lower level than q, so, by our choice of q to minimize the level, p is a proclet of both kens. So we must have $r \in \mathtt{Act}(p)$ for one ken and not for the other. We already saw that the `Act` functions of these two kens agree up to and including level l, so p must have level $\geq l+1$. That contradicts the fact that p has lower level than q. This contradiction completes the proof that the kens $K(l+1)$ and $K(l+2)$ have the same proclets up to and including level $l+1$.

Finally, for these common proclets at levels $\leq l+1$, the two kens $K(l+1)$ and $K(l+2)$ have the same mailboxes, since these mailboxes come from the outputs of the computations by proclets of levels $\leq l$, in initial local states given by $K(l)$ and $K(l+1)$, and we have seen that these outputs are the same in both cases.

This completes the inductive proof of the claimed agreement between $K(l)$ and $K(l+1)$ up to level l. Apply this result with l greater than the maximum length B of walks allowed by the Bounded Sequentiality Postulate. For such an l, we have $K(l) = K(l+1)$; the whole kens are identical. Rereading the definition of $K(l+1)$ in the light of this equality, we find that it says precisely that $K(l)$ is correct. □

Now that we have the existence and uniqueness of the correct ken for any state, we can formulate the **Update Postulate**: The update set of the algorithm in a (global) state is the set of all the updates of global dynamic functions produced by all the proclets of the correct ken, firing in the correct ken.

This is exactly like the corresponding postulate in [1] except that the notion of proclet now depends explicitly on the ken.

AB added another paragraph to the next remark. Note that the last sentence in that paragraph ("We could ...") is somewhat sloppy — nothing was said about what the ken should be for firings after the first, and the details of this could affect the reduction from a bounded number of firings to just one. I wouldn't mind omitting that last sentence.

Remark 23. The fact that, in each step of an algorithm, each proclet fires just once is formally contained in the Update Postulate. This postulate refers to a single firing, and so do the definitions on which it depends, like the definition of correct kens. And this postulate describes the whole influence of the proclets on the overall computation, since only the updated state persists to the next step.

The intuitive notion of bounded sequentiality requires bounds not only on the number of proclets that act in sequence, as formalized in the Bounded Sequentiality Postulate, but also on the sequentiality in the actions of a single proclet. The latter bound is ensured, in our postulates, by the requirements that proclets execute a sequential algorithm and that they fire only once per step. We could, without violating the idea of bounded sequentiality, allow proclets to fire a bounded number of times per step, but this would complicate our work without really increasing the generality, since a bounded number of steps could be coded into one. □

Remark 24. The Update Postulate implies that the updates of global dynamic functions produced by the various proclets do not clash. This is because the update set of an algorithm, as defined in connection with the Sequential Time Postulate in [1], will never contain conflicting updates. □

The next definition is like that in [1] but using our modified postulates.

Definition 25. A *parallel algorithm* is an algorithm satisfying the Sequential Time, Abstract State, Background, Proclet Algorithm, Bounded Sequentiality, and Update Postulates.

4. Abstract State Machines As Algorithms

This section is devoted to showing that ASMs can be viewed as parallel algorithms in the sense defined above. That is, we prove what was claimed in Section 8.5 of [1]. We do not repeat all the parts of Section 8.5 that are correct but concentrate on correcting the errors.

It will be useful first to make a few comments about the nature of the examples in Section 8 of [1]. Those examples involved various approaches to parallel computation — PRAMs, circuits, alternating Turing machines, first-order logic, fixed-point logic, and ASMs — and indicated how the algorithms of these models fit our postulates. The most important work here was to analyze these algorithms down to the level of proclets in order to say what the proclets and the proclet algorithm should be. Another aspect was adjoining, if necessary, the multisets, ordered pairs, etc., required by the Background Postulate, but this second aspect was fairly routine. Except for `Proclet`, everything required by the Background Postulate is determined once the "atomic" elements of the state are known. Thus, to make these other models of parallel computation fit our postulates, the procedure is roughly as follows. First analyze the algorithms to see what entities are directly involved. The small sequential processes that make up the parallel algorithm are among these entities, as proclets, and the way they work is the proclet algorithm. Then, if Boolean values, multisets, and ordered pairs are not already present, adjoin them, along with the basic operations on them. (See [1, Section 7], particularly Remarks 7.3 and 7.5, for comments on the naturality of adjoining these things.)

In the particular case of ASMs, the definition in [1, Section 9] required their vocabularies to contain everything listed in the Background Postulate. The ASMs constructed in the proof of the main result, Theorem 10.1, of [1] will indeed contain all these things, because they are

behaviorally equivalent to algorithms, as defined by the postulates; behavioral equivalence demands that the states and therefore the vocabulary are the same. But one can also consider other ASMs, for example the parallel ASMs of [3] (but without interactive features like external functions and importing reserve elements), and they should also fit our postulates when equipped with suitable proclets and a background containing multisets and ordered pairs.

For simplicity, we consider in detail only one version of ASMs, namely that defined in Section 9 of [1], with one clarification. Where we said that the vocabulary of an ASM should contain "all the items required by the Background Postulate", the last item in that postulate, "a variable-free term `Proclet` ..." (which would now become `PriProclet`) is to be omitted. If an ASM's voicabulary happens to contain a nullary function symbol `PriProclet`, then that symbol should be renamed to avoid a conflict with the `PriProclet` involved in our postulates. Note that a symbol `PriProclet` in some arbitrary ASM need not have anything to do with the actual proclets, the sequential subprocesses of which the algorithm is composed; the name `PriProclet` could be purely accidental. We prefer to reserve this name for the actual primary proclets.

Although we concentrate on one version of ASMs, the same ideas could be used to handle other versions, for example without multisets or without ordered pairs in the background. (These things, if missing from the ASM states, would, of course, be added when we define the states to be used in the postulates.) We believe that the version we consider exhibits all the difficulties of the natural ones, so that, given the following treatment of it, the reader will be able to treat the others also. The vocabulary of the algorithm we describe will consist of the vocabulary of the given ASM plus two new, static, unary function symbols defined as follows.

$$\texttt{Mult}(x, y) = \text{multiplicity of element } x \text{ in multiset } y$$

and

$$\texttt{pred}(n) = \{0, 1, \ldots, n-1\}$$

for natural numbers n. (`Mult` amd `pred` abbreviate "multiplicity" and "predecessors". What exactly the natural numbers are, in a state of our algorithm, is irrelevant to the functioning of the algorithm; we postpone discussion of the issue to Remark 26 after the presentation of the algorithm.)

The exposition in [1, Section 8.5] began with a rough but intuitively understandable description of the algorithm, containing three explicitly acknowledged difficulties, and it continued with a discussion of how to

circumvent these difficulties. In an attempt to retain intuitive understandability for our present, somewhat more complicated (but correct) algorithm, we again want to separate the main idea from the circumvention of the old difficulties. In this way, we can concentrate attention on the new aspects of the construction. Furthermore, of the three difficulties mentioned and circumvented in [1], two can be treated here in the same way as there, and one disappears entirely. We explain this first, in order to get all of these difficulties out of the way. (In [1], we described the algorithm first and eliminated the difficulties afterward, but in hte present context it seems clearer to handle the difficulties first.)

The first difficulty was that the rough description assumed that the ASM never produced conflicting updates. It was solved by showing, in [1, Section 9.2], how to convert any ASM into one that never produces such clashes. The same solution applies in the present context.

The second difficulty was that there are infinitely many proclets, though only finitely many become active in any step. This difficulty disappears in the present context. Our postulates require only the set of primary proclets to be finite, and the algorithm we describe will involve only a single primary proclet. Any element of the state is potentially a secondary proclet, and by activating some of these we can obtain all the proclets used by the construction in [1]. Thus, we no longer need the terms $\mathtt{MDA}(p)$ and $\mathtt{MA}(p)$ that we defined — incorrectly because we used comprehension terms — on pages 624–625 of [1].

The third difficulty concerned cycles in the information flow digraph, where one proclet activated another and gave it some information, and then the second proclet returned some information to the first. This difficulty was solved by replacing each proclet by two or three others, each performing a part of the original proclet's task. We shall refer to these two or three proclets as *incarnations* of the original one. Thus, the first incarnation of one proclet might activate another proclet, but the reply from the second proclet would then go to the second or third incarnation of the first. (More precisely, the first incarnation of the first proclet might activate the first incarnation of the second proclet, and eventually the last incarnation of the second proclet would reply to the second or third incarnation of the first proclet.) This solution continues to work in the present setting. "Activate" has, of course, a new meaning, the meaning given by our postulates, rather than merely displaying some information that is read by the proclet to be activated. But the idea remains the same. In fact, we shall describe our algorithm in the same format as in [1], numbering a proclet's tasks in a way that indicates which tasks are to be done by which incarnations.

In addition to getting these difficulties out of the way, we must make another preliminary comment, on the nature of the proclets used in our algorithm. In the rough description (before addressing the three difficulties) in [1, pages 622–623], the proclets were ordered pairs $\langle \hat{Z}, \bar{a} \rangle$, where Z is an occurrence of a term or rule in the given ASM, $\hat{Z}$ is a "name" for it, and $\bar{a}$ is a tuple of values for its pseudo-free variables. Recall that the names were assigned rather arbitrarily in [1]; they just need to be elements of the state that can be explicitly named, and this is easy to arrange since there are only finitely many of them. Recall also that a variable x is *pseudo-free* in an occurrence Z if Z lies in the scope of a comprehension term or a parallel rule that binds x; since an ASM program has no free variables, all the free variables of a term or rule are among its pseudo-free variables. To list the values of these variables in a tuple, we implicitly assume a particular ordering of the variables. It will be convenient to assume that the pseudo-free variables of any term or rule Z are listed in decreasing order of their scopes; these scopes are linearly ordered because they all contain Z. (This convention was already tacitly used in [1].)

Our proclets will be more complicated for two reasons. First, as indicated above in the solution of the third difficulty, each of the proclets in the rough description will actually have two or three incarnations, different proclets that divide the work in such a way as to avoid cycles. Some care will be needed in the choice of just which elements of the state serve as the second and third incarnations of proclets; we postpone the details until after we have presented enough of the algorithm to motivate the details. For now, the reader can use the following simple picture of incarnations. Because the necessary number of incarnations of a proclet $\langle \hat{Z}, \bar{a} \rangle$ is entirely determined by the nature of the term or rule Z, we can simply include, with $\hat{Z}$, a marker distinguishing the various incarnations of the proclet. Thus, instead of $\langle \hat{Z}, \bar{a} \rangle$, we would have $\langle \widehat{Z, k}, \bar{a} \rangle$ for the k^{th} incarnation of $\langle \hat{Z}, \bar{a} \rangle$.

The second complication in our proclets arises from the convention that each secondary proclet p is an ordered pair whose second component is the proclet that activated p. Thus, where one might intuitively think of a chain of activations, say

$$x \in \texttt{PriProclet}, \quad y \in \texttt{Act}(x), \quad z \in \texttt{Act}(y), \quad w \in \texttt{Act}(z),$$

the actual chain would look like

$$x \in \texttt{PriProclet}, \quad y \in \texttt{Act}(x), \quad z \in \texttt{Act}(\langle y, x \rangle), \quad w \in \texttt{Act}(\langle z, \langle y, x \rangle \rangle),$$

the last proclet activated here being $\langle w, \langle z, \langle y, x\rangle\rangle\rangle$. In our situation, this means that our proclets will not have the simple form $\langle\widehat{Z, k}, \bar{a}\rangle$ but will be ordered pairs whose first components have this form and whose second components are themselves proclets. The proclet previously called $\langle\widehat{Z, k}, \bar{a}\rangle$ will thus encode the list of all the occurrences of terms and rules within which Z lies.

Both of these complications in the proclets — exhibiting the incarnation numbers and the ancestors of the proclets — will contribute little to the essential ideas of the algorithm that we shall describe, but they threaten to obscure the ideas by cluttering the notation. Accordingly, we adopt the convention of writing simply $\langle\hat{Z}, \bar{a}, \dots\rangle$; the intention is that the "..." reminds us of all the extra information coded in the proclet, but we refrain from exhibiting this information and thus remain close to the $\langle\hat{Z}, \bar{a}\rangle$ notation used in [1].

In addition to the proclets $\langle\hat{Z}, \bar{a}, \dots\rangle$ that will play essentially the same roles as $\langle\hat{Z}, \bar{a}\rangle$ in the rough description in [1], our algorithm will involve some additional proclets. Most of these arise from the need to avoid comprehension terms in the proclet algorithm. In describing the activity of proclets $\langle\hat{u}, \bar{a}\rangle$ where u is a comprehension term, we incorrectly used comprehension terms in [1]. Now, the "work" done by these comprehension terms will be described honestly, using additional proclets.

There will also be some additional proclets in our description of the activity of proclets $\langle\hat{R}, \bar{a}, \dots\rangle$ when R is a parallel rule `do forall` $x \in r$, $R_0(x)$ `enddo`. These are needed in order to get the right format for the subsidiary proclets $\langle\widehat{R_0(x)}, \bar{a}^\frown c, \dots\rangle$.

The work of all these additional proclets will be described in the context of the work of their activators (or activators of activators). The "main" proclets $\langle\hat{Z}, \bar{a}, \dots\rangle$ serve the same purpose as $\langle\hat{Z}, \bar{a}\rangle$ did in [1]. When Z is a term, the purpose is to compute its value v, using $\bar{a}$ for the values of (pseudo-)free variables, and to push v to the parent (i.e., to the activator). More precisely, it pushes the pair $\langle v, \langle\hat{Z}, \bar{a}, \dots\rangle\rangle$, so the parent knows which child computed this v. When Z is a rule, the purpose is to ensure the execution of the updates that the rule produces, again using $\bar{a}$ to supply the values of free variables. To "ensure the execution" here means either to execute the updates or to activate enough other proclets that will ensure the execution.

In the following description of our algorithm, it is to be understood that, when a proclet (of the rough description) has several incarnations (in the precise description), then each of these incarnations except the

last is to activate the next one. This activation is to be added to the activations explicitly mentioned in the following description.

In the light of the preceding discussion, we can use, in our present algorithm, much of what was done in the rough description in [1]. Specifically, the activity of the proclets corresponding to variables, to terms of the form $f(t_1, \ldots, t_n)$, to update rules, and to conditional rules can be described exactly as in [1, pages 622–623] with just two modifications:

- Change every $\langle \hat{Z}, \bar{a} \rangle$ to $\langle \hat{Z}, \bar{a}, \ldots \rangle$.
- Delete the parenthetical comment that activation is done "by displaying an appropriate signal", since activation is now done by updating `myAct`.

Notice that, in each of these cases, a proclet activates a bounded set of other proclets, so the desired `myAct` can be explicitly given by the proclet algorithm.

There remain the proclets corresponding to comprehension terms and to parallel rules — the two ASM constructs that introduce unbounded parallelism. For these proclets, we cannot proceed exactly as in [1]. Indeed, the instructions in [1] for the second incarnation of such a proclet (i.e., the instructions labeled (2) in the rough description) involve activating an unbounded set of secondary proclets. To do so, a proclet would update `myAct` to mark a set for activation, but the necessary set could be described only by a comprehension term. The following instructions for these two sorts of proclets avoid this difficulty by marking for activation only a set directly available to the activating proclet, i.e., a set that can be named in the proclet's initial local state. The price for this is that the comprehension term to be avoided involved some parallel work, which must now be done by additional proclets. Here are the details.

Let p be the proclet $\langle \hat{u}, \bar{a}, \ldots \rangle$ where u is a comprehension term $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$. As in [1], the work of p will be in three parts (i.e., p will have three incarnations).

(1) Activate $\langle \hat{r}, \bar{a}, \ldots \rangle$.
(2) After receiving the value b computed by this secondary proclet, display b and mark $\mathrm{AsSet}(b)$ for activation.
(3) Push $\langle \texttt{myMail}, \texttt{me} \rangle$ to your parent.

Obviously, for this to be correct, a lot has to happen between instructions (2) and (3). Specifically, the proclets activated in (2) must somehow ensure that the mailbox of p is exactly the multiset that p is supposed to compute, the value of $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ when the free variables have values given by $\bar{a}$. That is achieved as follows.

The proclets activated as a result of (2) are $\langle c, p, \dots \rangle$ for all $c \in b$. (The "..." here refer only to an indication of the fact that the activator is the second incarnation of p.) Each such $\langle c, p, \dots \rangle$ does the following.

(1) Read b from the display of (the second incarnation of) p and mark for activation
$$\{\!\{\langle\widehat{t(x)}, \bar{a}^\frown c\rangle, \langle\widehat{\varphi(x)}, \bar{a}^\frown c\rangle\}\!\} \uplus \mathtt{pred}(\mathtt{Mult}(c, b)).$$
Recall that $\mathtt{pred}(\mathtt{Mult}(c, b)) = \{0, 1, \dots, m-1\}$ where m is the multiplicity of c as an element of b.

(2) When the proclets $\langle\widehat{t(x)}, \bar{a}^\frown c\rangle$ and $\langle\widehat{\varphi(x)}, \bar{a}^\frown c\rangle$ return values, if $\varphi(c) = \mathtt{true}$ then display $\{t(c)\}$; otherwise display $\varnothing$.

The additional proclets $\langle k, \dots \rangle$ (for $0 \le k < \mathtt{Mult}(c, b)$) activated here read the display of (the second incarnation of) $\langle c, p, \dots \rangle$ and, if it is nonempty, extract its element (using TheUnique) and mail that to (the third incarnation of) p. As a result of all this work, the proclets activated by $\langle c, p, \dots \rangle$ will contribute to the mailbox of p either nothing, if $\varphi(c) = \mathtt{false}$, or exactly $\mathtt{Mult}(c, b)$ copies of $t(c)$, if $\varphi(c) = \mathtt{true}$. Combining the results from all elements c of AsSet(b), we obtain as the mailbox of p exactly the required multiset, the value of $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ with free variables interpreted according to $\bar{a}$.

Finally, we consider the somewhat easier case of a parallel rule. Let p be the proclet $\langle \hat{R}, \bar{a}, \dots \rangle$, where R is the rule `do forall` $x \in r, R_0(x)$ `enddo`. Its instructions are

(1) Activate $\langle \hat{r}, \bar{a}, \dots \rangle$.

(2) After receiving the value b computed by this secondary proclet mark AsSet(b) for activation.

Each proclet $\langle c, p, \dots \rangle$ activated in (2) merely activates $\langle\widehat{R_0(x)}, \bar{a}^\frown c, \dots \rangle$. These proclets will then ensure the execution of $R_0(c)$ for all $c \in b$, as required.

To complete the description of our algorithm, we still have two tasks to accomplish. We must provide the term `PriProclet`, and we must keep our promise to provide details about which elements of the state serve as the second and third incarnations of our proclets.

The first of these tasks is remarkably easy. We need only a single primary proclet, the one corresponding to the whole ASM program Π, considered as a rule; all other proclets that we need are activated during the execution of the algorithm. We therefore define `PriProclet` to be $\{\!\{\langle \hat{\Pi}, \langle\rangle \rangle\}\!\}$. (Here $\langle\rangle$ is the empty tuple, since Π has no pseudo-free variables.)

Finally, we turn to incarnations. When a proclet p activates its next incarnation p', it does so by including an appropriate element q

in its `myAct`, so that $p' = \langle q, p\rangle$. But what is an appropriate q? The proclet $\langle q, p\rangle$ had better be distinct from all the other proclets activated by p. We need not worry about coincidences with other secondary proclets, activated by proclets other than p, for these will have their activators, not p, as their second components. We also need not worry about a coincidence with the primary proclet $\langle \hat{\Pi}, \langle\rangle\rangle$, since its second component $\langle\rangle$ is distinct from p. In most cases, our description of the algorithm tells what the other proclets activated by p are, and it is easy to find a suitable q. If, as suggested in [1], we use certain multisets as the codes $\hat{Z}$, then `true` will work as the required q in all but two cases. To see this, just use the fact that `true` is not a multiset or an ordered pair or a natural number. (See Remark 26 below about natural numbers.) The two cases where this choice might fail are those where p is the second incarnation of $\langle \hat{Z}, \bar{a}, \dots\rangle$ and Z is a comprehension term or a parallel rule. In those cases, the other proclets activated by p include $\langle c, p\rangle$ for all members c of b (in the notation used for describing the algorithm above). So we must ensure that $q \notin b$. Fortunately, there is a standard (in set theory) trick for getting an object that is not a member of a given (multi)set b, namely to take the object b itself. (There are also other options, for example $\{\!\{b\}\!\}$.) Thus, we can obtain the next incarnation, in the two cases where $q =$ `true` might not work, by taking $q = b$ instead. One final comment is needed here, namely that, in these cases, p should display b, so that the proclets it creates can tell, by reading the display, whether they are the next incarnation of p or one of the other proclets, $\langle c, p\rangle$ for $c \in b$, that p activated.

Remark 26. Our algorithm depended on the availability of natural numbers and the functions `Mult` and `pred`. There are at least two plausible ways to represent natural numbers by elements of the states of our algorithms. One is the coding, due to von Neumann, that has become standard in set theory. It sets (inductively) $n = \{0, 1, \dots, n-1\}$. With this coding `pred` is simply the identity function. An alternative coding, quite natural when multisets are available, is to represent n by a multiset consisting of n copies of some standard entity, for example `true` or $\varnothing$.

But in fact, we could do without any particular representation of the natural numbers. The algorithm never used `Mult` and `pred` individually but only in the combination `pred` $\circ$ `Mult`, and it never mattered that the elements of `pred` $\circ$ `Mult`(c, b) were numbers, only that they were `Mult`(c, b) distinct objects, all distinct from `true`. (Distinctness from `true` was used only in our discussion of using $q =$ `true` for producing

second and third incarnations.) Thus, we could simply assume the existence of some such function to serve in place of `pred` ∘ `Mult`. □

5. Algorithms Are Abstract State Machines

Our goal in this section is to show that the ASM thesis, that all algorithms are behaviorally equivalent to ASMs, holds for parallel algorithms in the sense of Definition 25. That is, the thesis is not damaged by our extension of the notion of parallel algorithm from [1] to allow intra-step creation of proclets. We do not change the definitions of ASMs [1, Section 9.1] and of behavioral equivalence [1, Definition 2.3]. Recall that behavioral equivalence is a very strong equivalence relation, requiring the same states, the same initial states, and the same one-step transition function. In particular, when we construct an ASM equivalent to a given algorithm A, it is obvious what the states and initial states of the ASM must be; the only issue is constructing an ASM program that produces the same transition function as A.

Theorem 27. *Every parallel algorithm is behaviorally equivalent to an ASM.*

Proof. The proof is very similar to the proof of the corresponding, weaker result, Theorem 10.1, in [1], so we only explain the changes that are required by our present, broader notion of algorithm.

The first part of the proof in [1], expressing the proclet algorithm as a sequential ASM with output, Π, requires only a notational change: Since the initial local state now interprets the dynamic, nullary symbol `myAct`, this symbol is added to the vocabulary of Π.

The next part of the proof in [1], starting at the bottom of page 631, gives a rough description of how the ASM's computation will proceed. As pointed out there, the most natural approach to computing the correct ken, namely imitating the level-by-level recursion used in the proof of Theorem 7.22, needs to be modified because information about the levels is not available to the proclets. So in [1] the rough description of the ASM's work is called a "rough description of the modification" of the natural approach.

We must now make a further modification because the notion of proclet is not fixed by the state (as it was in [1]) but changes from phase to phase as a result of activations. The work of the ASM still proceeds in B phases, with all proclets executing the proclet algorithm in certain initial local states at each phase, but the set of proclets here depends on the phase. In the first phase, the proclets are the elements of `PriProclet`. At any later phase, say phase k, the proclets are the elements of `PriProclet` and the elements activated by proclets

at phase $k-1$. The latter are, of course, the elements of the form $\langle q, p\rangle$ where p was a proclet at phase $k-1$ and it updated its `myAct` to a set containing q.

At each phase, the proclets of that phase execute the proclet algorithm in an initial local state where `myMail` and `Display` are interpreted as the results of the preceding phase (with $\texttt{myMail} = \varnothing$ and $\texttt{Display}(q) = \texttt{undef}$ in phase 1), exactly as in [1]. The argument in [1, page 632] showing that, in phase k, all proclets of level $\leq k$ are computing in the initial local states given by the correct ken carries over to the present context. It follows that these proclets do the correct pushing, displaying, and activating at this phase and that, as a result, the next phase has the correct set of proclets up to and including level $k+1$. (The reference to Lemma 7.25 in [1] is now replaced with a reference to Lemma 20.)

Just as in [1], the rough description must be supplemented with a decision to suppress all updates of the global state until phase B, during which the ASM is using the correct set of proclets with the correct initial local states at all levels.

In the formal presentation of the ASM, starting on page 633 of [1], the following extensions are needed to handle activation of proclets. First, in addition to the terms (or more precisely term schemas) $\texttt{Outmail}(p, M, D)$ and $\texttt{Dspl}(p, M, D)$ used there, we also have $\texttt{Asp}(p, M, D)$. Here `Asp` stands for "activate secondary proclets"; the intended meaning is that proclet p, with M as its mailbox and D as the display function, would execute updates of `myAct`, and $\texttt{Asp}(p, M, D)$ is the multiset of all the elements x such that p would update `myAct` to x, just as in [1] $\texttt{Dspl}(p, M, D)$ is the multiset of those x such that p updates `myDisplay` to x. (The ASM program Π can be arranged so that it produces at most one update of `myAct` and `myDisplay`, but it is convenient to give the definitions in a general form that does not presuppose this uniqueness.)

The definitions of $\texttt{Outmail}_R$ and $\texttt{Dspl}_R$ by induction on rules R, as given in [1], must be supplemented with clauses to define $\texttt{Asp}_R$. To formulate these clauses, we use for terms t the notation t', defined just as in [1] with the extra clause that `myAct` in t is to be replaced with $\varnothing$ in t'. Now the clauses defining $\texttt{Asp}_R$ are exactly analogous to the clauses for $\texttt{Dspl}_R$ in [1]:

- If R is an update rule of the form $\texttt{myAct} := t$, then $\texttt{Asp}_R(p, M, D)$ is $\{\!\{t'\}\!\}$.
- If R is any other update rule, then $\texttt{Asp}_R(p, M, D)$ is $\varnothing$.
- If R is `Push` t_0 `to` t_1, then $\texttt{Asp}_R(p, M, D)$ is $\varnothing$.

- If R is `do in parallel` $R_0, \ldots, R_k$ `enddo`, then $\mathtt{Asp}_R(p, M, D)$ is the sum $\mathtt{Asp}_{R_0}(p, M, D) \uplus \cdots \uplus \mathtt{Asp}_{R_k}(p, M, D)$.
- If R is `if` φ `then` R_0 `else` R_1 `endif`, then $\mathtt{Asp}_R(p, M, D)$ is

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

Continuing in analogy with how we handled `Dspl`, we define $\mathtt{Asp}(p, M, D)$ to be TheUnique(AsSet($\mathtt{Asp}_\Pi(p, M, D)$)). One more definition is needed, to incorporate both the automatic tagging of secondary proclets with their activators and the convention that, if a proclet marks for activation something other than a set, then it thereby activates nothing. Accordingly, we define $\mathtt{TP}(p, M, D)$ to be

$$\{\!\{\langle q, p\rangle : q \in \mathtt{Asp}(p, M, D) : \mathrm{AsSet}(\mathtt{Asp}(p, M, D)) = \mathtt{Asp}(p, M, D)\}\!\};$$

the notation `TP` stands for "tagged proclets".

Next, we modify the definitions in [1, page 634] of $\mathtt{Mailbox}_k(p)$ and $\mathtt{Display}_k(p)$ to take into account the possible variation of the set of proclets from one phase to another. The (unique) occurrence of `Proclet` in these definitions is to be replaced with $\mathtt{Proclet}_k$, which in turn is defined by induction on k simultaneously with $\mathtt{Mailbox}_k(p)$ and $\mathtt{Display}_k(p)$, as follows.

- $\mathtt{Proclet}_0$ is `PriProclet`.
- $\mathtt{Proclet}_{k+1}$ is $\mathtt{PriProclet} \uplus \biguplus\{\!\{\mathtt{TP}(p, \mathtt{Mailbox}_k(p), \mathtt{Display}_k) : p \in \mathtt{Proclet}_k : \mathtt{true}\}\!\}$.

It was shown in [1] that $\mathtt{Mailbox}_k(p)$ and $\mathtt{Display}_k(p)$ give the mailbox and display functions after k phases of the computation in our description of how the desired ASM works. The argument there extends to show that $\mathtt{TP}_k(p)$ gives the set of proclets activated by p in phase k and therefore that $\mathtt{Proclet}_k$ is the set of proclets that execute during phase $k + 1$. Arguing as on page 635 of [1], we find that the updates of the given algorithm A are matched by the ASM program

$$\mathtt{do\ forall}\ p \in \mathtt{Proclet}_{B-1}\ \Pi^*(p)\ \mathtt{enddo}$$

(the same as in [1] except for the subscript $B - 1$ on `Proclet`), where $\Pi^*(p)$ is obtained from Π by the same substitutions as in [1], except that `Skip` replaces not only updates of `myDisplay` but also updates of `myAct`. □

Acknowledgment. We thank Dean Rosenzweig for pointing out that we had incorrectly used comprehension terms in [1, Section 8] and for subsequent helpful discussions.

References

[1] Andreas Blass and Yuri Gurevich, "Abstract state machines capture parallel algorithms," *A. C. M. Trans. Computational Logic* 4 (2003) 578–651.

[2] Andreas Blass, Yuri Gurevich, "Ordinary interactive small-step algorithms, I," *A. C. M. Trans. Computational Logic* 7 (2006).

[3] Yuri Gurevich, "Evolving algebra 1993: Lipari guide," in *Specification and Validation Methods*, E. Börger, ed., Oxford Univ. Press (1995) 9–36.

[4] Yuri Gurevich, "Sequential abstract state machines capture sequential algorithms," *A. C. M. Trans. Computational Logic* 1 (2000) 77–111.

Mathematics Department, University of Michigan, Ann Arbor, MI 48109–1043, U.S.A.
E-mail address: ablass@umich.edu

Microsoft Research, One Microsoft Way, Redmond, WA 98052, U.S.A.
E-mail address: gurevich@microsoft.com