Persistent Queries in the Behavioral Theory of Algorithms

ANDREAS BLASS
University of Michigan
and
YURI GUREVICH
Microsoft Research

We propose an extension of the behavioral theory of interactive sequential algorithms to deal with the following situation. A query is issued during a certain step, but the step ends before any reply is received. Later, a reply arrives, and later yet the algorithm makes use of this reply. By a persistent query, we mean a query for which a late reply might be used. Our proposal involves issuing, along with a persistent query, a location where a late reply is to be stored. After presenting our proposal in general terms, we discuss the modifications that it requires in the existing axiomatics of interactive sequential algorithms and in the existing syntax and semantics of abstract state machines. (To make that discussion self-contained, we include a summary of this material before the modifications.) Fortunately, only rather minor modifications are needed.

Categories and Subject Descriptors: F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Bounded-action devices*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification Techniques*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*; I.6.1 [**Computing Methodologies**]: Simulation and Modeling—*Simulation Theory*

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words and Phrases: abstract state machines, interactive algorithms, sequential algorithms, small-step algorithms, step-for-step simulation

1. INTRODUCTION

We adopt the name *behavioral theory of algorithms* for the project, initiated in [Gurevich 2000] and extended in [Blass and Gurevich 2003; 2006-7; Blass et al. 2007], seeking to accomplish two goals for various natural classes of algorithms:

Axiomatization. Characterize the class in abstract terms by means of suitable postulates.

Representation. Prove that the algorithms in the class are exactly those express-

Authors' addresses: A. Blass, Mathematics Department, University of Michigan, Ann Arbor, MI 48109–1043, U.S.A.; Y. Gurevich, Microsoft Research, One Microsoft Way, Redmond, WA 98052, U.S.A.

Blass is partially supported by NSF grant DMS-0653696 and by a grant from Microsoft Research.

ible in a suitable, natural programming language.

The project has been carried out for small-step (sometimes also called sequential) algorithms in [Gurevich 2000]; for parallel algorithms in [Blass and Gurevich 2003]; for a limited but natural class of interactive, small-step algorithms, called *ordinary* ones, in [Blass and Gurevich 2006-7]; and for general interactive, small-step algorithms in [Blass et al. 2007]. The programming languages used for the second goal have been versions of abstract state machines (ASMs). Thus, the results concerning this goal have provided partial confirmation for the *ASM thesis*, the thesis that every algorithm can be represented, on its natural level of abstraction, by an ASM.

The present paper extends the behavioral theory of interactive, small-step algorithms, which was presented in [Blass et al. 2007].

Previous work on the behavioral theory of algorithms has concentrated on what happens during a single step. In particular, the papers cited above established, for various classes of algorithms, the representation theorem that every algorithm in the class can be matched, step for step, by an ASM. Almost nothing was said there about what happens between steps, because almost nothing can be said; the environment can make essentially arbitrary inter-step changes to the state.

Intra-step interaction with the environment, in contrast, was treated in great detail in [Blass and Gurevich 2006-7; Blass et al. 2007]. The key difference from inter-step interaction is that, although the environment can, during a step, give essentially arbitrary replies to the algorithm's queries, the effect of these replies on the state and thus on the future course of the computation is under the algorithm's control.

In the present paper, we use inter-step interaction to treat an issue arising out of intra-step interaction, namely the possibility of a query being answered after the completion of the step in which the query was issued. We modify slightly the axioms in [Blass et al. 2007] and we extend the ASM syntax of [Blass et al. 2007] to accommodate such late replies.

As in [Gurevich 2000; Blass and Gurevich 2006-7; Blass et al. 2007], we restrict attention to small-step algorithms. The amount of work that a small-step algorithm performs during any one step is bounded independently of the state or input. In the rest of this article, algorithms are by default small-step. In [Blass and Gurevich 2006-7, Part I], we argued that, in principle, the intra-step interaction of an algorithm with the environment reduces to the algorithm querying the environment and the environment answering these queries. In the case of ordinary algorithms [Blass and Gurevich 2006-7], every query issued during a step needs to be answered before the algorithm finishes the step. In the general case [Blass et al. 2007], however, the algorithm may finish a step without having all the replies.

If, in such a situation, the reply to a query arrives after the algorithm's step has ended, then the question arises how to handle the late reply. It may happen that the algorithm has no use for that late reply; consider for example an algorithm that issues two queries and sets x to 1 when at least one of the two replies arrives. So the step would end when one of the two queries has been answered; the other might be answered later. In such a case, the late reply can simply be ignored or discarded. But suppose that the algorithm eventually, at some later step, needs the late reply. In the framework of [Blass et al. 2007] where attention is restricted to

one step of an algorithm, the natural solution was this. If and when the algorithm needs a late reply, it issues an auxiliary query inquiring whether the reply is in. Another natural solution, in tune with current programming practice, is to fork out a separate computation thread that will wait for the late reply and will perhaps do some work with the late reply if and when it appears [Wikipedia 2008]. Here we propose a solution that does not require additional queries or additional computation threads.[1] Indeed, it stays very close to the notion of interactive small-step computation, without persistent queries, described in [Blass et al. 2007].

Accordingly, we base our discussion on the model of interactive computation introduced and analyzed in [Blass et al. 2007], which we review in Sections 6 and 8. This model differs from the earlier, more special model of [Blass and Gurevich 2006-7] in two ways, one of which is the possibility of completing a computation step without waiting for replies to all the queries issued during the step.[2] It is this possibility that opens the door to the topic of late replies and their subsequent use by the algorithm. The primary purpose of this paper is to describe a minor modification of the axioms of [Blass et al. 2007] and an ASM model that incorporate such *persistent queries* and their *late replies*.

The paper is organized, apart from this introduction, into three parts, each containing several sections. Part I gives an informal explanation of the problem in Section 2 and our proposed solution in Section 3. We conclude Part I with a discussion in Section 4 of possible extensions of this work and with a quick review in Section 5 of related work. Parts II and III present the solution in a formal framework, Part II for the axiomatic part of behavioral theory of algorithms and Part III for the ASM part. In Part II, Section 6 is a review of the axiomatic part of [Blass et al. 2007] and Section 7 describes the modifications of the axioms that we propose in order to deal with persistent queries and late replies. In Part III, we again begin, in Section 8, with a review of material from [Blass et al. 2007], namely the ASM syntax developed there to establish the representation theorem for interactive algorithms. In Section 9 we modify that syntax to include persistent queries. Finally, in Section 10, which may be regarded as a sort of appendix to the paper, we prove a theorem characterizing the parts of an ASM program that could produce a query whose answer might come only at a later step.

This paper is intended to be more pragmatic than in the theoretical work of [Gurevich 2000; Blass and Gurevich 2003; 2006-7; Blass et al. 2007]. In those papers, the primary goal was to axiomatize in complete generality certain natural classes of algorithms and to provide representation theorems for those classes. Considerable effort went into justifying, from first principles, the proposed axioms. The present paper was originally motivated by the need to handle persistent queries in the ASM-based specification language AsmL.[3] Our goal is to present what we believe is a good way to fill that need, but we do not claim that this way is demanded

[1] The reader may wonder how our solution relates to futures [Wikipedia 2008]. We discuss the issue in Part I.

[2] The other is that the algorithm can take into account the order in which replies are received during a step.

[3] We thank a referee for encouraging us to reorganize the paper in a way that separates the axiomatic aspects (Part II), applicable to algorithms in general, from the aspects concerned specifically with ASMs (Part III).

by first principles. There are alternative ways to deal with persistent queries; we shall describe a few of them and compare them with our proposal in Section 2. And there are certainly alternatives to the specific axioms and ASM syntax that we propose in Sections 7 and 9, although we believe that our choices are quite natural.

As indicated above, we limit ourselves here to small-step algorithms, i.e., algorithms that work in a sequence of discrete steps (as opposed to distributed algorithms where there may be no clear notion of (global) step because agents act asynchronously) and do only a bounded amount of work per step, with the bound depending only on the algorithm, not on the input or state (as opposed to, for example, massively parallel algorithms where the number of available processors may be increased according to the input size).

There are several justifications for this limitation. First, many of the algorithms used in practice are small-step.

Second, even in massively parallel or distributed algorithms, the individual processors or agents are usually small-step algorithms. Communication between agents is, from the point of view of any one agent, an interaction with its environment. An important motivation for developing a general model for interaction between an algorithm and its environment is this situation where the algorithm under consideration is one agent — a small-step algorithm — while the environment includes the other agents.

Third, small-step algorithms are the only class of algorithms for which the general theory of interactive algorithms has been worked out in detail and for which interactive ASMs have been proved to be adequate to capture all algorithms of the class [Blass and Gurevich 2006-7; Blass et al. 2007]. For parallel algorithms, the analogous work has been done only in the absence of intra-step interaction [Blass and Gurevich 2003], and the case of distributed algorithms remains entirely in the domain of future work. Thus, the foundation on which we shall build in the present paper is currently available only for small-step algorithms.

Finally, it is reasonable to expect that what we do here for small-step algorithms will suggest how to do the analogous tasks for broader classes of algorithms, once the necessary framework is in place. What we do here may also be useful in extending our work on parallel algorithms [Blass and Gurevich 2003] to include external interactions, because it allows greater flexibility in handling the flood (or trickle) of replies that a parallel algorithm might receive all at once.

PART I: AN IMPROVED INTERACTIVE SMALL-STEP COMPUTATION MODEL

In this part we explain on an informal level a new model of interactive small-step algorithms and the associated abstract state machines, allowing us to handle persistent queries. In the second and third parts of the paper we shall cover the formal semantics of the new model in full detail, dealing first with the axiomatics of this class of algorithms and then with the ASM model.

2. PERSISTENT QUERIES AND LATE REPLIES

As indicated earlier, we are concerned in this paper with providing an axiomatic foundation and an ASM formalism that conveniently handle the following situation:

An algorithm issues a query during a certain step, but finishes the step without getting an answer to that query. The answer arrives later and is then used in some subsequent step of the algorithm.

By a *late reply*, we mean a reply from the environment to a query q, reaching the algorithm after the completion of the step in which q was issued. If a late reply to q can influence the subsequent work of the algorithm, then we call q a *persistent query*.

Remark 2.1. The most natural meaning of "can influence" in the preceding sentence involves what can actually happen in runs of the algorithm. Like other run-time properties, persistence is then undecidable in general. That undecidability does no harm to our work in this paper. On the other hand, when writing programs, one is faced with the need to decide which queries should be considered persistent and treated by the methods of this paper. For this purpose, one should interpret "can influence" to mean that the programmer does not know with certainty that a late answer will never be used. It does no harm if a program treats a query as persistent even when, at run time, it turns out not to be persistent. □

Example 2.2. Consider, as an example, a broker who has a block of shares to sell and offers the entire block to two clients, as in Example 3.3 of [Blass et al. 2007, Part I]. As soon as he gets a positive reply from either client, he sells all the shares to that client. (The situation where positive replies from both clients reach the broker simultaneously is discussed in [Blass et al. 2007, Part I, Example 3.20], but it need not concern us here.) Suppose that the broker has sold the shares to client A and completed his step (with an update to his state, recording the sale) without having received any reply from client B. Later, he gets a reply from B, who also wants to buy the shares. He should then tell B, "Sorry, I already sold the shares to someone else, whose acceptance of my offer reached me before yours." Thus, the actions of the broker (regarded as an algorithm) take into account B's reply, even though the reply came after the completion of the step in which the associated query (the offer to sell the shares) was issued. So that query is persistent.

Example 2.3. A pollster sends questionnaires to many people. Being a small-step algorithm, the pollster sends the questionnaires a few at a time, so the sending occupies numerous steps. Later, the filled-in questionnaires arrive and the pollster processes them. Usually, a questionnaire will be filled in and returned only after the end of the (pollster's) step in which it was sent out. So the filled-in questionnaires are late replies, and the associated queries, the original, blank questionnaires, are persistent queries.

Even if one of the respondents is so quick that the pollster gets the reply in the same step in which he issued the query (so we are dealing with a traditional reply, as in [Blass and Gurevich 2006-7; Blass et al. 2007], not a late reply), the pollster will probably want to postpone processing this reply until after he finishes mailing all the questionnaires. More generally, an algorithm may well treat all replies the same, whether they are late or not.

How should persistent queries and late replies be treated in the axiomatic framework of [Blass et al. 2007]?

Let us first say a few words about the role of queries and their answers in interactive small-step algorithms; details are found in [Blass and Gurevich 2006-7] or [Blass et al. 2007] and reviewed in Section 6 below. The queries and replies from any single step of the computation form a *history* (in [Blass et al. 2007], an *answer function* in [Blass and Gurevich 2006-7]), which is empty at the beginning of a step and gradually grows as queries are issued and answered. The history influences the algorithm's actions (issuing additional queries, ending the step, updating the state) during that step, but it is reset to empty for the start of the next step. This is in accordance with the general principle — intuitively the definition of "state" — that the state must include all the information from past steps that can influence the future progress of the computation. In particular, if the reply to some query is to be used after the step has ended, then that reply must be recorded in the state; it will not survive in the history.

When we deal with the syntactic representation of algorithms by ASM's, we adopt, as in previous work [Blass and Gurevich 2006-7; Blass et al. 2007], the so-called *Lipari convention*, namely that if the same external function symbol occurs several times in an ASM program, and if its arguments at different occurrences happen to evaluate to the same elements in a particular state, then all those occurrences result in only a single query during any single step of the algorithm. For a discussion of alternative conventions and of our reasons for adopting the Lipari convention, see [Blass and Gurevich 2006-7, Part II, Section 4].

Remark 2.4. One of those alternative conventions, the *must-vary convention*, is commonly used in practice. This convention requires all occurrences of external functions in a program to produce different queries, even if they involve the same function with the same arguments. The idea is that, whenever a query is issued, an additional component, an ID, is added automatically, and all these IDs are distinct. Thus, even if two queries look the same to the ASM, the IDs make them distinct. (In [Blass and Gurevich 2006-7], the must-vary convention was defined for the queries issued within a single step, but we naturally take it to also apply to queries from different steps. All the IDs are distinct, whether from the same step or not.) It appears that, given a suitable formalization of ASM semantics under the must-vary convention, what we do in this paper would work under that convention as well. We do not attempt to develop the must-vary version of the theory here, but we shall add occasional remarks about how this convention would affect our discussion. □

With this rough description of the general context (see Sections 6 and 8 for a detailed description), we turn to the question of handling persistent queries and late replies in the context of interactive, small-step algorithms.

Perhaps the first approach that comes to mind is that, when an algorithm wants to use a late reply to a previously issued query q, it simply re-issues q. Then the late reply to the old q would appear, in the history of the later step, as the reply to the new q.

Remark 2.5. Under the must-vary convention, this approach would not arise, since there would be no such thing as re-issuing a query. □

The trouble with this approach is that re-issuing an old query already has a dif-

ferent meaning: It is an entirely new query, not related (in general) to the previous query. In particular, if q is issued and answered at some step and then issued again at a later step, it may get an entirely different reply the second time. In other words, the histories that occur in different steps of a computation are not required to agree in any way. Recall that each step of an algorithm's computation begins with an empty history and gradually builds up to a larger history as queries are issued and answered, but at the end of the step, its history disappears, so there is no connection between the query q issued at two different steps. Formally, this fact is incorporated in the definition of coherence and the Step Postulate in [Blass et al. 2007] (reviewed in Section 6 below), which make no allowance for any influence of the histories of earlier steps. Informally, the same fact is a consequence of the general principle that all the information from the computation's past that can affect its future must be in the state, not in some other memory of histories from previous steps.

An algorithm, having issued q in some earlier step but having received no answer, might well want to both use a late reply to that q and also issue q anew for a possibly different reply. Obviously, this situation cannot be modeled by using a re-issued q to represent looking for a late reply.

The same difficulty can also be seen in the two examples above. If the broker looked for a late reply from client B by re-issuing the query, then this would look to B like a new offer to sell a (possibly different) block of shares. Similarly, for a pollster to look at the replies he has received is quite different from sending out the questionnaires again.

A second approach also uses queries whereby the algorithm looks for late replies, but these queries will not be repetitions of the original queries. Instead, this approach is similar to the use of *implicit queries*, which was introduced in [Blass and Gurevich 2006-7, Part I, Section 2] as a way to represent an algorithm's paying attention to unsolicited information from the environment. (See also [Blass and Gurevich 2006-7, Part II, Example 5.14] and [Blass et al. 2007, Part I, Remark 3.7].) The idea here is that, when it wants to use a late reply to a query q, the algorithm is, in effect, asking the environment to provide that late reply, if one exists. That is, the algorithm issues a query asking, "What late reply, if any, has been received for the query q?"

Actually, this new query needs to be more detailed. The query q could have been issued at several earlier steps, and these occurrences of q would be treated by the environment as distinct queries, which could receive different answers. So the environment might have late replies for several of these occurrences. The algorithm needs to say which occurrence it wants. So the implicit query might have the form "What late reply, if any, has been received for the query q that I issued in step n?" To avoid the need for both the algorithm and the environment to count steps, the algorithm might assign some tags to persistent queries at the time it issues them, and inform the environment about the tags, so that it can later ask "What late reply, if any, has been received for the query q that I issued with tag t?"

Remark 2.6. Unlike the first, this second approach makes sense in the must-vary context. The preceding paragraph would, however, be modified. Instead of needing tags in case the same query is issued at several previous steps (an impossibility

under must-vary), the algorithm would need to know the IDs that were attached to its previous queries. □

A version of this approach was suggested in [Blass and Gurevich 2006-7, Part I, Section 2], even though the algorithms of that paper never finished a step with unanswered queries. Nevertheless, the possibility of a query-reply pair spanning several steps was addressed as follows. The query should be regarded as a simple message to the environment, whose answer received in the same step is an uninformative "OK," and the "real" answer in some later step should be regarded as a message from the environment, which is formally regarded as the reply to an implicit query "I'm willing to receive a message." This version leaves it up to the environment to say which old query it is answering with its new message.

The use of new queries to request late replies to old queries has some drawbacks. It requires additional work from the environment, namely storing all late replies until the algorithm asks for them, and then delivering them immediately. This produces a mismatch between the model and what would ordinarily happen in practice. A real environment would probably deliver a late reply as soon as it can and expect the algorithm to deal with it from then on.

If a reply is not yet available[4] when requested, the algorithm might well keep issuing the same request, step after step. Just reading all those requests would be a burden for the environment. This hardly matters as long as we take the algorithm's point of view and regard the environment as given. But it would matter in a distributed situation, where an agent's environment consists of other agents and the algorithms executed by those agents would have to include ways of handling a barrage of queries for which the answers aren't available.

Such approaches also clash, on a more philosophical level, with the standard notion of state, as understood in the behavioral theory of algorithms and the theory of ASMs. Once a late reply is available for delivery to the algorithm, it is something that resulted from the past steps of the computation and may be relevant to the future; so it ought to be part of the state.

A third approach, quite common in practice, is the use of futures. This is actually a whole spectrum of approaches, because there is a great variety of ways to handle futures. (For the purpose of this discussion, we do not distinguish between futures and their close relatives like promises.) At the one end would be extremely simple futures, merely waiting for a reply. At the other end would be new threads (or processes, agents, etc., but let us concentrate on threads), created to unblock the parent thread. The new thread may do plenty of work with a late reply before reporting the result to the parent thread. That work may in general have side effects and generate additional threads. Such futures would take us far from the realm of sequential computation The new threads need not be synchronized with the parent thread or with each other, so we would no longer have a global state advancing step by step.

What we do in this paper falls under the general heading of futures but barely. We stay at the simple end of the spectrum. (In fact, we don't know any system

[4]Unless otherwise specified, "available" means that the environment has the reply and is ready to provide it to the algorithm.

in the literature with such a restrictive use of futures yet interacting with an external environment.) Our proposal is that a late reply is to be recorded directly in the algorithm's state. As we shall see, this approach keeps the computation very nearly small-step. More precisely, it allows us to treat late replies as a simple, well-controlled inter-step intervention of the environment; apart from this intervention, the algorithm remains sequential. As a result, we retain the advantages of clarity, ease of programming, and ease of verification that ordinarily result from sequentiality.

The next section explores this approach in somewhat more detail.

3. ALGORITHMS WITH PERSISTENT QUERIES

In this section, we discuss the last approach mentioned above for handling persistent queries and late replies. When a late reply becomes available, the environment should record it in the state of the algorithm.

The environment's action of recording the late reply, since it takes place without a new query from the algorithm, is an inter-step interaction. It directly updates the algorithm's state, without any action by the algorithm. So this update cannot occur earlier than the first inter-step moment after the environment has a reply. It might occur later, if the environment is busy with other tasks or if communication is slow. Fortunately, such a delay makes no difference to our discussion (though it may make a difference to the efficiency of the algorithm). Indeed, from the point of view of an algorithm (or ASM) it makes no difference if a late reply, received at the start of a certain step, was actually known much earlier to the environment; the algorithm simply doesn't see that. As far as the algorithm is concerned, the only notion of "available" is "delivered to me by the environment."

If this method of communication between the environment and the algorithm is to succeed, they must agree as to where, in the algorithm's state, a late reply to a particular query is to be recorded. The environment must know where to put the reply, and the algorithm must know where to find the reply when needed. We propose that this agreement be achieved as follows. Whenever it issues a query for which a late reply might be relevant (a persistent query), the algorithm should give the environment, along with the query, a *reply location*, where any late reply to this query should be recorded.

As in [Gurevich 1995] (and all subsequent work on ASMs and the behavioral theory of algorithms), we take *location* to mean a pair $\langle f, \mathbf{a}\rangle$ where f is a function symbol of the algorithm's vocabulary and $\mathbf{a}$ is a tuple of elements of the state, an n-tuple if f is n-ary.

Recall (from [Blass and Gurevich 2006-7; Blass et al. 2007] or see Section 6 below) that a query is a tuple of elements of the disjoint union $X \sqcup \Lambda$, where X is (the underlying set of) the state and Λ is a set of labels. If the function symbol f is among the labels, then a location $\langle f, \mathbf{a}\rangle$ is almost a query. "Almost" because the location is $\langle f, \langle a_1, \ldots, a_n\rangle\rangle$ while the query is $\langle f, a_1, \ldots, a_n\rangle$; we shall ignore such bracketing distinctions in the future and write as if locations are queries.

It is not enough, however, for the algorithm to issue, along with any persistent query, its reply location as a second query (an `Output` in the sense of [Blass and Gurevich 2006-7] or an `issue` in the sense of [Blass et al. 2007], to which the envi-

ronment gives an automatic, immediate, and uninformative reply). The algorithm must tell the environment which reply location goes with which query. After all, the algorithm might issue many persistent queries simultaneously.

The simplest way for the algorithm to convey the necessary information to the environment is to issue, along with any persistent query q, a second query that contains both q and the reply location. We adopt, by convention, the following format for this second query. It is the concatenation of three sequences:

—the query q,
—the one-term sequence $\langle \texttt{rl} \rangle$, and
—the reply location l.

Here the special label `rl` (abbreviating "reply location") marks where the query ends and the reply location begins (and indicates that there *is* a reply location, i.e., that this is not just another query); we assume that this marker `rl` is chosen to be distinct from all other labels used by the algorithm.

Here a simplification is possible, if the environment is willing to cooperate. The information in the original query q is repeated in the first part of the additional message $\langle q, \texttt{rl}, l \rangle$ that specifies the reply location l. So there is no real need to issue q; it would suffice to issue $\langle q, \texttt{rl}, l \rangle$ if the environment is smart enough to interpret it as follows: Regard the part before `rl` as a query in the traditional sense, but, if the reply is late, then put it into the location given after `rl`.

As a further simplification, we adopt the convention that the reply to a persistent query should be put into its reply location even if the reply arrives during the step in which the query was issued. We impose no requirement, however, on how soon such a reply is put into the reply location. It need not happen at the end of the step in which the query was issued; it might happen at the end of some later step. The reason for this flexibility is that we do not wish to impose requirements on how fast the environment works, or even on the relative speed of different parts of the environment. Thus, one part of the environment may be able to provide an immediate reply directly to the algorithm (as in [Blass and Gurevich 2006-7]) while the part of the environment responsible for inter-step changes to the algorithm's state is slower. Fortunately, this flexibility does no harm to our theory.

Instead of having the environment put on-time replies into the reply location, we could program algorithms so that, when a persistent query is issued and answered during the same step, the algorithm writes the reply into the reply location. Our convention relieves the algorithm of this duty, assigning it to the environment instead.

Does this reassignment unduly burden the environment? One can argue that it actually makes the environment's job easier. If only late replies are to be written to the reply location, then the environment must watch the step-by-step progress of the algorithm's work, in order to know whether a particular reply is late. With our convention, the environment need not monitor the algorithm in such detail; all replies to persistent queries go into the assigned reply locations. The only difference between on-time and late replies is that the former are seen by the algorithm, in its history (or answer function), without having to wait until the end of the step.

We wish, of course, to maintain the correspondence between algorithms in general and ASMs, as established in [Gurevich 2000; Blass and Gurevich 2003; 2006-7; Blass

et al. 2007]. Accordingly, when we extend, as outlined above, the sorts of queries that algorithms can issue, we should correspondingly extend the class of ASM programs describing these algorithms. We propose the following ASM syntax for generating the combined queries — persistent query combined with reply location. (The same syntax could also be used in a framework where q and $\langle q, \mathtt{rl}, l\rangle$ are issued separately.) Suppose the query q results from a term $g(\mathbf{u})$ in an ASM. So g is an m-ary external function symbol for some m and $\mathbf{u}$ is an m-tuple of terms u_i; q results from inserting the values (in the algorithm's current state) of the u_i's in the template associated to g. (Recall from [Blass and Gurevich 2006-7, Part II, Section 4.2][5] that a template is like a query but with placeholders instead of elements of the state. An ASM provides, for each external function symbol g, a template $\hat{g}$. The query issued by g with arguments u_i is obtained by replacing the placeholders in $\hat{g}$ by the values of the u_i's.) Suppose further that the desired location for late replies is $\langle f, \mathbf{a}\rangle$. The components a_j of the tuple $\mathbf{a}$ must be the values, in the current state, of some terms t_j, in order for the algorithm to be able to refer to them. Then the algorithm can specify the desired location by means of the term $f(\mathbf{t})$. To say, in an ASM program, that the algorithm should ask the query arising from $g(\mathbf{u})$ and to specify $\langle f, \mathbf{a}\rangle$ as its reply location, we write

$$g(\mathbf{u})[=: f(\mathbf{t})].$$

For human readability, the brackets indicate that the main query here is produced by $g(\mathbf{u})$, and the reverse-assignment notation =: indicates that $f(\mathbf{t})$ is to be read as specifying a location (like the left side of an update rule written with :=) and that the value to be put there is the (eventual) value of $g(\mathbf{u})$.

In the situation described here, since $\langle f, \mathbf{a}\rangle$ is a location, the function symbol f must be in the state vocabulary, not an external function symbol. In fact, we require all function symbols in the terms $\mathbf{t}$ to be from the state vocabulary also. That is, the ASM should not need to issue queries in order to determine reply locations. This requirement arises from the combination of two circumstances. First, a reply location for a query should be determined when the query is issued, not in some later step. So any queries arising from external function symbols in $\mathbf{t}$ need to be answered in the current step, not later. Second, it turns out that whether a query must be answered in the current step depends only on the context in which it appears in the ASM program. (The relevant contexts for persistent queries are timing guards, guards built with Kleene connectives, and issue rules. In all other contexts, queries must be answered in the current step. This will be proved formally in Proposition 10.6 below.) But a persistent query $g(\mathbf{u})$ and its reply location $f(\mathbf{t})$ share the same context. So if the latter must be answered in the current step, so must the former. And then the former doesn't need a reply location.

Remark 3.1. We could relax this requirement and allow $\mathbf{t}$ to issue queries provided we have some assurance, from a source other than the context in the ASM program, that these queries will be answered in the current step. Such assurance could come from knowledge about the environment. It could also come from other parts of the ASM program. A simple example of the latter possibility is given by

[5] Or see Section 6 below for a summary

the program

```
if t = t then
   if c ≺ g(u)[=: f(t)] then
      x := 0
   endif
endif.
```

Here t, c, and g are external but f is in the state vocabulary. The symbol $\prec$ indicates a timing comparison; $r \prec s$ means that the replies needed for the evaluation of r were received before (the last of) those needed for s. Thus $r \prec s$ can have a truth value even if not all the queries it causes have been answered. For example, a step of the example algorithm can finish without a value for $g(u)$, provided c has a value. But it cannot finish without a value for t, because of the guard $t = t$ (whose sole purpose is to require that t have a value).

The same method can be used quite generally to make a query blocking; just add a guard like $t = t$ that forces the computation to wait until t has a value. Of course, if this method were to be used frequently, it might be worthwhile to introduce syntactic sugar for it, so that one could simply mark a query as blocking rather than having to explicitly write a trivial guard for it. We thank one of the referees for suggesting that such syntactic marking could be useful. □

Example 3.2. Consider a simplified version of the broker example as detailed in [Blass et al. 2007, Part I, Example 3.20]; the purpose of the simplification is to avoid hiding the currently relevant topic, persistent queries and late replies, in a sea of other considerations. We regard the broker's offers to the two clients (whom we name 0 and 1) as given by nullary external functions q_0 and q_1 (instead of ternary functions having the stock, the number of shares, and the price as arguments), and we assume the broker breaks ties (when he gets positive answers from both clients simultaneously) in favor of client 0 (rather than non-deterministically or randomly). Also, we assume that, as long as the broker has received no answer from either client, or has a negative answer from one client and no answer from the other, he simply waits. The resulting algorithm, which halts after a single step, is represented by the following ASM, in the notation of [Blass et al. 2007, Part II]. We assume that the dynamic function symbols s_0 and s_1 are used to indicate a sale to client 0 or 1, respectively, so they have the value `false` in initial states. The replies to the queries q_i represent the clients' answers, `true` indicating acceptance of the broker's offer.

```
if ¬Halt then
   do in parallel
      if s0 = s1 = false ⅄ q0 = true ⅄ (q0 ⪯ q1 ϒ q1 = false)
         then s0 := true endif
      if s0 = s1 = false ⅄ q1 = true ⅄ (q1 ≺ q0 ϒ q0 = false)
         then s1 := true endif
      if s0 = s1 = false ⅄ q0 = false ⅄ q1 = false
         then skip endif
      Halt := true
   enddo
endif
```

The so-called *Kleene conjunction* ⅄ and *Kleene disjunction* ϒ that are used in this ASM program are like ordinary conjunction ∧ and disjunction ∨ except that p ⅄ q is false as soon as one conjunct is false, even if the other is undefined, and dually for ϒ. (For more details, see [Blass et al. 2007, Part II, Section 2.3] or Section 8 below.) Thus, the first two statements in the "`do in parallel`" block say that, if one of the clients accepts the offer and the other either declines or answers too late, then the sale is made to the former client. The third statement says that if both clients decline then there is no sale. The last line, setting `Halt` to `true`, stops the algorithm after a single step; note that the step will end only after the guards in the previous three statements have acquired truth values, i.e., after it is decided whether there is to be a sale and, if so, to which client.

Convention 3.3. In future examples of ASM programs, we shall omit "`if ¬Halt then`" and the assocated "`endif`", adopting instead the convention that an ASM program is to be executed repeatedly until `Halt` becomes true. This convention supersedes the one from [Gurevich 1995] that the iteration continues until there is no change of state from one step to the next. The new convention allows an algorithm to continue waiting for a late reply without making any changes to its state.

Now suppose, as in Example 2.2 above, that we want the algorithm to respond to a late reply from a losing client with a letter explaining that the shares have already been sold. We assume (again for simplicity, to avoid hiding the relevant issues) that the broker's vocabulary contains nullary symbols l_0 and l_1 denoting appropriate letters to the two clients. And we assume that it also has nullary symbols a_0 and a_1, initially denoting `undef`, to be used as the reply locations. Then the modified algorithm, which behaves like the one above but also sends the appropriate letter, is given in our proposed syntax by the following ASM.

```
do in parallel
   if s_0 = s_1 = false ∧ q_0 = true ∧ (q_0 ≼ q_1[=: a_1] ∨ q_1 = false)
      then s_0 := true endif
   if s_0 = s_1 = false ∧ q_1 = true ∧ (q_1 ≺ q_0[=: a_0] ∨ q_0 = false)
      then s_1 := true endif
   if s_0 = s_1 = false ∧ q_0 = false ∧ q_1 = false
      then skip endif
   if s_0 = true ∧ a_1 = true then issue(l_1) endif
   if s_1 = true ∧ a_0 = true then issue(l_0) endif
   if (a_0 = true ∨ a_0 = false) ∧ (a_1 = true ∨ a_1 = false)
      then Halt := true endif
enddo
```

The first two lines have been modified by attaching reply locations a_i to the two query-producing terms q_i. (It doesn't really matter which occurrence of q_i is annotated with a_i. We chose to use the occurrence that is primarily responsible for the possibility of finishing the step without a reply.) Two new lines have been added, containing instructions for issuing the appropriate letter to the losing client. The last line makes the algorithm end its run when both clients have answered; until then, even if the shares have been sold to one client, it waits for an answer from the other client.

This example serves to illustrate a general feature of our notation. The part of the program that tells what to do after receiving late replies to the queries q_i does not mention those queries at all. Rather, it mentions the locations a_i where the late replies are to be found. The executor of the algorithm need not remember, when using a late reply, the query that it answers; only the location of the late reply is relevant, and it is used like any other location in the state.

Remark 3.4. The example also has a somewhat special property, namely that it doesn't need modes. It is common, in ASM programs, to use certain nullary, dynamic symbols as modes, to keep track of what sort of work the algorithm is currently doing. In the present example, there would be two modes, one indicating that the broker is waiting for a positive reply in order to sell the stock, and one indicating that the stock has been sold to one of the clients but the broker may still need to send a letter to the other client. It is often convenient to include such modes and update them explicitly in an ASM program. In the present case, however, this would be redundant, as the first mode is already described by $s_0 = s_1 = \texttt{false}$ and the second mode by the negation of this. □

Example 3.5. Consider again Example 2.3 about the pollster. Let us assume that the pollster sends out N questionnaires, numbered from 0 to $N - 1$, that the replies will be numbers, and that the desired output is the sum of all these numbers. For simplicity, we also assume that the questionnaires are sent one at a time and that all the replies eventually arrive, though perhaps late and out of order; our pollster algorithm will keep running without producing an output until all the replies have been received and added. The pollster first sends out all the questionnaires (using an internal variable i to keep track of where he is in this process) and then goes through all the replies, adding them one at a time (re-using

i to keep track of this process as well). We describe what the pollster does as an ASM, using the following vocabulary. As already indicated, i is a dynamic, nullary symbol ranging from 0 to N and indexing the queries and their replies (with the value N indicating that the sending or the processing has just been completed); it is initially 0. An additional dynamic, nullary symbol `all-sent`, initially `false`, tells whether all the questionnaires have been sent. Unary functions q and l send each i to the i^{th} questionnaire $q(i)$ and its reply location $l(i)$. The initial value of $l(i)$ is `undef` for each i. A dynamic, nullary function `sum`, initially 0, represents, at each step, the sum of the replies that have been added so far. Elementary arithmetic is assumed to be available to the algorithm, particularly $+$, $<$, and names for specific numbers. Here is the ASM:

```
do in parallel
   if all-sent = false ∧ i < N then do in parallel
      issue(q(i)[=: l(i)])
      i := i + 1
   enddo endif
   if all-sent = false ∧ i = N then do in parallel
      i := 0
      all-sent := true
   enddo endif
   if all-sent = true ∧ i < N ∧ l(i) ≠ undef then do in parallel
      sum := sum + l(i)
      i := i + 1
   enddo endif
   if all-sent = true ∧ i = N then Halt := true endif
enddo
```

Recall here Convention 3.3 that a run of the ASM ends when `Halt` becomes true; until then the program is executed repeatedly. We also assume that `Halt` is initially false, so that the program runs.

Remark 3.6. When justifying the "query and reply" paradigm for intra-step interaction in [Blass and Gurevich 2006-7, Part I, Section 2], we wrote that, if an algorithm sends a message to the outside world without expecting a reply, then this situation can be modeled by imagining an automatic, immediate, and uninformative reply "OK," essentially just an acknowledgment that the message was sent. The `Output` rules in [Blass and Gurevich 2006-7, Part II] and the `issue` rules in [Blass et al. 2007, Part II] were introduced to produce such messages.[6] There is, however, nothing in the official semantics in [Blass and Gurevich 2006-7] or [Blass et al. 2007] to require the environment to produce only "OK" as a reply to such queries. Although an `issue` rule cannot make use of any nontrivial information provided by its reply, nothing prohibits the existence of such information.

In fact, there are situations where such nontrivial information is to be expected, for example in

[6]In fact, `issue` rules don't need any reply at all; as far as they are concerned, the queries they issue are not blocking.

```
do in parallel
   x := q
   issue(q)
enddo
```

(where q is an external nullary symbol and x an internal dynamic one). In this (admittedly silly) program, the query produced by the issue line is also produced, with the intention of using its reply, by the update rule x := q.

Following the official semantics given for ASMs in [Blass et al. 2007, Part II], we make no special assumptions about the replies to queries that result from issue rules. These replies can be any elements of the state, just as for any other queries. □

This aspect of the official semantics was used in Example 3.5, because the queries produced by issue(q(i)[=: l(i)]) are the questionnaires, whose replies should be the numbers stored in locations $l(i)$ and then added.

This example also used the earlier convention, whereby replies go into the reply locations even if they are not late. Without this convention, the ASM program would have to include instructions whereby, if an answer to $q(i)$ appears in the same step in which the query was issued, the algorithm would put that answer into location $l(i)$.

4. EXTENSIONS

The approach outlined above imposes a requirement, albeit a weak one, on the environment: it must correctly understand queries with reply locations, and it must (eventually) put any replies that arrive into the indicated locations. To increase the expressivity of algorithms, one may prefer that environments cooperate in more complicated ways with the algorithm. In this section, we discuss some aspects of the cooperation between algorithms and their environments as well as some related aspects of levels of abstraction.

A natural way to implement queries and replies in an object-oriented approach is to implement each query as an object with a field for the reply. Even if one ignores the possibility of late replies and one works in the framework of the earlier papers [Blass and Gurevich 2006-7] and [Blass et al. 2007], this implementation is a natural explanation of how the environment's replies suddenly appear in the algorithm's answer function or history.

Depending on the desired level of abstraction, these query-objects can be regarded as part of the algorithm or they can be abstracted away as part of the environment. Let us first consider the low-level point of view, where these objects are part of the algorithm. Then the behavior of the environment exactly matches what we described in Section 3. Each query comes with a specified location into which the environment puts the reply, namely the reply field in the object that implements the query.

Now let us consider the higher-level point of view where the query-objects are part of the environment, not part of the algorithm. In particular, the reply-fields of these objects are not the reply locations that the algorithm indicates when it issues persistent queries. In this situation, the delivery of replies to the algorithm is part

of the task of these query-objects. In order to match our description in Section 3, the query-objects should have a method, triggered by the arrival of a reply, to write it into the desired reply-location.

It is often useful to think in terms of a "near environment" and a "far environment." The former consists of things like the query-objects here, usually on the same machine as the algorithm and hence rapidly and reliably accessible by the algorithm. The far environment is usually located elsewhere, accessed via a network, and perhaps not so rapidly or reliably accessible. As in our discussion of query-objects above, the near environment may be programmed along with the algorithm, so that, on a low-level view, it would actually be part of the algorithm; it counts as environment because it has been abstracted away from the algorithm in the passage to a higher level of abstraction.

Given the distinction between near and far environments, one has a picture with two boundaries, one between these two parts of the environment, where information is transferred by a network protocol such as TCP/IP, and one between the near environment and the algorithm, where information is transferred more directly.[7] Depending on the level of abstraction, either of these boundaries could be regarded as the boundary between the environment and the algorithm. The far-near boundary plays this role at low levels of abstraction, while the other boundary plays the same role at high levels of abstraction.

Concerning the interface between far and near environments, we shall simply assume that it works correctly, and we note that "correctly" here corresponds well with our description in Section 3 of what is expected from the environment — it puts replies to the algorithm's queries into the places requested by the algorithm.

The interface between near environment and algorithm includes in particular the action of the query-objects. This action is under the programmer's control. The simplest action, namely copying the replies into appropriate locations in the algorithm's state, again corresponds to our description in Section 3. The query-objects could, however, do other work. For example, instead of putting replies into prespecified locations, they could put them into a linked list in lexicographic order (if the replies are such that this ordering makes sense), or they could put them into an array in order of arrival. One can obviously imagine far more complicated tasks that might be performed; the query-objects could, in effect, function as new computation threads. So there is a wide spectrum of possible complexities for the behavior of query-objects. It is an interesting open problem to identify natural cut-points along this spectrum, such that useful consequences follow from the assumption that the behavior of query-objects is on the "simple" side of such a cut-point.

Our model, implemented with the query-objects as the part of the environment responsible for getting replies to the right locations, is at the simple end of this spectrum. The query-objects merely copy their reply fields into the appropriate locations. General futures can be at the complex end of the spectrum. There the query-objects could behave like processes in a parallel algorithm or even like asynchronous agents in a distributed algorithm. The simple end of the spectrum has advantages in clarity, ease of implementation, and ease of verification. In particular, in the high-level picture where the query-objects are viewed as part of the environ-

[7] This description assumes that the algorithm does not interface directly with the far environment.

ment, our model allows algorithms to be viewed as sequential, thereby simplifying all aspects of working with them. Even though the query-objects deliver replies to the algorithm in parallel (and possibly many of them act at a single step of the algorithm), this parallelism is absorbed into the simple principle that replies arrive in the prescribed locations, and apart from this principle everything is sequential.

A natural topic for future investigation would be the possibility of similar high-level descriptions of situations where the query-objects do some slightly more complicated work, perhaps the chronological or lexicographic sorting mentioned above. The goal would be to describe a class of behaviors of query-objects that can be usefully summarized in terms of the results of those behaviors, regarded as inter-step interventions by the (near) environment. The criterion for such summaries to be useful is that, by allowing us to regard the algorithm as sequential except for these interventions, they should serve to simplify the implementation, understanding, and analysis of the algorithm.

As an example of software to which such an analysis would be applicable, consider a conference manager system like EasyChair [EasyChair 2009]. Such a system is a natural example for this sort of analysis, since much of its work consists of managing interactions with the (far) environment, including authors, program committee members, and referees. That work includes several tasks that could be handled by query-objects in the near environment and could thus be abstracted away in a high-level view of the software. Among these tasks are putting the incoming submissions into alphabetical order by authors' surnames (or perhaps into chronological order by time of submission), linking the referees' reports and the program committee's comments to the proper papers, and putting the program committee's comments on each paper into chronological order, perhaps with links indicating which comments were replies to which previous comments.

All these tasks seem to fall within the scope of near-environment work that can be usefully abstracted away as assumptions about how the environment will deliver replies. Indeed, one of these tasks, that of attaching referees' reports to the right papers, is covered by the extremely simple model of the present paper. Each request to a referee is a query whose (late) reply is to go into a predetermined location associated with the corresponding paper and referee. The other tasks are slightly more complicated, but their results are easily described, so they could reasonably be accommodated in a simple extension of our present work.

5. RELATED WORK

This paper belongs to behavioral computation theory, which started from the analysis of small-step algorithms in [Gurevich 2000]. Parallel algorithms were analyzed in [Blass and Gurevich 2003], ordinary interactive algorithms were analyzed in [Blass and Gurevich 2006-7], and general interactive algorithms were analyzed in [Blass et al. 2007].

Rosenzweig and Runje [Rosenzweig and Runje 2005] have analyzed in considerable detail what can happen (and what cannot) during runs of small-step algorithms, either in isolation or with ordinary interaction with the environment. One prospective application of this work is in ASM models of cryptographic protocols, in the style of [Rosenzweig et al. 2003], where it is important to know that certain

things cannot happen during execution of a protocol.

Reisig [Reisig 2008] described small-step ASMs from a more syntactic point of view, and he used this description both to analyze the connection between ASMs and the classical notion of computability and to extend that classical notion to general structures.

Millar [Millar 2006] has analyzed the effect, in the presence of ordinary interaction, of weakening the Bounded Exploration Postulate for small-step algorithms, to merely require finite exploration.

Yavorskaya [Gurevich and Yavorskaya 2006, Chapter II], following the ideas sketched in [Gurevich 2000], has extended the ASM representation theorem to small-step algorithms that involve bounded nondeterminism. For unbounded nondeterminism, Glausch and Reisig [Glausch and Reisig 2008] provided an axiomatization and an ASM syntax and semantics, and they proved the ASM representation theorem for this situation.

In [Glausch and Reisig 2009], Glausch and Reisig achieved analogous results for a certain (restricted) class of distributed algorithms. Each algorithm in this class, though distributed, consists of only a fixed finite number of agents, each executing a small-step algorithm.

Boker and Dershowitz [Boker and Dershowitz 2005] built on and modified the axioms for sequential algorithms in [Gurevich 2000] to describe models of computation and, in particular, effective models. For example, a particular Turing machine defines an algorithm; Turing machines as a class constitute a model of computation. The same authors showed in the second part of [Boker and Dershowitz 2008], that Turing machines can simulate any effective model of computation (over any countable domain). For more on the connection between ASMs and the Church-Turing thesis, see also [Dershowitz and Gurevich 2008].

Beauquier and Slissenko (see [Beauquier and Slissenko 2006] and the references there) set up a version of ASM semantics to handle timing and showed how to use it for the formal verification of properties of timed algorithms. The use of ASMs for a particular example of a timed algorithm had been worked out earlier in [Gurevich and Huggins 1996].

The present paper can also be regarded as being about futures, specifically about bringing an extremely simple version of futures into the scope of small-step algorithms and the corresponding ASMs. There is, however, a problem: There is an ocean of work on futures (and their close relatives like promises), and we don't know any work on futures that is very close to what we are doing here. So let us just say a few words on the origin of the notion of futures and quote the papers advised by one of our referees.

The original idea of futures goes all the way back to 1976 paper [Friedman and Wise 1976] where Daniel Friedman and David Wise redefined the `cons` function of LISP to be non-strict. "Instead of evaluating its arguments, it builds suspensions of them which are not coerced until the suspension is accessed by a strict elementary function. This approach turns out to be useful: The resulting evaluation procedures are strictly more powerful than existing schemes for languages such as LISP." A year later, Henry Baker and Carl Hewitt coined a name for the unusual evaluation order [Baker and Hewitt 1997]: call-by-future. "[I]n call-by-future, each formal

parameter of a function is bound to a separate process (called a 'future') dedicated to the evaluation of the corresponding argument. This mechanism allows the fully parallel evaluation of arguments to a function, and has been shown to augment the expressive power of a language." The name stuck.

One of the referees noted a similarity between persistent queries and event-driven programming [Dabek et al. 2002]. The idea is that, by issuing a persistent query, an algorithm subscribes to a single event, namely a (possibly late) reply to that query. We point out, however, that the computation need not respond to such events promptly; a late reply might become relevant to the computation only many steps after it has been inserted into the state. In particular, even if an algorithm has issued many persistent queries in the past and then suddenly receives a vast number of replies, it will immediately deal with only a bounded number of these replies. It remains a small-step algorithm.

PART II: AXIOM DETAILS

6. INTERACTIVE SMALL-STEP ALGORITHMS

We now begin a more formal treatment of algorithms with persistent queries. We build on the axiomatic treatment described in [Blass et al. 2007]. In the present section, we summarize the definitions, conventions, and postulates from [Blass et al. 2007] that we need here. This material is taken from [Blass et al. 2007, Part I, Section 3]. This summary also serves to explain some things that were taken for granted in the preceding sections. We do not, however, repeat the extensive discussion offered in [Blass et al. 2007] to motivate and explain the model.

As in earlier papers [Gurevich 2000; Blass and Gurevich 2003; 2006-7; Blass et al. 2007], the postulates are at first stated in terms of "the algorithm" in an informal sense; subsequently, Definition 6.14 gives a formal meaning to "interactive small-step algorithm" as any entity satisfying the postulates.

Also as in earlier papers, we use the following conventions concerning vocabularies and structures, explaining how our usage differs from what is traditional in model theory. (Readers who don't know or don't care about the traditional usage can take Convention 6.1 as a definition.) The first, fourth, and last of these conventions are common in general algebra; several of the others would be common in model theory if relations were always regarded as truth-functions.

Convention 6.1.

—A vocabulary Υ consists of function symbols with specified arities.

—Some of the symbols in Υ may be marked as *static*, and some may be marked as *relational*. Symbols not marked as static are called *dynamic*.

—Among the symbols in Υ are the logic names: nullary symbols `true`, `false`, and `undef`; unary `Boole`; binary equality; and the usual propositional connectives. All of these are static and all but `undef` are relational.

—An Υ-*structure* X consists of a nonempty base set, usually denoted by the same symbol X, and interpretations of all the function symbols f of Υ as functions f_X on that base set.

—In any Υ-structure, the interpretations of `true`, `false`, and `undef` are distinct.

—In any Υ-structure X, the interpretations of relational symbols are functions whose values lie in $\{\mathtt{true}_X, \mathtt{false}_X\}$.
—In any Υ-structure X, the interpretation of `Boole` maps $\mathtt{true}_X$ and $\mathtt{false}_X$ to $\mathtt{true}_X$ and everything else to $\mathtt{false}_X$.
—In any Υ-structure X, the interpretation of equality maps pairs of equal elements to $\mathtt{true}_X$ and all other pairs to $\mathtt{false}_X$.
—In any Υ-structure X, the propositional connectives are interpreted in the usual way when their arguments are in $\{\mathtt{true}_X, \mathtt{false}_X\}$, and they take the value $\mathtt{false}_X$ whenever any argument is not in $\{\mathtt{true}_X, \mathtt{false}_X\}$.
—We may omit subscripts X, for example from `true` and `false`, when there is no danger of confusion. □

States Postulate: The algorithm determines

—a finite vocabulary Υ,
—a nonempty set $\mathcal{S}$ of *states*, which are Υ-structures,
—a nonempty subset $\mathcal{I} \subseteq \mathcal{S}$ of *initial states*,
—a finite set Λ of *labels* (to be used in forming queries).

Definition 6.2. A *potential query* in state X is a finite tuple of elements of $X \sqcup \Lambda$. A *potential reply* in X is an element of X. □

Here $X \sqcup \Lambda$ means the disjoint union of X and Λ. So if they are not disjoint, then they are to be replaced by disjoint isomorphic copies. We shall usually not mention these isomorphisms; that is, we write as though X and Λ were disjoint.

Definition 6.3. An *answer function* for a state X is a partial map from potential queries to potential replies. A *history* for X is a pair $\xi = \langle \dot{\xi}, \leq_\xi \rangle$ consisting of an answer function $\dot{\xi}$ together with a linear pre-order $\leq_\xi$ of its domain. By the *domain* of a history ξ, we mean the domain $\mathrm{Dom}(\dot{\xi})$ of its answer function component, which is also the field of its pre-order component. □

Recall that a *pre-order* of a set D is a reflexive, transitive, binary relation on D, and that it is said to be *linear* if, for all $x, y \in D$, $x \leq y$ or $y \leq x$. The equivalence relation defined by a pre-order is given by

$$x \equiv y \iff x \leq y \leq x.$$

The equivalence classes are partially ordered by

$$[x] \leq [y] \iff x \leq y,$$

and this partial order is linear if and only if the pre-order was.

We also write $x < y$ to mean $x \leq y$ and $y \not\leq x$. (Because a pre-order need not be antisymmetric, $x < y$ is in general a stronger statement than the conjunction of $x \leq y$ and $x \neq y$.) When, as in the definition above, a pre-order is written as $\leq_\xi$, we write the corresponding equivalence relation and strict order as $\equiv_\xi$ and $<_\xi$. The same applies to other subscripts and superscripts.

We use histories to express the information received by the algorithm from its environment during a step. The answer function part $\dot{\xi}$ of a history tells what replies the environment has given to the algorithm's queries, and the pre-order part $\leq_\xi$ tells in what order these replies were received. Specifically, if q is in the domain of ξ, then $\dot{\xi}(q)$ is the environment's answer to the query q. If $p, q \in \mathrm{Dom}(\xi)$ and $p <_\xi q$, this means that the answer $\dot{\xi}(p)$ to p was received strictly before the answer $\dot{\xi}(q)$ to q. If $p \equiv_\xi q$, this means that the two answers were received simultaneously.

We emphasize that the timing we are concerned with here is logical time, not physical time. That is, it is measured by the progress of the computation, not by an external clock. In particular, we regard a query as being issued by the algorithm as soon as the information causing that query (in the sense of the Interaction Postulate below) is in the history. This is why we include, in histories, only the relative ordering of replies. The ordering of queries relative to replies or relative to each other is then determined. The logical time of a query is the same as the logical time of the last of the replies needed to cause that query.

Definition 6.4. Let $\leq$ be a pre-order of a set D. An *initial segment* of D with respect to $\leq$ is a subset S of D such that whenever $x \leq y$ and $y \in S$ then $x \in S$. An *initial segment* of $\leq$ is the restriction of $\leq$ to an initial segment of D with respect to $\leq$. An *initial segment* of a history $\langle \dot{\xi}, \leq_\xi \rangle$ is a history $\langle \dot{\xi} \restriction S, \leq_\xi \restriction S \rangle$, where S is an initial segment of $\mathrm{Dom}(\dot{\xi})$ with respect to $\leq_\xi$. (We use the standard notation $\restriction$ for the restriction of a function or a relation to a set.) We write $\eta \trianglelefteq \xi$ to mean that the history η is an initial segment of the history ξ. If $q \in D$, then we define two associated initial segments as follows.

$$(\leq q) = \{d \in D : d \leq q\}$$
$$(< q) = \{d \in D : d < q\}. \quad \square$$

Interaction Postulate For each state X, the algorithm determines a binary relation $\vdash_X$, called the *causality relation*, between finite histories and potential queries.

The intended meaning of $\xi \vdash_X q$ is that, if the algorithm's current state is X and the history of its interaction so far (as seen by the algorithm during the current step) is ξ, then it will issue the query q unless it has already done so in the current step. When we say that the history so far is ξ, we mean not only that the environment has given the replies indicated in $\dot{\xi}$ in the order given by $\leq_\xi$, but also that no other queries have been answered. Thus, although ξ explicitly contains only positive information about the replies received so far, it also implicitly contains the negative information that there have been no other replies. Of course, if additional replies are received later, so that the new history has ξ as a proper initial segment, then q is still among the issued queries, because it was issued at the earlier time when the history was only ξ. This observation is formalized as follows.

Definition 6.5. For any state X and history ξ, we define sets of queries

$$\mathrm{Issued}_X(\xi) = \{q : (\exists \eta \trianglelefteq \xi)\, \eta \vdash_X q\}$$
$$\mathrm{Pending}_X(\xi) = \mathrm{Issued}_X(\xi) - \mathrm{Dom}(\dot{\xi}). \quad \square$$

Thus, $\mathrm{Issued}_X(\xi)$ is the set of queries that have been issued by the algorithm, in state X, by the time the history is ξ, and $\mathrm{Pending}_X(\xi)$ is the subset of those that have, as yet, no replies.

The following definition describes the histories that are consistent with the given causality relation. Informally, these are the histories where every query in the domain has a legitimate reason, under the causality relation, for being there.

Definition 6.6. A history ξ is *coherent*, with respect to a state X or its associated causality relation $\vdash_X$, if $\mathrm{Dom}(\dot{\xi})$ is finite and

$$(\forall q \in \mathrm{Dom}(\dot{\xi}))\, q \in \mathrm{Issued}_X(\xi \restriction (< q))$$

□

Remark 6.7. In [Blass et al. 2007, Part I, Definition 3.12], the definition of coherence did not require $\mathrm{Dom}(\dot{\xi})$ to be finite; instead, it had the weaker requirement that the linear order of $\equiv_\xi$-classes induced by $\leq_\xi$ is a well-order. The stronger requirement of finiteness was, however, deduced later from the Bounded Work Postulate for all attainable histories; see [Blass et al. 2007, Part I, Corollary 3.28]. Since we omit such deductions here, it seems clearer to build finiteness explicitly into the notion of coherent history. □

Definition 6.8. A history ξ for a state X is *complete* if $\mathrm{Pending}_X(\xi) = \varnothing$. □

The terminology reflects the fact that, if a complete history has arisen in the course of a computation, then there will be no further interaction with the environment during this step. No further interaction can originate with the environment, because no queries remain to be answered. No further interaction can originate with the algorithm, since ξ and its initial segments don't cause any further queries. So the algorithm must either terminate its run (successfully) if `Halt` becomes true, or proceed to the next step (by updating its state), or fail. The next definitions and postulates describe these end-of-step matters. They do not explicitly mention termination (other than by failure), but this is covered anyway, since updates are covered and termination amounts to an update of `Halt` to the value `true`.

Definition 6.9. A *location* in a state X is a pair $\langle f, \mathbf{a}\rangle$ where f is a dynamic function symbol from Υ and $\mathbf{a}$ is a tuple of elements of X, of the right length to serve as an argument for the function f_X interpreting the symbol f in the state X. The *value* of this location in X is $f_X(\mathbf{a})$. An *update* for X is a pair (l, b) consisting of a location l and an element b of X. An update (l, b) is *trivial* (in X) if b is the value of l in X. We often omit parentheses and brackets, writing locations as $\langle f, a_1, \ldots, a_n\rangle$ instead of $\langle f, \langle a_1, \ldots, a_n\rangle\rangle$ and writing updates as $\langle f, \mathbf{a}, b\rangle$ or $\langle f, a_1, \ldots, a_n, b\rangle$ instead of $(\langle f, \mathbf{a}\rangle, b)$ or $(\langle f, \langle a_1, \ldots, a_n\rangle\rangle, b)$. □

The intended meaning of an update $\langle f, \mathbf{a}, b\rangle$ is that the interpretation of f is to be changed (if necessary, i.e., if the update is not trivial) so that its value at $\mathbf{a}$ is b.

Step Postulate — Part A The algorithm determines, for each state X, a set $\mathcal{F}_X$ of *final histories*. Every complete, coherent history has an initial segment (possibly the whole history) in $\mathcal{F}_X$.

Intuitively, a history is final for X if, whenever it arises in the course of a computation in X, the algorithm completes its step, either by failing or by executing its updates and proceeding to the next step or terminating the run if `Halt` has become true.

Definition 6.10. A history for a state X is *attainable* (in X) if it is coherent and no proper initial segment of it is final. □

The attainable histories are those that can occur under the given causality relation and the given choice of final histories. That is, not only are the queries answered in an order consistent with $\vdash_X$ (coherence), but the history does not continue beyond where $\mathcal{F}_X$ says it should stop.

Step Postulate — Part B For each state X, the algorithm determines that certain histories *succeed* and others *fail*. Every final, attainable history either succeeds or fails but not both.

Definition 6.11. We write $\mathcal{F}_X^+$ for the set of successful final histories and $\mathcal{F}_X^-$ for the set of failing final histories.

The intended meaning of "succeed" and "fail" is that a successful final history is one in which the algorithm finishes its step and performs a set of updates of its state, while a failing final history is one in which the algorithm cannot continue — the step ends, but there is no next state, not even a repetition of the current state. Such a situation can arise if the algorithm computes inconsistent updates. It can also arise if the environment gives inappropriate answers to some queries.

Step Postulate — Part C For each attainable history $\xi \in \mathcal{F}_X^+$ for a state X, the algorithm determines an *update set* $\Delta^+(X, \xi)$, whose elements are updates for X. It also produces a *next state* $\tau(X, \xi)$, which

—has the same base set as X,
—has $f_{\tau(X,\xi)}(\mathbf{a}) = b$ if $\langle f, \mathbf{a}, b\rangle \in \Delta^+(X, \xi)$, and
—otherwise interprets function symbols as in X.

Convention 6.12. In notations like $\mathcal{F}_X$, $\mathcal{F}_X^+$, $\mathcal{F}_X^-$, $\Delta^+(X, \xi)$, and $\tau(X, \xi)$, we may omit X if only one X is under discussion. We may also add the algorithm A as a superscript if several algorithms are under discussion. □

Any isomorphism $i : X \cong Y$ between states can be extended in an obvious, canonical way to act on queries, answer functions, histories, locations, updates, etc. We use the same symbol i for all these extensions.

Isomorphism Postulate Suppose X is a state and $i : X \cong Y$ is an isomorphism of Υ-structures. Then:

—Y is a state, initial if X is.
—i preserves causality, that is, if $\xi \vdash_X q$ then $i(\xi) \vdash_Y i(q)$.

—i preserves finality, success, and failure, that is, $i(\mathcal{F}^+_X) = \mathcal{F}^+_Y$ and $i(\mathcal{F}^-_X) = \mathcal{F}^-_Y$.
—i preserves updates, that is, $i(\Delta^+(X, \xi)) = \Delta^+(Y, i(\xi))$ for all histories ξ for X.

Convention 6.13. In the last part of this postulate, and throughout this paper, we adopt the convention that an equation between possibly undefined expressions is to be understood as implying that if either side is defined then so is the other. □

Bounded Work Postulate

—There is a bound, depending only on the algorithm, for the lengths of the tuples in $\text{Issued}_X(\xi)$, for all states X and final, attainable histories ξ.
—There is a bound, depending only on the algorithm, for the cardinality $|\text{Issued}_X(\xi)|$, for all states X and final, attainable histories ξ.
—There is a finite set W of Υ-terms (possibly involving variables), depending only on the algorithm, with the following property. Suppose X and X' are two states and ξ is a history for both of them. Suppose further that each term in W has the same value in X as in X' when the variables are given the same values in $\text{Range}(\dot{\xi})$. Then:
  —If $\xi \vdash_X q$ then $\xi \vdash_{X'} q$ (so in particular q is a query for X').
  —If ξ is in $\mathcal{F}^+_X$ or $\mathcal{F}^-_X$, then it is also in $\mathcal{F}^+_{X'}$ or $\mathcal{F}^-_{X'}$, respectively.
  —$\Delta^+(X, \xi) = \Delta^+(X', \xi)$.

Definition 6.14. An *interactive, small-step algorithm* is any entity satisfying the States, Interaction, Step, Isomorphism, and Bounded Work Postulates. □

Since these are the only algorithms under consideration in most of this paper, we often omit "interactive, small-step."

Definition 6.15. A set W with the property required in the third part of the Bounded Work Postulate is called a *bounded exploration witness* for the algorithm. Two pairs (X, ξ) and (X', ξ), consisting of states X and X' and a single ξ that is a history for both, are said to *agree* on W if, as in the postulate, each term in W has the same value in X as in X' when the variables are given the same values in $\text{Range}(\dot{\xi})$. □

This completes our review of the notion of interactive small-step algorithm, as defined in [Blass et al. 2007, Part I]. This notion will be slightly modified in Section 7 to accommodate our proposal for handling persistent queries. A modification is needed because, when an algorithm issues a combination $\langle q, \texttt{rl}, l\rangle$ of a query q and a reply location l, the reply (if received in the same step) is a reply to q. So it is q, not $\langle q, \texttt{rl}, l\rangle$, that should appear in the domain of the history. See Section 7 for more details.

7. ANNOUNCING LOCATIONS FOR LATE REPLIES

In this section, we present the small modifications of the axiomatic part of [Blass et al. 2007] needed to accommodate the $\langle q, \mathtt{rl}, l\rangle$ method, proposed in Section 3, for handling persistent queries. Later, in Section 9, we shall do the same for the ASM part of [Blass et al. 2007]. In both of these sections, we shall use the heading "Modification" for the essential changes in the axiomatic set-up and the syntax and semantics of ASMs. If these modifications are violated, then our algorithms and ASM programs with persistent queries won't make sense. Other changes describe what we expect to see in programs and in the environment's behavior. These concern either constraints on the environment or good programming practice; we label these "Intention." If they are violated, algorithms and ASM programs with persistent queries will still make sense, but it may not be the sense that was intended.

Modification 1. The set Λ of labels contains the "reply location marker" `rl`.

The purpose of this modification is of course to ensure availability of the queries $\langle q, \mathtt{rl}, l\rangle$ that we want to use when issuing a persistent query q with reply location l. It may seem that we should also require that all dynamic function symbols f should be among the labels, so that they can be used in the location part l of $\langle q, \mathtt{rl}, l\rangle$. This requirement would do no harm, but it may be overkill, since there may be many dynamic function symbols that will not be used for reply locations in a particular program. Accordingly, we do not impose this requirement but instead use the following definition to keep track of which function symbols are available to serve as the first component of a reply location.

Definition 7.1. A function symbol is *reply-available* if it is dynamic and is also a member of the set Λ of labels.

Remark 7.2. We have insisted here that the function-symbol component of a reply location be dynamic. This is not strictly necessary; one could imagine using a static function — one that the algorithm can never update — in this role, since the updates would be done by the environment, not by the algorithm. But it seems strange to allow this when the update is being done at the request of the algorithm.

Notice, for example, the following undesirable consequence of allowing reply locations that begin with a static function. Suppose the environment can provide echoes; that is, an algorithm can issue a query of the form "answer this with x," where x is an element of the state, and get reply x. Then by issuing the query

$$\langle \text{answer this with } x, \mathtt{rl}, f, \mathbf{t}_X\rangle,$$

the algorithm can achieve (after the end of the current step) the effect of the update $f(\mathbf{t}) := x$. That should not be possible when f is static. □

Remark 7.3. The definition of reply-available is designed to cohere with our convention in Section 3 about the format of the queries that provide reply locations. Had we chosen a different format, for example using some codes for the function symbols, then the definition should be modified accordingly. □

The next modification provides that, when the algorithm issues a query that contains the `rl` label, it should get an answer to the "query part" preceding `rl`, since the rest merely specifies a reply location. It turns out that the only change

needed in the definitions and postulates from [Blass et al. 2007] is that, when $\langle q, \mathtt{rl}, l\rangle$ is caused by an initial segment of a history ξ, it is not this query itself but rather the initial segment q that counts as issued.

Modification 2. The definition of $\mathrm{Issued}_X(\xi)$ in Definition 6.5 is amended as follows. $\mathrm{Issued}_X(\xi)$ consists of those queries q such that

—q does not contain `rl`, and

—for some initial segment η of ξ, either $\eta \vdash_X q$ or $\eta \vdash_X \langle q, \mathtt{rl}, l\rangle$ for some sequence l.

Queries should contain at most one occurrence of `rl`. Nevertheless, our modification of the definition of Issued can be applied to queries with several `rl`'s; the first occurrence of `rl` is the one that counts.

Remark 7.4. An algorithm might produce, within one step, two or more queries that differ only in their reply locations, i.e., $\langle q, \mathtt{rl}, l\rangle$ with the same q but different l's. Our redefinition of Issued says that in this situation only the query q is put into $\mathrm{Issued}_X(\xi)$. According to Intention 7.5 below, a reply to this single query is to be written into all of the associated reply locations l.

To see that this is as it should be, consider the algorithm (in the traditional sense) that results from deleting all the reply locations. Instead of issuing any $\langle q, \mathtt{rl}, l\rangle$, it would simply issue q, the same q regardless of l. In view of the Lipari convention, q would be issued once during this step. Thus, our modifications and definitions ensure that the algorithm with reply locations behaves, in this respect, the same as the one without reply locations. □

It remains only to formally state, as intentions, the constraints that the environment should obey in order to make our algorithms with persistent queries behave as intended.

Intention 7.5. If the algorithm has produced the query $\langle q, \mathtt{rl}, l\rangle$, thereby issuing q, and if l is a location, then the answer to q should be written into location l. In case several answers are to be put into the same l at the same time, the environment chooses one of them arbitrarily. The environment should not write into reply locations except as prescribed here.

Usually, a program will not use the same reply location for several different queries, and so the need for an arbitrary choice will not arise. If, however, the program does assign the same reply location to several queries, then not only might it encounter the nondeterminism described here, but replies might be overwritten.

Note that, in telling the environment to write replies into the prescribed locations, we have made no exception for on-time replies. If a query is answered during the same step in which it was issued, then the reply goes into both the history of that step and (after the step ends) the reply location.

Intention 7.6. Replies to persistent queries are different from `undef`.

The point of this is to enable an algorithm to detect whether a query has received a late reply. If the value of the reply location is initialized¡ to `undef` and is never updated except by a reply to q, then the presence of a reply can be detected by comparing the value of this location to `undef`.

PART III: ABSTRACT STATE MACHINES

8. REVIEW OF ABSTRACT STATE MACHINES

We now turn to the notion of abstract state machine (ASM) from Part II of [Blass et al. 2007]. ASMs describe algorithms, and the main result of [Blass et al. 2007] is that all algorithms (as defined in Section 6) are behaviorally equivalent (in a very strong sense defined in [Blass et al. 2007, Part I, Section 4]) to ASMs. We begin our review of ASMs by summarizing the syntactic definitions from [Blass et al. 2007, Part II, Section 2]; afterward, we shall also summarize the semantics.

An ASM uses a vocabulary Υ, subject to Convention 6.1, and a set Λ of labels as in Section 6. In addition, it has an *external vocabulary* E, consisting of finitely many *external function symbols*. These symbols are used syntactically exactly like static, non-relational symbols from Υ, but their semantics will be quite different. If f is an n-ary external function symbol and $\mathbf{a}$ is an n-tuple of arguments from a state X, then the value of f at $\mathbf{a}$ is not stored as part of the structure of the state but is obtained from the environment as the reply to a query. If the history contains no reply to this query, then f has no value at $\mathbf{a}$.

Definition 8.1. The set of *terms* is the smallest set containing $f(t_1, \dots, t_n)$ whenever it contains $t_1, \dots, t_n$ and f is an n-ary function symbol from $\Upsilon \cup \mathrm{E}$. (The basis of this recursive definition is, of course, given by the 0-ary function symbols.) A *Boolean term* is a term of the form $f(\mathbf{t})$ where f is a relational symbol. □

Convention 8.2. By Υ-*terms*, we mean terms built using the function symbols in Υ and variables. These are terms in the usual sense of first-order logic for the vocabulary Υ. They occur, for example, in the Bounded Work Postulate as elements of the bounded exploration witness. Terms as defined above, using function symbols from $\Upsilon \cup \mathrm{E}$ but not using variables, will be called *ASM-terms* when we wish to emphasize the distinction from Υ-terms. A term of the form $f(\mathbf{t})$ where $f \in \mathrm{E}$ is called a *query-term*.

We introduce timing explicitly into the formalism with the notation $(s \preceq t)$, which is intended to mean that the replies needed to evaluate the term s arrived no later than those needed to evaluate t. As explained in [Blass et al. 2007], $\preceq$ differs from function symbols in that $s \preceq t$ can have a truth value even when only one of s and t has a value.

We also use a version of the Boolean connectives with similar behavior, so that, for example, a disjunction counts as true as soon as one of the disjuncts is, even if the other disjunct has no truth value. This behavior characterizes the connectives of Kleene's strong three-valued logic. We use the notations $\curlywedge$ and $\curlyvee$ for the conjunction and disjunction of this logic; the traditional conjunction and disjunction, which have values only when both constituents do, will continue to be written $\wedge$ and $\vee$.

Definition 8.3. The set of *guards* is defined by the following recursion.

—Every Boolean term is a guard.

—If s and t are terms, then $(s \preceq t)$ is a guard.

—If φ and ψ are guards, then so are $(\varphi \curlywedge \psi)$, $(\varphi \curlyvee \psi)$, and $\neg\varphi$.

□

Definition 8.4. The set of ASM *rules* is defined by the following recursion.

—If $f \in \Upsilon$ is a dynamic n-ary function symbol, if $t_1, \ldots, t_n$ are terms, and if t_0 is a term that is Boolean if f is relational, then

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

is a rule, called an *update* rule.

—If $f \in \mathrm{E}$ is an external n-ary function symbol and if $t_1, \ldots, t_n$ are terms, then

$$\texttt{issue}\ f(t_1, \ldots, t_n)$$

is a rule, called an *issue* rule.

—`fail` is a rule.

—If φ is a guard and if R_0 and R_1 are rules, then

$$\texttt{if}\ \varphi\ \texttt{then}\ R_0\ \texttt{else}\ R_1\ \texttt{endif}$$

is a rule, called a *conditional* rule. R_0 and R_1 are its *true* and *false branches*, respectively.

—If k is a natural number (possibly zero) and if $R_1, \ldots, R_k$ are rules then

$$\texttt{do in parallel}\ R_1, \ldots, R_k\ \texttt{enddo}$$

is a rule, called a *parallel combination* or *block* with the subrules R_i as its *components*.

□

The correspondence between external function calls and queries is mediated by a template assignment, defined as follows.

Definition 8.5. For a fixed label set Λ, a *template* for n-ary function symbols is any tuple in which certain positions are filled with labels from Λ while the rest are filled with the *placeholders* $\#1, \ldots, \#n$, occurring once each. We assume that these placeholders are distinct from all the other symbols under discussion ($\Upsilon \cup \mathrm{E} \cup \Lambda$). If Q is a template for n-ary functions, then we write $Q[a_1, \ldots, a_n]$ for the result of replacing each placeholder $\#i$ in Q by the corresponding a_i. □

Thus if the a_i are elements of a state X then $Q[a_1, \ldots, a_n]$ is a potential query in X.

Definition 8.6. For a fixed label set and external vocabulary, a *template assignment* is a function assigning to each n-ary external function symbol f a template $\hat{f}$ for n-ary functions. □

The intention, which will be formalized in the semantic definitions below, is that when an ASM evaluates a term $f(t_1, \ldots, t_n)$ where $f \in \mathrm{E}$, it first computes the values a_i of the terms t_i, then issues the query $\hat{f}[a_1, \ldots, a_n]$, and finally uses the answer to this query as the value of $f(t_1, \ldots, t_n)$.

By assigning templates to external function symbols, rather than to their occurrences in a rule, we incorporate into our framework the "Lipari convention" of

[Blass and Gurevich 2006-7, Part II, Section 4.3]. This means that, if an external function symbol has several occurrences in an ASM program and if its arguments have the same values at these occurrences, then only a single query will be issued in any one step as a result of all of these occurrences. See Sections 4.3–4.6 of [Blass and Gurevich 2006-7, Part II] for a discussion of alternative conventions, and see [Blass and Gurevich 2006-7, Part III, Section 7] for additional information comparing these conventions.

Definition 8.7. An *interactive, small-step, ASM program* Π consists of

—a finite vocabulary Υ,

—a finite set Λ of labels,

—a finite external vocabulary E,

—a rule R, using the vocabularies Υ and E, the *underlying rule* of Π,

—a template assignment with respect to E and Λ.

Convention 8.8. We use the following abbreviations:

$(s \prec t)$	for	$\neg(t \preceq s)$,
$(s \approx t)$	for	$(s \preceq t) \curlywedge (t \preceq s)$,
$(s \succeq t)$	for	$(t \preceq s)$, and
$(s \succ t)$	for	$(t \prec s)$

We abbreviate the empty block `do in parallel enddo` as `skip`. We may omit parentheses when no confusion results. □

This completes the syntax of ASMs; we turn next to the semantics, as presented in [Blass et al. 2007, Part II, Section 3]. We treat terms, guards, and rules in turn. Their semantics are defined in the presence of a state X, a template assignment, and a history ξ.

The semantics of terms specifies, by induction on terms t, the queries that are caused by ξ under the associated causality relation $\vdash^t_X$ and sometimes also a value $\mathrm{Val}(t, X, \xi) \in X$. In the case of query-terms, the semantics may specify also a query called the query-value $\text{q-Val}(t, X, \xi)$. Evaluation of a query-term t should first issue the query $\text{q-Val}(t, X, \xi)$; the reply, if any, to this query is the actual value $\mathrm{Val}(t, X, \xi)$ of t.

It may be useful to observe that, as pointed out in Lemmas 3.4 and 3.10 of [Blass et al. 2007, Part II], a term or guard causes at least one query if and only if it has no value. There is, however, no analog of this simple observation in the case of rules; a rule can cause a query even if it is final or produces updates or both.

Definition 8.9 *Semantics of Terms.* Let t be the term $f(t_1, \ldots, t_n)$.

(1) If $\mathrm{Val}(t_i, X, \xi)$ is undefined for at least one i, then $\mathrm{Val}(t, X, \xi)$ is also undefined, and $\xi \vdash^t_X q$ if and only if $\xi \vdash^{t_i}_X q$ for at least one i. If $f \in \mathrm{E}$ then $\text{q-Val}(t, X, \xi)$ is also undefined.

(2) If, for each i, $\mathrm{Val}(t_i, X, \xi) = a_i$ and if $f \in \Upsilon$, then $\mathrm{Val}(t, X, \xi) = f_X(a_1, \ldots, a_n)$, and no query q is caused by ξ.

(3) If, for each i, $\mathrm{Val}(t_i, X, \xi) = a_i$, and if $f \in \mathrm{E}$, then $\text{q-Val}(t, X, \xi)$ is the query $\hat{f}[a_1, \ldots, a_n]$.
—If $\text{q-Val}(t, X, \xi) = q \in \mathrm{Dom}(\dot{\xi})$, then $\mathrm{Val}(t, X, \xi) = \dot{\xi}(q)$, and no query is caused by ξ.
—If $\text{q-Val}(t, X, \xi) = q \notin \mathrm{Dom}(\dot{\xi})$, then $\mathrm{Val}(t, X, \xi)$ is undefined, and q is the unique query such that $\xi \vdash^t_X q$.

□

The semantics of guards, unlike that of terms, depends not only on the answer function but also on the preorder in the history. Another difference from the term case is that the values of guards, when defined, are always Boolean values.

Definition 8.10 *Semantics of guards.* Let φ be a guard and ξ a history in an Υ-structure X.

(1) If φ is a Boolean term, then its value (if any) and causality relation are already given by Definition 8.9.

(2) If φ is $(s \preceq t)$ and if both s and t have values with respect to ξ, then $\mathrm{Val}(\varphi, X, \xi) = \texttt{true}$ if, for every initial segment $\eta \trianglelefteq \xi$ such that $\mathrm{Val}(t, X, \eta)$ is defined, $\mathrm{Val}(s, X, \eta)$ is also defined. Otherwise, $\mathrm{Val}(\varphi, X, \xi) = \texttt{false}$. Also declare that $\xi \vdash^\varphi_X q$ for no q.

(3) If φ is $(s \preceq t)$ and if s has a value with respect to ξ but t does not, then define $\mathrm{Val}(\varphi, X, \xi)$ to be $\texttt{true}$; again declare that $\xi \vdash^\varphi_X q$ for no q.

(4) If φ is $(s \preceq t)$ and if t has a value with respect to ξ but s does not, then define $\mathrm{Val}(\varphi, X, \xi)$ to be $\texttt{false}$; again declare that $\xi \vdash^\varphi_X q$ for no q.

(5) If φ is $(s \preceq t)$ and if neither s nor t has a value with respect to ξ, then $\mathrm{Val}(\varphi, X, \xi)$ is undefined, and $\xi \vdash^\varphi_X q$ if and only if $\xi \vdash^s_X q$ or $\xi \vdash^t_X q$.

(6) If φ is $\psi_0 \curlywedge \psi_1$ and both ψ_i have value $\texttt{true}$, then $\mathrm{Val}(\varphi, X, \xi) = \texttt{true}$ and no query is produced.

(7) If φ is $\psi_0 \curlywedge \psi_1$ and at least one ψ_i has value $\texttt{false}$, then $\mathrm{Val}(\varphi, X, \xi) = \texttt{false}$ and no query is produced.

(8) If φ is $\psi_0 \curlywedge \psi_1$ and one ψ_i has value $\texttt{true}$ while the other, ψ_{1-i}, has no value, then $\mathrm{Val}(\varphi, X, \xi)$ is undefined, and $\xi \vdash^\varphi_X q$ if and only if $\xi \vdash^{\psi_{1-i}}_X q$.

(9) If φ is $\psi_0 \curlywedge \psi_1$ and neither ψ_i has a value, then $\mathrm{Val}(\varphi, X, \xi)$ is undefined, and $\xi \vdash^\varphi_X q$ if and only if $\xi \vdash^{\psi_i}_X q$ for some i.

(10) The preceding four clauses apply with $\curlyvee$ in place of $\curlywedge$ and $\texttt{true}$ and $\texttt{false}$ interchanged.

(11) If φ is $\neg\psi$ and ψ has a value, then $\mathrm{Val}(\varphi, X, \xi) = \neg\mathrm{Val}(\psi, X, \xi)$ and no query is produced.

(12) If φ is $\neg\psi$ and ψ has no value then $\mathrm{Val}(\varphi, X, \xi)$ is undefined and $\xi \vdash^\varphi_X q$ if and only if $\xi \vdash^\psi_X q$.

□

The semantics of a rule, for an Υ-structure X, an appropriate template assignment, and a history ξ, consists of a *causality relation*, declarations of whether ξ is *final* and whether it *succeeds* or *fails*, and a set of *updates*.

Definition 8.11 *Semantics of Rules.* Let R be a rule and ξ a history for the Υ-structure X. In the following clauses, whenever we say that a history succeeds or that it fails, we implicitly also declare it to be final; contrapositively, when we say that a history is not final, we implicitly also assert that it neither succeeds nor fails.

(1) If R is an update rule $f(t_1, \ldots, t_n) := t_0$ and if all of the terms t_i have values $\mathrm{Val}(t_i, X, \xi) = a_i$, then ξ succeeds for R, and it produces the update set $\{\langle f, \langle a_1, \ldots, a_n\rangle, a_0\rangle\}$ and no queries.
(2) If R is an update rule $f(t_1, \ldots, t_n) := t_0$ and if some t_i has no value, then ξ is not final for R, it produces the empty update set, and $\xi \vdash^R_X q$ if and only if $\xi \vdash^{t_i}_X q$ for some i.
(3) If R is `issue` $f(t_1, \ldots, t_n)$ and if all the t_i have values $\mathrm{Val}(t_i, X, \xi) = a_i$, then ξ succeeds for R, it produces the empty update set, and $\xi \vdash^R_X q$ for the single query $q = \hat{f}[a_1, \ldots, a_n]$ provided $q \notin \mathrm{Dom}(\dot{\xi})$; if $q \in \mathrm{Dom}(\dot{\xi})$ then no query is produced.
(4) If R is `issue` $f(t_1, \ldots, t_n)$ and if some t_i has no value, then ξ is not final for R, it produces the empty update set, and $\xi \vdash^R_X q$ if and only if $\xi \vdash^{t_i}_X q$ for some i.
(5) If R is `fail`, then ξ fails for R; it produces the empty update set and no queries.
(6) If R is a conditional rule `if` φ `then` R_0 `else` R_1 `endif` and if φ has no value, then ξ is not final for R, and it produces the empty update set. $\xi \vdash^R_X q$ if and only if $\xi \vdash^{\varphi}_X q$.
(7) If R is a conditional rule `if` φ `then` R_0 `else` R_1 `endif` and if φ has value `true` (resp. `false`), then finality, success, failure, updates, and queries are the same for R as for R_0 (resp. R_1).
(8) If R is a parallel combination `do in parallel` $R_1, \ldots, R_k$ `enddo` then:
   —$\xi \vdash^R_X q$ if and only if $\xi \vdash^{R_i}_X q$ for some i.
   —The update set for R is the union of the update sets for all the components R_i. If this set contains two distinct updates at the same location, then we say that a *clash* occurs (for R, X, and ξ).
   —ξ is final for R if and only if it is final for all the R_i.
   —ξ succeeds for R if and only if it succeeds for all the R_i and no clash occurs.
   —ξ fails for R if and only if it is final for R and either it fails for some R_i or a clash occurs.

□

Definition 8.12. Fix a rule R endowed with a template assignment, and let X be an Υ-structure and ξ be a history for X. If ξ is successful and final for R over X, then the *successor* $\tau(X, \xi)$ of X with respect to R and ξ is defined from the update set $\Delta^+(X, \xi)$ as in the Step Postulate, Part C.

It is easy to check (see [Blass et al. 2007, Part II, Lemma 3.18]) that τ is well-defined; Δ^+ will not prescribe two contradictory updates of the same location under a successful, final history.

Definition 8.13. An *interactive, small-step, ASM* consists of

—an ASM program Π in some vocabulary Υ,

—a nonempty set $\mathcal{S}$ of Υ-structures called *states* of the ASM, and

—a nonempty set $\mathcal{I} \subseteq \mathcal{S}$ of *initial states*,

subject to the requirements that $\mathcal{S}$ and $\mathcal{I}$ are closed under isomorphism and that $\mathcal{S}$ is closed under transitions in the following sense. If $X \in \mathcal{S}$ and if ξ is a successful, final history for Π in X, then the successor $\tau(X, \xi)$ of X with respect to Π and ξ is also in $\mathcal{S}$. □

It is shown in [Blass et al. 2007, Part II] that ASMs are algorithms, in the sense defined above by the postulates, and that, conversely, all algorithms are behaviorally equivalent to ASMs.

Remark 8.14. Because ASMs are algorithms, we can apply to ASMs all the concepts introduced earlier for algorithms, for example $\text{Issued}_X(\xi)$. Furthermore, when the sets $\mathcal{S}$ of states, $\mathcal{I}$ of initial states, Λ of labels, and E of external function symbols and a template assignment are specified, then an ASM rule describes an algorithm, and so we may apply algorithm concepts directly to rules. □

9. ASMS WITH PERSISTENT QUERIES

In this section, we indicate how to modify the syntax and semantics of ASMs, as presented in Section 8, to include our proposed method of handling persistent queries. As before, we use the heading "Modification" for the changes needed in order that our new ASM programs make sense, and we use "Intention" for constraints ensuring that the sense is what we intend.

Our first task is to understand, in a way that fits the general notions of algorithms and ASMs, the external function calls accompanied by reply locations. We can fit this syntactic construct

$$g(u_1, \ldots, u_m)[=: f(t_1, \ldots, t_n)]$$

into the ASM framework by treating it as a new external function symbol with all of $u_1, \ldots, u_m, t_1, \ldots, t_n$ as arguments. That is, we require the availability of a new $(m + n)$-ary function symbol, which we denote by $g[=: f]$, and we treat $g(\mathbf{u})[=: f(\mathbf{t})]$ as syntactic sugar for $g[=: f](\mathbf{u}, \mathbf{t})$. The following modification and definition formalize this convention.

Modification 3. For certain pairs g, f, where g is an external function symbol and f a reply-available function symbol, an external function symbol $g[=: f]$ is designated, with arity equal to the sum of the arities of g and f.

Definition 9.1. When $g[=: f]$ is defined, we say that f is *reply-available for* g. In this case, if g is m-ary and f is n-ary, then $g(u_1, \ldots, u_m)[=: f(t_1, \ldots, t_n)]$ means $g[=: f](u_1, \ldots, u_m, t_1, \ldots, t_n)$.

At this stage, we have ensured that ASM programs written with the $g(\mathbf{u})[=: f(\mathbf{t})]$ notation are syntactically correct, provided f is reply-available for g. As a first step toward semantic correctness, we want them to issue the right queries.

Intention 9.2. When $g[=: f]$ is defined, the associated template is

$$\widehat{g[=: f]} = \langle \hat{g}, \mathtt{rl}, f, \#(m+1), \ldots, \#(m+n)\rangle,$$

where g is m-ary and f is n-ary.

Remark 9.3. We have, once again, taken some liberties with the bracketing. Without liberties, we would have $\hat{g}^\frown\langle \mathtt{rl}, f, \#(m+1), \ldots, \#(m+n)\rangle$, where $^\frown$ denotes concatenation of sequences. It may also be worth noting that $\hat{g}$ is the same as $\hat{g}[\#1, \ldots, \#m]$. □

Intention 9.4. The only external function symbols whose templates contain `rl` are those of the form $g[=: f]$.

Remark 9.5. The preceding two "Intentions" imply that no template contains more than one occurrence of `rl`. □

Remark 9.6. It is possible for an ASM program to prescribe two different reply locations for what turns out to be the same query. For example, we might have both $g(\mathbf{u})[=: f(\mathbf{t})]$ and $g(\mathbf{u}')[=: f'(\mathbf{t}')]$ where, in some (or even every) state $\mathbf{u}$ and $\mathbf{u}'$ have the same values $\mathbf{a}$ but $f \neq f'$. Then the queries resulting from these two occurrences are different, but they differ only after the `rl`. So our redefinition of Issued says that only a single query is issued, namely $\hat{g}[\mathbf{a}]$. (See Remark 7.4 concerning the reasonableness of this convention.) According to Intention 7.5 above, a reply to this single query is to be written into both of the reply locations. □

Remark 9.7. We briefly indicate an alternative approach that does not require the environment to reinterpret $\langle q, \mathtt{rl}, l\rangle$ as the query q (and does not require us to redefine Issued). In this approach, $g(\mathbf{u})[=: f(\mathbf{t})]$ should produce two queries, namely the query q that would be issued by $g(\mathbf{u})$ alone and the additional query $\langle q, \mathtt{rl}, l\rangle$ giving its reply location l. The environment treats q like any other query, answering it (if possible) in the usual way. It treats $\langle q, \mathtt{rl}, l\rangle$ as a message, answering it with an automatic, immediate "OK," but remembering it so that it knows where to write a reply to q later.

This approach requires a modification to [Blass et al. 2007] to allow two queries to be caused at a single point in an ASM program. The template assignment should now be multivalued, assigning to $g[=: f]$ both the template $\langle \hat{g}, \mathtt{rl}, f, \#(m+1), \ldots, \#(m+n)\rangle$ used above and the template $\hat{g}$. A secondary modification is to allow an m-ary template $\hat{g}$ to be used for an $(m+n)$-ary function symbol $g[=: f]$. □

10. IMPATIENCE

We call an algorithm *patient* if it never finishes a step until the environment has answered all queries from that step. (It patiently waits for answers to all its queries.) Formally, this means that all final, attainable histories are complete. Otherwise, we call the algorithm *impatient*.

A query is said to be *blocking* if, once it is issued, the algorithm's step cannot end without a reply to this query. Thus, an algorithm is patient if and only if all its queries are blocking.

In this paper, we are concerned with non-blocking queries, and specifically with the possibility that the reply to such a query may arrive and be used by the algorithm in a later step than the one that produced the query. The present section describes where, in an ASM program, non-blocking queries can originate. Of course, we must first say precisely what it means for a query to originate in a particular part of a rule — or of a term, or of a guard.

We present the material in this section in the context of the traditional ASM syntax and semantics described in Section 8. It applies equally, however, to the modified syntax and semantics that was described in Section 3 and formalized in Section 9. The changes we introduced do not affect the proofs in the present section.

The discussion will be simplified by the following definitions and convention.

Definition 10.1. Let S be a term or a guard or a rule, let X be a state, and let ξ be a history for X. We define

$$\mathrm{Issued}^S_X(\xi) = \{q : (\exists \eta \trianglelefteq \xi)\, \eta \vdash^S_X q\}$$
$$\mathrm{Pending}^S_X(\xi) = \mathrm{Issued}^S_X(\xi) - \mathrm{Dom}(\dot{\xi}). \quad \square$$

Note that, in the case of a rule, this definition agrees with Definition 6.5 for algorithms. (See Remark 8.14 concerning treating rules as algorithms.) We are just extending the "Issued" and "Pending" notation to apply also to terms and guards (and adding the superscript S, which was unnecessary earlier because the role of S was played there by a fixed algorithm). The next definition also extends to terms and guards terminology already available for algorithms and therefore for rules.

Definition 10.2. Let S be a term or a guard, let X be a state, and let ξ be a history for X. We say that the history ξ is *final* for S in X if $\mathrm{Val}(S, X, \xi)$ is defined.

Convention 10.3. When we speak of syntactic parts of a term, guard, or rule, we mean occurrences of those syntactic parts. Thus, for example, "subrule" really means "occurrence of subrule."

We now define the origins of a query caused by a term or guard or rule. The definition involves going systematically through the definitions of the semantics of term, guards, and rules (Definitions 8.9, 8.10, and 8.11), and checking all the clauses where a query is caused. For the reader's convenience, we append to some clauses of the following definition some additional information, in brackets, about the circumstances in which those clauses can apply. These bracketed comments can easily be verified by inspection of Definitions 8.9, 8.10, and 8.11.

Definition 10.4. Let X be a state, and ξ a history for it, and q a potential query.

—If t is a term $f(t_1, \ldots, t_n)$, if $\mathrm{Val}(t_i, X, \xi)$ is undefined for at least one i, and if $\xi \vdash^t_X q$, then the origins of q in t are the origins of q in all those t_i for which $\xi \vdash^{t_i}_X q$.

—If t is a term $f(t_1, \ldots, t_n)$, if $\mathrm{Val}(t_i, X, \xi)$ is defined for all i, and if $\xi \vdash^t_X q$, then q has exactly one origin in t, namely t itself. [Here f is an external function symbol and q is the q-value of t.]

—If φ is $(s \preceq t)$ and $\xi \vdash_X^\varphi q$, then the origins of q in φ are its origins in s (if any, i.e., if $\xi \vdash_X^s q$) and its origins in t (if any). [According to the semantics of guards, if either s or t has a value, then $(s \preceq t)$ issues no queries. So the present clause applies only when ξ is not final for either of these terms.]

—If φ is $\psi_0 \curlywedge \psi_1$ or $\psi_0 \curlyvee \psi_1$ and $\xi \vdash_X^\varphi q$, then the origins of q in φ are its origins in ψ_0 (if any) and its origins in ψ_1 (if any). [At most one of ψ_0 and ψ_1 has a value under ξ, and if one does then that value is `true` in the case of $\curlywedge$ and `false` in the case of $\curlyvee$.]

—If φ is $\neg\psi$ and $\xi \vdash_X^\varphi q$, then the origins of q in φ are the same as in ψ.

—If R is an update rule $f(t_1, \ldots, t_n) := t_0$ and $\xi \vdash_X^R q$, then the origins of q in R are the origins of q in all those t_i for which $\xi \vdash_X^{t_i} q$.

—If R is `issue` $f(t_1, \ldots, t_n)$, if all the t_i have values $\mathrm{Val}(t_i, X, \xi) = a_i$, and if $\xi \vdash_X^R q$, then q has exactly one origin in R, namely $f(t_1, \ldots, t_n)$. [Here f is an external function symbol and q is the q-value of $f(t_1, \ldots, t_n)$.]

—If R is `issue` $f(t_1, \ldots, t_n)$, if some t_i has no value, and if $\xi \vdash_X^R q$, then the origins of q in R are the origins of q in all those t_i for which $\xi \vdash_X^{t_i} q$.

—If R is a conditional rule `if` φ `then` R_0 `else` R_1 `endif`, if φ has no value under ξ, and if $\xi \vdash_X^R q$, then the origins of q in R are the origins of q in φ.

—If R is a conditional rule `if` φ `then` R_0 `else` R_1 `endif`, if φ has value `true` (resp. `false`), and if $\xi \vdash_X^R q$, then the origins of q in R are its origins in R_0 (resp. R_1).

—If R is a parallel combination `do in parallel` $R_1, \ldots, R_k$ `enddo` and if $\xi \vdash_X^R q$, then the origins of q in R are its origins in all those R_i for which $\xi \vdash_X^{R_i} q$.

In the preceding definition, X and ξ were fixed and were therefore not mentioned in the "origin" terminology. When necessary, we make them explicit by a phrase like "origin of q in R with respect to X and ξ."

LEMMA 10.5. *Let S be a term or guard or rule, let X be a state, let ξ be a history for X, and let q be a potential query in X. Then $\xi \vdash_X^S q$ if and only if q has at least one origin in S with respect to X and ξ. Any origin of q is a query-term t, a subterm of S, with q-Val$(t, X, \xi) = q$. Furthermore, this t is also the (unique) origin of q in t; in particular, $\xi \vdash_X^t q$.*

PROOF. Proceed by induction, first on terms, then on guards, and finally on rules. In every case, the proof is just a comparison of the definition of "origin" with the parts of Definitions 8.9, 8.10, and 8.11 that describe the causality relation. □

After these preliminaries, we can look in detail at impatience, the phenomenon of an ASM's step ending even though some of its queries have not been answered. In more detail, the phenomenon involves five entities:

—an ASM program Π,

—a state X of Π,

—a history ξ that is final for X with respect to Π (so the step ends),

—a query $q \in \mathrm{Pending}_X^\Pi(\xi)$ (so q has been issued but not answered during this step), and

—an initial segment $\eta \trianglelefteq \xi$ such that $\eta \vdash_X^\Pi q$.

In connection with the last of these items, η, recall that for q to be issued during a step where the history is ξ it must be caused by some initial segment of ξ, though not necessarily by ξ itself.

We have simplified the notation by using the same symbol Π for an ASM program and for its underlying rule, even though the program also includes additional material, particularly the template assignment. This additional material will remain fixed, so our abuse of notation will not cause confusion.

The following proposition describes the possible origins of queries that remain unanswered at the end of a step. Notice that, when the S in the proposition is a rule Π, then the hypotheses of the proposition describe the five items listed above.

PROPOSITION 10.6. *Let S be a term or guard or rule. Let X be a state and let $\eta \trianglelefteq \xi$ be two histories for X, such that ξ is final for X with respect to S. Let q be a query such that $\eta \vdash_X^S q$ but $q \notin Dom(\xi)$. Then all origins of q in S with respect to X and η are of one of the following sorts:*

—*query-subterms of s or t in a timing guard $s \preceq t$ within S,*
—*query-subterms of ψ_0 or ψ_1 in a Kleene-conjunction $\psi_0 \curlywedge \psi_1$ or Kleene-disjunction $\psi_0 \curlyvee \psi_1$ within S,*
—*arguments t of issue-rules $\texttt{issue}(t)$ within S.*

PROOF. Assume that S, X, η, ξ, and q are as in the hypothesis of the proposition and that o is an origin of q in S with respect to X and η. Assume also, as an induction hypothesis, that the proposition becomes true if S is replaced by any proper subterm, subguard, or subrule (while X, η, ξ, q, and o are unchanged.)

To save a little writing later, observe that the hypothesis that $\eta \vdash_X^S q$ is redundant, because, according to Lemma 10.5, if it didn't hold then there would be no origin of q in S with respect to X and η, and so the conclusion of the proposition would hold vacuously.

The proof will repeatedly use the observation that, if the conclusion of the proposition holds when S is replaced by some subterm, subguard, or subrule S' of S, then it also holds for S itself. The reason is that the conclusion refers to S only in the context of saying that some guard or rule occurs within S. If we find the desired guard or rule within S' then we certainly have it within S.

The fact that $\eta \vdash_X^S q$ must arise from one of the clauses of Definition 8.9, 8.10, or 8.11, with η in place of the ξ in the definition. And this clause cannot be one of the many clauses where the definition says that no query is caused, i.e., clause 2 and the first case in clause 3 of Definition 8.9; clauses 2, 3, 4, 6, 7, and 11 as well as the part of clause 10 analogous to clause 7 in Definition 8.10; and clauses 1 and 5 of Definition 8.11. We examine the remaining possibilities in turn, labeling them according to the clause in Definition 8.9, 8.10, or 8.11 that provides $\eta \vdash_X^S q$.

8.9-1: S is a term $f(t_1, \ldots, t_n)$ and, for at least one i, $\mathrm{Val}(t_i, X, \eta)$ is undefined. According to Definition 10.4, o is an origin of q in some t_i (with respect to X and η). For such an i, ξ will be final with respect to t_i, i.e., $\mathrm{Val}(t_i, X, \xi)$ will be defined, because otherwise, the same clause of Definition 8.9 (now applied to ξ rather than η) would contradict the assumption that ξ is final for S. Thus, the hypotheses of the proposition are satisfied with t_i in place of S. By induction hypothesis, the conclusions of the proposition hold for t_i, and, as observed above, it immediately

follows that they also hold for S. (Though it isn't needed for the proof, it may help the reader if we point out that this case cannot actually occur. Indeed, by what we have just proved, we would have a guard (involving $\prec$ or $\curlywedge$ or Υ) or an issue-rule within a term, and this cannot happen in the ASM syntax.)

8.9-3, second part: S is a term $f(t_1, \ldots, t_n)$ where f is an external function symbol; each t_i has a value $\mathrm{Val}(t_i, X, \eta) = a_i$; and

$$q = \text{q-Val}(S, X, \eta) = \hat{f}[a_1, \ldots, a_n].$$

(This clause in Definition 8.9 also says that $q \notin \mathrm{Dom}(\dot{\eta})$, but this is immediate from the assumptions that $q \notin \mathrm{Dom}(\dot{\xi})$ and $\eta \trianglelefteq \xi$.) Lemma 3.6 of [Blass et al. 2007, Part II] gives us that, when we pass from η to its extension ξ, the values of the t_i's do not change. So we have $\mathrm{Val}(t_i, X, \xi) = a_i$ and therefore (by the same clause of Definition 8.9)

$$\text{q-Val}(S, X, \xi) = \hat{f}[a_1, \ldots, a_n] = q.$$

But then, since $q \notin \mathrm{Dom}(\dot{\xi})$, the same clause tells us that ξ is not final for S. This contradicts the hypothesis of the proposition, so this case simply cannot arise.

8.10-1: Here S is guard that is a Boolean term, so this case is included in the cases already treated where S is a term.

8.10-5: Here S is a timing guard ($s \preceq t$) (and neither of the terms s, t has a value with respect to η). By Lemma 10.5, o is a subterm of this timing guard, and so we have the first of the three alternatives in the conclusion of the proposition.

8.10-8 or 9: Here S is a Kleene conjunction, and so its subterm o satisfies the second alternative in the proposition.

8.10-10: Here S is a Kleene disjunction, and so we again get the second alternative of the proposition.

8.10-12: Here S is $\neg\psi$. By definition, origins in S are the same as in ψ. Also, by definition, since ξ is final for S, it is also final for ψ. Thus, the induction hypothesis applies and tells us that the conclusion of the proposition holds with ψ in place of S. But then it also holds for S.

8.11-2: S is an update rule $f(t_1, \ldots, t_n) := t_0$ and not all t_i have values with respect to η. The argument here is essentially the same as for case 8.9-1 above. o is an origin of q in some t_i, and ξ must be final for t_i as otherwise it would not be final for S. By induction hypothesis, the conclusion of the proposition holds with t_i in place of S, and therefore it also holds for S.

8.11-3: Here S is an issue-rule and, by definition of "origin," o is its argument. So we have the third alternative in the proposition.

8.11-4: The argument here is again essentially the same as for cases 8.9-1 and 8.11-2; we spare the reader (and ourselves) a third occurrence of this same argument.

8.11-6: Here S is a conditional rule whose guard φ has no value with respect to η. By definition, o is an origin of q in φ with respect to η. Furthermore, ξ must be final for φ, because otherwise it could not be final for S. So the induction hypothesis applies and we get the conclusion of the proposition with φ in place of S, and therefore also for S.

8.11-7: S is a conditional rule `if` φ `then` R_0 `else` R_1 `endif` and φ has a value with respect to η. We assume $\mathrm{Val}(\varphi, X, \eta) = \texttt{true}$; the case of `false` is the same with R_0 and R_1 interchanged. By definition, the origins of q in S are the same as in R_0. Also, by Lemma 3.12 of [Blass et al. 2007, Part II], $\mathrm{Val}(\varphi, X, \xi) = \texttt{true}$, so the finality of ξ for S implies that ξ is also final for R_0. So the induction hypothesis applies and gives us the conclusion of the proposition with R_0 in place of S. As usual, the conclusion for S follows.

8.11-8: S is a parallel combination with components R_i. The definition of "origin" says that o is an origin of q in at least one of the R_i. And ξ must be final for that R_i because otherwise it could not be final for S. So the induction hypothesis gives us the conclusion of the proposition with R_i in place of S, and the conclusion for S follows. □

Remark 10.7. In view of Proposition 10.6, we can limit the use of the new syntax $g(\mathbf{u})[=: f(\mathbf{t})]$ to the places described in the proposition, namely subterms of timing guards, of Kleene conjunctions, and of Kleene disjunctions, and arguments of issue-rules. External function symbols occurring anywhere else in an ASM program produce blocking queries, so there is no need to provide locations for late replies. And if a reply-location is provided for a blocking query, with the intention of having an on-time reply recorded there, then the program can easily be altered so that the ASM reads the reply in its history and writes it into the desired location. □

Example 10.8. Here are some trivial examples showing that all the alternatives in the conclusion of Proposition 10.6 can occur (with S being a rule). Assume that the vocabulary has three external, nullary function symbols a, b, c and that the templates $\hat{a}, \hat{b}, \hat{c}$ assigned to them are distinct.

For the first alternative in Proposition 10.6, consider the rule

$$\texttt{if } a \prec b \texttt{ then } x := 1 \texttt{ else } x := 2 \texttt{ endif}.$$

The empty history causes both $\hat{a}$ and $\hat{b}$. Any history ξ with domain $\{\hat{a}\}$ is final and has $\hat{b}$ pending. This $\hat{b}$ has exactly one origin, namely the unique b in the rule. (The updates of x could be replaced by certain other rules, for example `skip`, without affecting the idea.) The same program also serves as an example if the reply for b arrives before that for a; then the history with only the reply for b is final, a is pending, and its only origin is in the timing guard $a \prec b$.

For the second alternative, consider

$$\texttt{if } (a = b) \curlywedge (a = c) \texttt{ then } x := 1 \texttt{ else } x := 2 \texttt{ endif}.$$

The empty history causes all three of $\hat{a}, \hat{b}, \hat{c}$. Any history ξ with domain $\{\hat{a}, \hat{b}\}$ and with $\xi(\hat{a}) \neq \xi(\hat{b})$ is final and has $\hat{c}$ pending. The only origin of $\hat{c}$ in this rule is the unique occurrence of c.

There is an analogous example with $\curlyvee$ in place of $\curlywedge$. Just use a ξ that gives the same reply to the two queries $\hat{a}$ and $\hat{b}$.

Finally, for the third alternative, just use the rule `issue`(a). The empty history causes $\hat{a}$ and is final, with $\hat{a}$ pending.

Remark 10.9. We take this opportunity to clarify Remark 3.17 of [Blass et al. 2007, Part II], which begins: "Issue rules are the only way an ASM can issue a query

without necessarily waiting for an answer." This appears to deny the possibility of the first two alternatives in Proposition 10.6. Indeed, if "waiting for an answer" means "waiting until an answer is received," then the examples just given show that this is wrong. It becomes correct, however, if "waiting for an answer" means "waiting at least for a moment," i.e., not finishing the step immediately. Note that, in the parts of Example 10.8 that don't use `issue`, the pending query is caused not by the final history but by a proper initial segment. (In the notation of Proposition 10.6, $\eta \neq \xi$.) In this sense, the ASM does wait after issuing the query and before finishing the step.

Another description of what happens in these examples is that the unanswered query q is issued by the final history ξ (in the sense of being in Issued(ξ)) but it is not caused by the final history ($\xi \not\vdash q$). In the second sentence of Remark 3.17, we used the phrase "a history causes a rule to issue a query," which is ambiguous in view of the difference between causing and issuing. It should be interpreted as causing, not merely issuing. □

Conclusion and Future Prospects

Previous work on behavioral computation theory dealt with what happens during any step of an algorithm's execution; interventions by the environment between steps were not studied because they are entirely arbitrary. In the present paper, we have assumed that the environment is somewhat cooperative in its inter-step actions, namely that it will put late replies into locations as requested by the algorithm. We have extended both the axiomatic framework and the ASM model from [Blass et al. 2007] to handle late replies in a simple and natural way. Fortunately, the necessary modifications of [Blass et al. 2007] are quite minor. It is reasonable to expect that, once the theory in [Blass et al. 2007] is extended beyond the case of small-step algorithms, to cover large-scale parallelism and distributed computing, the methods of this paper will be applicable in those contexts also.

Furthermore, when one deals with parallel or distributed algorithms, the methods developed in [Blass et al. 2007] and in the present paper would be applicable not only to the interaction between the whole algorithm and its external environment but also to the interaction between the individual processes or agents within the algorithm.

REFERENCES

Henry Baker and Carl Hewitt, The Incremental Garbage Collection of Processes. *Proc. 1977 Symp. on Artificial Intelligence and Programming Languages*, ACM SIGPLAN Notices 12, 1997.

Danièle Beauquier and Anatol Slissenko, Periodicity Based Decidable Classes in a First Order Timed Logic *Ann. Pure Appl. Logic* 139 (2006), 43–73

Andreas Blass and Yuri Gurevich, Abstract state machines capture parallel algorithms. *ACM Trans. Computational Logic,* 4:4 (2003), 578–651. *Correction and extension*, *ibid.* 9:3 2008, article 19.

Andreas Blass and Yuri Gurevich, Ordinary interactive small-step algorithms. *A*CM Trans. Computational Logic. Part I, 7:2 (2006), 363–419; Part II, 8:3 (2007), article 15; Part III, 8:3 (2007), article 16.

Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman, Interactive small-step algorithms. *Logical Methods in Computer Science.* Part I: Axiomatization, Vol. 3:4

(2007), paper 3. Part II: Abstract state machines and the characterization theorem, Vol. 3:4 (2007), paper 4.

Udi Boker and Nachum Dershowitz, Abstract effective models. *Electronic Notes in Theoretical Computer Science* 135 (2005), 15–23.

Udi Boker and Nachum Dershowitz, The Church-Turing thesis over arbitrary domains. *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, ed. A. Avron, N. Dershowitz, and A. Rabinovich. Springer-Verlag, Lecture Notes in Computer Science 4800, (2008), 199–229.

Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris, Event-driven programming for robust software. *Proc. 10th ACM SIGOPS European Workshop*, (Sept. 2002), 186–189.

Nachum Dershowitz and Yuri Gurevich, A natural axiomatization of computability and proof of Church's Thesis. *Bull. Symbolic Logic* 14 (2008), 299–350.

EasyChair. http://easychair.org/. (Viewed on December 20, 2009.)

Daniel Friedman and David Wise, CONS should not evaluate its arguments. In *Automata, Languages and Programming*, ed. S. Michaelson and R. Milner. Edinburgh University Press 1976, 257–284.

Andreas Glausch and Wolfgang Reisig, A semantic characterization of unbounded-nondeterministic ASMs. *Rigorous Methods for Software Construction and Analysis: Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday.* Springer-Verlag Lecture Notes in Computer Science 4624 (2008) 46–60.

Andreas Glausch and Wolfgang Reisig, An ASM characterization of a class of distributed algorithms. *Proc. Dagstuhl Seminar on Rigorous Methods in Software Construction and Analysis.* Springer Lecture Notes in Computer Science, 5115, 2009.

Yuri Gurevich, Evolving algebras 1993: Lipari guide. *Specification and Validation Methods,* ed. E. Börger. Oxford University Press 1993, 9–36.

Yuri Gurevich, Sequential abstract state machines capture sequential algorithms. *ACM Trans. Computational Logic,* 1:1 (2000), 151–176.

Yuri Gurevich and James Huggins, The railroad crossing problem: an experiment with instantaneous actions and immediate reactions. *Computer Science Logic, Selected Papers from CSL'95*, ed. H. Kleine-Büning. Springer-Verlag, Lecture Notes in Computer Science 1092 (1996) 266–290.

Yuri Gurevich and Tatiana Yavorskaya, On Bounded Exploration and Bounded Nondeterminism. *Microsoft Research Tech. Report* MSR-TR-2006-07, 2006.

Jessica Millar, Finite Work. *Microsoft Research Tech Report* MSR-TR-2006-06, 2006.

Wolfgang Reisig, The computable kernel of abstract state machines. *Theoret. Comp. Sci.* 409, 126–136.

Dean Rosenzweig, Davor Runje and Neva Slani, Privacy, Abstract Encryption and Protocols: an ASM Model—Part I. *Abstract State Machines—Advances in Theory and Applications: 10th International Workshop, ASM 2003*, ed. E. Börger, A. Gargantini, and E. Riccobene. Springer-Verlag, Lecture Notes in Computer Science 2589 (2003) 372–390.

Dean Rosenzweig and Davor Runje, Some Things Algorithms Cannot Do. *Microsoft Research Tech. Report* MSR-TR-2005052, 2005.

Wikipedia, Futures and promises. (Viewed on August 1, 2008.)

Received November 2008; revised January 2010; accepted April 2010