Relativity and Abstract State Machines

Edel Sherratt

Department of Computer Science, Aberystwyth University,
Penglais, Aberystwyth SY23 3DB, Wales
eds@aber.ac.uk

Abstract. The Abstract State Machine (ASM) formalism has proved an effective and durable foundation for the formal semantics of SDL. The distributed ASMs that underpin the SDL semantics are defined in terms of agents that execute ASM programs concurrently, acting on partial views of a global state. The discrete identities of successive global states are ensured by allowing input from the external world only between steps, and by having all agents refer to an external global time. But distributed systems comprising independent agents do not have a natural global time. Nor do they have natural global states. This paper takes well-known concepts from relativity and applies them to ASMs. The spacetime in which an ASM exists and moves is defined, and some properties that must be preserved by transformations of the frame of reference of an ASM are identified. Practical implications of this approach are explored through reservation and web service examples.

Keywords: Abstract state machines, Formal semantics, Distributed system, SDL, spacetime, frame of reference.

1 Introduction

The Abstract State Machine (ASM) model of computation was introduced by Gurevich[1, 2] under the name 'evolving algebras', and was subsequently developed by Blass, Gurevich, Börger and many others [3–11].

The formal semantics of SDL was defined by Glässer, Gotzhein and Prinz in terms of distributed Abstract State Machines [12, 13]. This comprises a number of cooperating agents, each with a partial view of a global state of the ASM.

Throughout the development of Abstract State Machines, the notion of a global state and a global time has formed a recurring theme. While this does not diminish the expressive power of abstract state machines, it leads to awkward formulations of computations involving interaction and persistence.

This paper explores the consequences of abandoning the demand for a global state and global time. Drawing on ideas from relativity, it proposes independent ASMs that can observe projections of each others' states onto commonly accessible locations. This exploration reveals limitations on the kinds of observations that can be made, and therefore on the kinds of interaction that are possible between parallel ASMs.

Ø. Haugen, R. Reed, and R. Gotzhein (Eds.): SAM 2012, LNCS 7744, pp. 105–120, 2013.

A reservation service with client applications is outlined below to illustrate the kind of system that motivates the attempt to develop transformations between the location-value pairs accessible to different abstract state machines. The ideas of state and time in Abstract State Machines are then reviewed. Parallels are drawn with concepts from relativity, leading to an exploration of interaction between ASMs and of observations of ASMs by one another. This analogy facilitates reasoning about the the kinds of interaction that are possible between ASMs, and enables identification of conditions that must be fulfilled by any admissible transformation between the observations made by independent parallel ASMs. Those constraints represent a small but essential first step towards developing transformations that define communication between fully independent ASMs that do not share a global state space and that do not acknowledge a global time.

2 Abstract State Machines' Power and Limitations

Since their original introduction, abstract state machines have repeatedly been shown to be both versatile and powerful. Examples ranging from a simple clock, through Conway's game of life, ambiguous grammars, lift control, Internet telephony, database recovery and more are demonstrated in [4]. That abstract state machines capture every kind of parallel algorithm is shown in [5, 6]. Their application to generalized asynchronous communication is shown in [7], and their capacity to interact and operate in parallel is demonstrated, for example, in [11] and [14] .

Now, all these examples model processes in terms of abstract states and sequences of state transitions. Inherent in this is a notion of global state and global time. But some applications, like database clients and web services, do not directly lend themselves to a model that demands a global state and a global time.

For example, the SDL diagram in Figure 1 illustrates a ticket reservation service that is accessed by an arbitrary number of clients. Each client process progresses through its state transitions, and the reservation process does likewise. The client processes alternate, and the reservation process runs independently of the client processes. However, according to the SDL reference manual [15], all the client processes, and also the reservation process, have access to a global system clock, which supplies an absolute system time by way of the *now expression*, **now**. Furthermore, they all refer to a global state [12].

The model describes asynchronous, parallel processes that capture essential properties of the reservation system, including creation and destruction of clients and update of the state by different processes. However, the existence of global time, external to all the processes of the reservation system, means that something extraneous is being added to the system model.

A better approach would allow the system as a whole to be considered from the perspective of an individual client, or from the perspective of the reservation system, and would also allow a client to see the system state and time as the

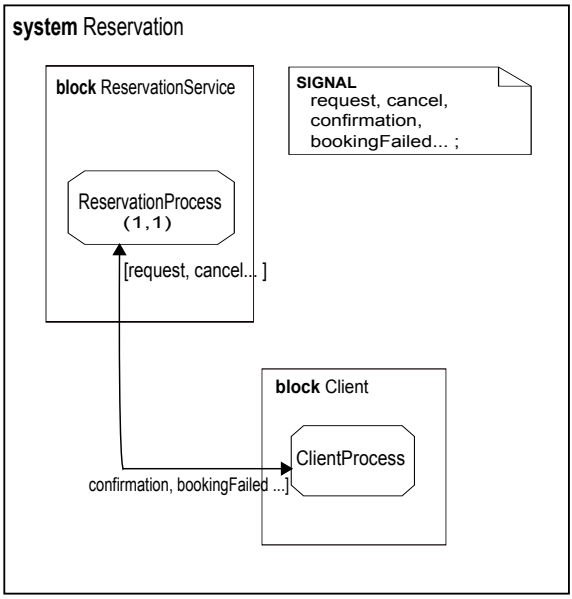

Fig. 1. Reservation system

reservation system sees it, and vice versa. In other words, it would indicate how to transform perspective between different agents.

As a preliminary to enabling such transformations, the following section will explore concepts of state and time in abstract state machines.

3 State and Time in Abstract State Machines

Notions of state and time in various formulations of abstract state machines are explored. These concepts are later compared with concepts from relativity, with a view to identifying constraints on the transformations that would allow a designer to transfer focus between interacting abstract state machines.

3.1 Basic Abstract State Machine

A basic abstract state machine is made up of abstract states with a transition rule that specifies transformations of those abstract states. The rule is often expressed as an ASM *program*.

A *state* in an ASM is defined as the association of values from an underlying base set with the symbols that form the *signature* of the ASM. This is also expressed by stating that the ASM has a vocabulary, whose symbols are interpreted over a base set, and that interpretation defines a *state*.

Some states are called *initial states*.

The symbols that comprise the signature of an ASM are function symbols, each with an arity. The interpretation of symbols is constrained so that a 0-ary function symbol is interpreted as a single element of the base set, and an n-ary function symbol is interpreted as an n-ary function over the base set. Terms are constructed from the signature in the usual way, and are interpreted recursively. To provide modularity and to enhance legibility, new symbols can be defined as abbreviations for complex terms. In SDL, these are called *derived names* [12].

The signature includes the predefined names *True*, *False* and *undef*, and three distinct values of the base set serve as interpretations for these. Certain function symbols are further classified as predicate names and domain names. A predicate name is interpreted as a function that delivers a truth value, which is the interpretation of *True* or of *False*. A domain name is a unary predicate name that classifies base set elements as being of a particular sort.

The ASM model is a dynamic model. Starting from an initial state, an abstract state machine moves through its state space by means of *transitions*, also called *moves* or *steps*. Each transition produces a new state from an existing state. The differences between old and new states are described in terms of *updates* to *locations*. Such a sequence of states is called a *run* of the abstract machine.

A function symbol f with a tuple of elements $\overline{a}$ that serves as an argument of f identifies a *location*. The term $f(\overline{a})$ identifies a location and evaluates to a value in a state. In a subsequent state, the value of that location may have changed, and $f(\overline{a})$ may evaluate to a new value. In that case, an *update* indicates what the new value will be, and is expressed using the values of terms in the current state. Updates are written as triples $(f, \overline{a}, b)$, to indicate that $f(\overline{a}) = b$ will be true in the new state. In order to limit the cardinality of the update set, [5, 6] also asserts that $f(\overline{a}) = b$ should not be true in the previous state. However trivial updates, where the new and old values of a location are the same are allowed in [11].

Updates are sometimes specified as programming-style assignments, such as: $f(\overline{a}) = b$.

Function names like *True*, *False* and *undef*, whose interpretation is the same in all the states of an abstract state machine, are called *static names*. Names like f above, that identify locations that are subject to updates, are called *dynamic* names.

The set of updates that transformes an ASM state is specified as a rule, expressed as an ASM *program*. Each state transition is realised by interpreting the ASM *program* in a given state. Interpretation of the program delivers a set of updates, which are applied simultaneously to produce the new state. The constructs used to write ASM programs vary (for example, see [3, 7, 12]), and usually allow for non-determinism in the set of updates that is generated.

Two situations can prevent further progress through the state space:

- the update set resulting from interpretation of the program in a given state is empty;
- the update set is inconsistent; that is, it includes two updates $(f, \overline{a}, b)$ and $(f, \overline{a}, c)$ where $b \neq c$.

In either case, the ASM remains in its current state, either by *stuttering* (repeatedly moving back to the current state), or by treating the current state as final and halting. Stuttering allows for external intervention to modify the state so as to enable further progress, or for a non-deterministic ASM program to yield a viable update set on re-evaluation.

That every step in a run of a basic abstract state machine yields an abstract state with a definite interpretation for each element of the ASM signature, is formulated in terms of the Abstract State and Sequential Time postulates of [5, 6]. That the work done at each step is bounded, is formulated as the Bounded Exploration Postulate [5, 6]. Together, these ensure that a basic abstract state machine has at every step a well-defined global state, and that there is a finite amount of work to be done to move from one state to the next.

This, in turn, gives rise to a notion of global time, which increases monotonically with each step.

However, forcing the reservation system and its clients, with its multiple threads of control, into the single process defined by a basic ASM leads to premature sequentialization of moves and ignores alternative scheduling strategies.

3.2 Complex Moves

A state transition can entail activation of one or more sub-machines. If this is done using a 'black-box' approach, in which the moves of the sub-machine are not made visible, the containing ASM is called a turbo-ASM [3]. Alternatively, subcomputations can be interleaved under the control of the containing process – a 'white-box' view of subcomputation[3]. In the first of these cases, moves of the containing ASM preserve the Abstract State and Sequential Time postulates by construction. In the second, a constraint is added by [3] that allows interleaved processes to act in parallel only if they all contribute to a consistent update set. This again means the the computation proceeds through a well defined sequence of abstract states.

Using this kind of approach would allow, for example, client requests to be defined as sub-machines of the reservation system, and those sub-machines could run in parallel. However, it is still not satisfactory because execution of client requests is bounded within steps of the containing ASM that models the reservation system.

3.3 Interaction with the Environment

Interaction between an abstract state machine and its environment is achieved through mutually accessible locations.

Locations that are subject to updates are named by dynamic names. Dynamic names are futher classified as *monitored*, *controlled* and *shared*. A monitored, dynamic name refers to a location that is updated by the environment and read by the ASM. A controlled, dynamic name refers to a location that is updated by the ASM, and may possibly be read by the environment. A shared, dynamic name refers to a location that is updated by the environment and by the ASM.

In order that each abstract state should form a well defined first order structure, values cannot change within the state. If the statement $f = v$ is satisfied by a given state, then $f = w$ where $w \neq v$ cannot be satisfied by the same state. So the value of a location identified by a monitored name cannot change during a state. That is, updates performed by the environment can only take place between states. This leads to an interpretation in [3] in which each transition is made by applying the updates defined by the rule of an ASM, followed by the updates made by the environment. A similar position is taken in [5, 6] where intervention of an environment only takes effect between steps.

Again, this leads to a model of computation in which the abstract state machine proceeds through a sequence of clearly defined global states.

However, this does facilitate independent modelling of the reservation system and its clients. Using the approach of [14], a client is modelled as an abstract state machine that views the reservation system as part of the environment. Interaction is modelled in terms of queries and replies. This approach is extended by [11] so that queries made at one step can be answered by replies that become available at a later step. Histories, return locations and suitable guards ensure that responses are properly associated with queries and are collected at appropriate times.

The strength of this approach is that it allows the client to view the reservation system as a kind of oracle, that grants or declines requests for reservation in an inscrutable way, and it allows the reservation system to schedule client requests in any way it sees fit. It does not, however, provide any guidance for transforming between client and reservation system views of the interaction.

3.4 Distributed Abstract State Machine

In order to model computations with multiple threads of control, the concept of *ASM agent* is introduced. Agents form part of the distributed ASMs used in the semantic definition of SDL [12, 13], and AsmL [7]. An agent actively interprets an ASM program and so drives the movement of a distributed ASM from state to state.

A distributed Abstract State Machine has a single base set. An agent is distinguished from other agents by having its own unique interpretation of a function *Self* [12] or *me* [5, 6]. In SDL, each agent has its own partial view of a current global state. This view, and, by implication, the locations accessible to the agent, are determined by the agent's program. Agents can be associated with programs using a function *program* as in [12], or a collection of agents can execute a single program, whose branches are selected based on the value of *Self*.

A distributed abstract state machine can be synchronous or asynchronous [3]. A synchronous multi-agent ASM is defined by [3] as a set of agents that execute their own ASMs in parallel, synchronized using an implicit global system clock. The signature of the synchronized multi-agent ASM is formed as the union of the signatures of its component single-agent ASMs. The global clock synchronizes the moves of the multi-agent ASM through a global state, so that all updates that can be performed at a step being performed instantaneously at that step.

Distributed asynchronous multi-agent ASMs [3] are used in the definition of SDL [12, 13] and AsmL [7].

A distributed asynchronous multi-agent ASM consists of a set of (agent, ASM) pairs, each of which executes its own ASM [3]. A run of an asynchronous multi-agent ASM is a partially ordered set of moves $(M, <)$, with the following properties [1, 3, 12]:

- each move has only finitely many predecessors;
- the moves performed by a given agent are linearly ordered by $<$;
- for every finite initial segment X of $(M. <)$, and every maximal element $m \in X$, there is a unique state $\sigma(X)$ that results from performing m in the state $\sigma(X \setminus \{m\})$.

This definition allows a great deal of freedom in constructing a run. Moves can be carried out in parallel, or by interleaving the moves of different agents, or by creating a explicit schedule. However, the last of the three conditions above means that there is a confluence of state transformation, in the sense that every linearization of every initial segment of $(M, <)$ results in the same state. This in turn means that every run of the ASM proceeds through a well defined sequence of abstract states.

This approach also facilitates communication between a reservation system and its clients. Following the approach of [7] a new kind of agent, called a *communicator*, is introduced that transfers messages between communicating applications like a client and the reservation system. This makes a clear separation between the behaviour of the network (as modelled by the communicator), and the behaviour of the reservation system and its client applications.

However, it means that the whole system, comprising the reservation system, its clients and the network are modelled as a single distributed ASM. This in turn means that the end state of every finite prefix of a partially ordered run is pre-determined – a stronger condition than the serializability condition normally required of a database schedule.

3.5 Global Time and Abstract State Machines

Single threaded basic abstract state machines imply a notion of global time, in that moves are said to come before or after each other. In a distributed ASM, moves are partially ordered, but any initial segment of a partially ordered run gives a definite state, and that state can be said to come before the states that result from extending the run. This again gives rise to an implicit notion of global time.

SDL provides an explicit definition of global time in distributed real-time ASMs using a real-valued monitored function *currentTime*, which increases montonically over ASM runs, and is consistent with the notion of moves that come before or after other moves.

A detailed treatment of time in abstract state machines is presented in [17]. Focusing on moves, called events, rather than on states, time is added to event

structures in a way that is consistent with the sequences of moves defined by an abstract state machine. All moves are ordered according to some notion of global time, but it is also possible for non-conflicting moves to have an undefined order according to the local time of a single thread.

But global time is not intrinsic to the reservation system and its clients. The reservation system grants or declines client requests according to its own rules, which may include its perception of the relative arrival times of those requests. And even with a single client, the fact that requests are issued in a particular order does not guarantee that the reservation system will perceive them in that order.

3.6 Summary

In summary, different kinds of abstract state machine have been explored, and all include the notion of progress in time through a sequence of well defined global states.

This is achieved by construction for single-agent abstract state machines. For distributed abstract state machines, that every initial segment of every run results in the same state also implies that a sequence of states through which the ASM progresses can be identified.

For independent parallel ASMs, the introduction of an abstract communicator [7] means that the parallel ASMs are brought together into a distributed ASM as before.

Alternatively, focus is given to one of the ASMs, with everything outside that being regarded as an environment that can be queried in [11]. This again means that progress in time is modelled, but it does not say how progress as perceived, for example, by a client, can be transformed to progress as seen by the reservation system.

The following section presents an approach that treats the point of view of every ASM equally, and explores what it means for one ASM to observe the state of another ASM. The requirements that must be met by the transformations that enable such observations are then elucidated.

4 Analogy with Concepts from Relativity

The notion of global state is fundamental to much of the work reviewed above. The term *sequential time* is defined in [5] to describe the progress of an ASM from state to state. That distributed algorithms are not, in general, sequential time algorithms is stated in [14], where intra-step interaction with other agents in the environment of an ASM is explored in depth. The notion of global state of a distributed ASM is defined in [12] by partially ordering the moves of the ASM agents so that contentious moves that would lead to inconsistent update sets are ordered, and by identifying as global the states the result from application of any maximal move in any finite prefix of the partially ordered set of moves. This is similar to the notion of global state defined in terms of cuts through Petri net representations of distributed runs by Glausch and Reisig [8].

A different approach is taken in [14]. There focus is given to a single ASM in a distributed environment. The ASM can query its environment, which has ASM agent-like behaviour. The ASM can also, while executing a step, observe updates that are made by the environment. The ASM operates in sequential time, and the states of the environment are not of concern.

Here, the aim is to enable any ASM agent to see shared locations as other agents see them. No preference is to be given to the perspective of any agent. This is inspired by ideas from relativity concerning permissible transformations between different frames of reference [16]. But for this purpose, it is not necesseary to have a global state, but only to have sufficient overlap between the states that contain the communicating ASM agents.

Towards this end, each ASM has its own state space and its own time. Each ASM progresses through its own well defined states (locally sequential time), but there is no demand for a coherent global state. The challenge is to describe interaction between ASMs and to identify the properties that must be maintained when facts that can be observed by one ASM agent are identified with facts observable by another.

4.1 Space, Time and Abstract State Machines

The space through which an ASM progresses is its state space. The symbols from its signature, excluding the derived names, form a basis for that state space. The paths that an ASM can follow through the state space are constrained by the initial state and by the ASM program.

The passage of time for an ASM is closely tied to a run of the ASM. In SDL, distributed real-time ASMs are defined using a real-valued monitored function *currentTime*, which refers to an external, physical time [12]. Consistency between *currentTime* and a concept of time based on the progress of an ASM through its state space is maintained by requiring that *currentTime* should increase monotonically over ASM runs. The passage of time while an ASM is in a given state is described in [11], where the logical time of an ASM is expressed in terms of interactions with the environment during a single step. But there too the passage of time through a run is described in terms of persistent queries made by an ASM in one state and the delayed responses received in a different, later state.

Here, an ASM has its own local time, which has a value of zero or more in an initial state, and which is incremented by a positive amount at the end of every transition. The increment is demanded so that no two states in a run are identical, even if infinite runs pass and re-pass through otherwise identical states, for example, to model continuous services.

Relativity describes concepts including *spacetime*, *coordinate system* and *frame of reference*. In relativity, events in spacetime are defined by assigning numbers to four spacetime coordinates. These numbers depend on a frame of reference, which is a system used to assign the numbers [16]. The values in one frame of reference can be transformed and used in another frame of reference, so long as the transformation maintains underlying physical laws.

For an ASM, a comparison can be made between spacetime coordinates and the symbols defined by the signature, including a symbol for local time but excluding derived names. The ASM program provides a frame of reference, by which values are assigned to these coordinates.

In general, different ASMs will occupy different state spaces and will have and different local times.

This reveals a point at which the analogy breaks down. In general, different ASMs exist in different state spaces, so translating an observation about the state occupied by one ASM agent to another ASM agent demands not only changes of coordinate values, or even of coordinate systems, but a change of dimensionality. That is, states must be projected onto the shareable dimensions in order to move the information between different state spaces.

A further difference between spacetime as known in relativity and the ASM spacetime outlined here is that the ASM spacetime is not continuous. In place of four real-valued spacetime coordinates, that values that can be assigned to an ASM vocabulary element do not, in general, form a continuous set. On the other hand, a concept not unlike differentiability within a local region is retained for an ASM state space in that the difference between a current state and a next state is contained; this is expressed as the Bounded Exploration Postulate in [5] or, more restrictively, as the small step requirement of [14].

4.2 The History of an ASM

An ASM has a *local history*. A local history of a single-threaded ASM is any sequence of states in a run of the ASM. An event in the ASM's history is a state, including the local time of the ASM. A single threaded ASM has a single ASM agent, and a history of a single threaded ASM is also the history of its agent.

A local history of a distributed ASM is the sequence of states associated the initial segments of a serialization of the partially ordered set of moves of the ASM.

This description of the history of a distributed ASM differs from the notion of history defined in [12]. There, only states that directly contribute to the computation represented by the run are retained, and states that differ from their predecessors only in their values of the external, physical *currentTime* are dropped. It also differs from the histories defined in [11], which record the order in which an ASM receives inputs from its environment during a single step.

The partial ordering of updates in a distributed ASM means that all the agents of the distributed ASM share a common time. Regarding the distributed ASM as a model of a single, multi-threaded computation, that common time, although global to the ASM agents, is local to the ASM. It need not be the same as the time observed by agents of other, parallel, independent ASMs.

4.3 Interaction and Interpretation of Events

In the distributed ASMs described by Glausch and Reisig [8] and in SDL [12], interaction is defined in terms of updates to locations that can be read or updated by more than one ASM. Interaction with the external environment is described

in a similar way. In SDL, locations that are updated by the environment and read by the ASM are called *monitored* locations, and locations that are read and written by the environment and by the ASM are called *shared* locations.

Updates performed by the environment are treated as occurring between transitions of the ASM by Blass and Gurevich [5], though it is also possible to model updates made by the environment during a move [11].

Outputs from an ASM to its environment are modelled as updates made by the ASM to locations that can be read by the environment [3, 5]. In this way, the environment is treated as moves by one agent in a distributed ASM that are always in contention with the moves of the original ASM.

Continuing the analogy with concepts from relativity, a state represents an event in the history of an ASM. If an external ASM agent is to observe any part of that event, then there must be an overlap between the state of the external ASM and the opservable part of the state of the original ASM. Also, the external ASM can only observe the event in terms of its own signature and its own local time. Only that part of the event that affects shared locations is visible to the external ASM, and from the perspective of the external ASM agent, the whole event is represented by its projection onto the shared locations.

Moreover, in general the state of an ASM is not stable while the ASM is performing a move [11, 14], though moves can be expressed as having a start, an intermediate point (during which the state is unobservable), and an end [17]. This addresses the fact that an observation of an incomplete state is likely to be unreliable.

4.4 Observation of an Event

Consider two abstract state machines, A and B. A in state S_A associates the value v with a location identified as $f_A(\overline{a})$. B in state S_B identifies the same location as $f_B(\overline{a})$. That is, there is an overlap between state S_B of B and state S_A, a state (event) in the history of A, which in turn means that an agent of B can observe part of a state in the history of A.

To enable this observation, part of S_A is transformed to the corresponding part of B's frame of reference. That is, S_A is projected onto $f_A(\overline{a})$, and $f_A(\overline{a})$ is mapped to $f_B(\overline{b})$, and so B observes the value of the shared location as A sees it.

This example illustrates the first requirements on transformations that map observations made by one ASM to the frame of reference of another ASM.

Common location

Suppose A and B are two ASMs. Then a transformation of states from the frame of reference of A to that of B

- *is defined for projections of the states of A onto locations that are also accessible to B;*
- *maps locations of A to locations of B so that $f_A(\overline{a})$ is mapped to $f_B(\overline{b})$ iff the two terms refer to the same location.*

But suppose $f(\overline{a}) = v$ is true in S_A, and $f_B(\overline{b}) = w$ is true in S_B and $v \neq w$. In that case, the two states cannot coincide.

This leads to the following requirements affecting transformation of the time of observations between abstract state machines:

Consistent perspective

Suppose A and B are two abstract state machines.

- *Before the local times of A and B can be synchronized, A and B must be in states that associate the same values to all common locations;*
- *If $f(\overline{a}) = v$ is true in S_A, and $f_B(\overline{b}) = w$ is true in S_B and $v \neq w$, then either S_B is in the past of S_A and S_A is in the future of S_B or vice versa. So if a transformation of the event S_A to the frame of reference of B enables an agent of B to observe that values of one or more commonly accessible locations are different from A's perspective, then that transformation must transform the local time of A to a value that is different from the local time of B. Furthermore, if B's local time is t_B in state S_B, and A's local time is t_A in state S_A, then the transformation must map t_A to a value representing the local time of a state of S_B for which all the commonly accessible locations share the same values from the points of view of both A and B.*

This still allows ASM A in state S_A and B in state S_B to have different values for a given location, provided B does not claim see the value as seen by A as occuring at Bs current local time and vice versa.

It also gives form to the conditions for synchronizing and merging the histories of two ASMs.

Synchronize and merge

In order to synchronize two ASMs and bring their processes together to form a distributed ASM with a shared local time;

- *all commonly accessible locations must be updated so that they have the same values regardless of which ASM agent observes them;*
- *the local time of the two ASMs must be synchronized;*
- *the ASMs must proceed in step.*

While an ASM is mid-transition, its state is not stable. Updates could be rolled back (undone) before the transition is complete. Repeated observations of the value of a location may produce inconsistent results during a transition

so only location-value associations on entry to or exit from a transition can be viewed as observations of definite events in the history of an ASM.

The ASMs discussed in [14] allow multiple interactions between an ASM and its environment to occur during a step, with the proviso that an ASM never completes a step before all the queries emanating from that step have been answered, and that the only information about the environment available to the ASM is that obtained in response to queries.

Queries that persist beyond a single step are addressed by [11]. There the ASM that makes the query also supplies a location to receive its response.

These approaches do not rely on direct observation of environmental values by the ASM, but instead allow the environment to control when answers to queries are released, and the ASM to control where those answers are located. Provided the environment only releases stable values, and places those values at the locations stipulated by the ASM, no problems arise.

This leads to the following constraint on observable locations.

Scratch work is private

Locations that can be observed should not be used for changeable 'scratch work' carried out by an ASM during a step. In other words, if an agent of B observes that $f_A(\overline{a}) = v$ is true in state S_A of A, and that $f_A(\overline{a}) = w$ is true in state S'_A, then either $v = w$ or $S_A \neq S'_A$.

A final constraint concerns the immutability of an ASM's history.

History is immutable

If S_A precedes S'_A in the history of A, then any observation of these two states by B must maintain that ordering. That is, the local time of S_A according to A is less than the local time of S'_A, and this ordering must be maintained when these events are represented in B's history.

5 Practical Applications

The requirements on interaction outlined above are discussed below with reference to the ticket reservation service and its client applications, and to the reservation service treated as a web service.

5.1 The Reservation Service Revisited

Suppose the ticket reservation service were modelled as an abstract state machine, *Resv*. Now suppose a booking agency created an application, modelled as

an abstract state machine *App*, that made use of *Resv*. Merging *Resv* with *App* as a single distributed ASM is impractical, because *Resv* is already deployed, and is likely to have clients other than *App*, with timing requirements likely to be quite different from those of *App*.

How do the requirements outlined above help define non-contentious interaction?

The requirements concerning naming commonly accessible locations, **Common location** is essential to any interaction.

History is immutable means, for example, that if the history of *Resv* records a ticket as available in one state of *Resv*, and as unavailable in a subsequent state of *Resv*, then the states of *App* that correspond to those facts must also occur in the same order.

This leaves two further requirements to consider.

The first concerns consistency between the perspectives of *Resv* and *App*.

The requirement for **Consistent perspective** means that if *App* observes a common location as having a different value from that represented by *Resv*, then that observation must be at a point in time that is different from *App*'s current local time. If the observation refers to a ticket that *Resv* shows as available and *App* wished to book, then the two must be brought into harmony in a future state of *App* and *Resv*. This also applies to all the other applications that might be attempting to book the same ticket, but given an application-specific limit on the number of bookings applicable to each ticket, the two requirements will conspire to show the ticket(s) booked by some application(s) and not booked by others in a future of all the ASMs.

It does not say how the required transaction agency is to be preserved, but only that it should be preserved. For example, a timestamping approach might model the tickets themselves as ASMs, where a ticket maintained its last read and last update timestamps using its own local time, passing that time to applications via *Resv*.

The second concerns visibility of incomplete states; for example, it concerns the possibility that *App* might observe an uncommitted reservation in *Resv*. The requirement **Scratch work is private** prevents observation by other applications of an uncommitted update made, for example, by the application *App*.

5.2 Web Application

Web services provide a good example of distinct agents enacting parallel processes that can be modelled using abstract state machines whose steps proceed without reference to a global state or time.

A web application provides a service that can be accessed using a web browser. Registering a web application with the Universal Description, Discovery and Integration (UDDI) directory service, make it possible to integrate that application into other applications [18]. The original application then becomes a component of the new application. According to the w3schools tutorial on web services description and UDDI, if there were an industry-wide UDDI standard for checking rates and reserving flights, and if airlines registered their services in a UDDI

directory, then travel agencies could communicate with the airlines' reservations services using the interface published in the UDDI directory [18].

Suppose the web application for reservations were represented as *Resv*, the UDDI directory service as *Dir* and the client application as *App*. The *Dir* entry for *Resv* defines the locations that are observable by *App*, and helps enable definition of the transformations by which each *App* interacts with *Resv*. But *Dir* is itself an ASM, with a large observable space, whose transformations must comply with the above requirements.

Developing such transformations would mean that different variants of *Resv* and of *App* could be modelled as abstract state machines, and could interact without the need for a global state and time. This would be more in tune with the brokering philosophy of UDDI than would a model that prescribed, for example, an absolute global system time.

6 Summary

Some promising initial steps towards ASM modelling that allows parallel ASMs to interact without demanding that they should refer to a common global state is outlined above. Inspired by concepts from relativity, the observation of states of one ASM by another ASM is described in terms of the requirements that must be fulfilled by an transformation of observations between the different state spaces occupied by the ASMs. Some of the practical implications of these requirements are briefly discussed. A fuller study based on application of these ideas to real systems would be desirable, and would help the development of practical transformations.

References

1. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In: Specification and Validation Methods, pp. 9–36. Oxford University Press (1995)
2. G.Y.: Evolving Algebras 1993. Lipari Guide, 2005 version with table of contents and footnote, `http://research.microsoft.com/en-us/um/people/gurevich/opera/103.pdf`
3. Börger, E., Stärk, R.: Abstract State Machines – a Method for High-Level System Design and Analysis. Springer (2003)
4. Börger, E.: The ASM Refinement Method. Formal Aspects of Computing 15, 237–257 (2003)
5. Blass, A., Gurevich, Y.: Abstract State Machines Capture Parallel Algorithms. ACM Transactions on Computational Logic 4(4), 578–651 (2003)
6. Blass, A., Gurevich, Y.: Abstract State Machines Capture Parallel Algorithms: Correction and Extension. ACM Transactions on Computational Logic 9(3), Article 19 (2008)
7. Glässer, U., Gurevich, Y., Veanes, M.: Abstract Communication Model for Distributed Systems. IEEE Transactions on Software Engineering 30(7) (2004)
8. Glausch, A., Reisig, W.: Distributed Abstract State Machines and Their Expressive Power. Humboldt University Berlin (2006)

9. Glausch, A., Reisig, W.: An ASM-Characterization of a Class of Distributed Algorithms. In: Abrial, J.-R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 50–64. Springer, Heidelberg (2009)
10. Börger, Cisternino, A., Gervasi, V.: Ambient Abstract State Machines with applications. Journal of Computer and System Sciences 78(3), 939–959 (2011)
11. Blass, A., Gurevich, Y.: Persistent Queries in the Behavioral Theory of Algorithms. ACM Transactions on Computational Logic 12(2), Article 16 (2011)
12. International Telecommunication Union (ITU): Recommendation Z.100 Annex F1 (11/00) SDL formal definition – General overview. http://www.itu.int/rec/T-REC-Z.100-200011-S!AnnF1/en
13. Glässer, U., Gotzhein, R., Prinz, A.: The formal semantics of SDL-2000: Status and perspectives. Computer Networks 42, 343–358 (2003)
14. Blass, A., Gurevich, Y.: Ordinary Interactive Small-Step Algorithms – I. ACM Transactions on Computational Logic 7(2), 363–419 (2006)
15. International Telecommunication Union (ITU): Recommendation Z.101 (12/11) – Specification and Description Language - Basic SDL-2010 (2010) http://www.itu.int/rec/T-REC-Z.101-201112-I/en
16. Norton, J.D.: General Covariance and the foundations of general relativity– eight decades of dispute. Reports on Progress in Physics 56(7), 791–858 (1993)
17. Graf, S., Prinz, A.: Time in State Machines. Fundamenta Informaticae, 77(1-2), 143–174 (2007)
18. WSDL and UDDI, http://www.w3schools.com/wsdl/wsdl_uddi.asp