System Executions

Uri Abraham *

April 4, 2014

Abstract

A Tarskian system execution is a structure that describes a run of a system that is composed of several processes. Two types of system executions are defined: temporal based and precedence based. We exemplify the usage of system executions with the specification different types of registers: serial, atomic, safe, and regular.

1 Tarskian system executions

The notions of signature and its interpretation that were introduced and explained in the first lectures are very general and are suitably applied in a great variety of mathematical situations. It turns our that for specification of systems and for the correctness proofs of concurrently operating processes a rather restricted family of signatures and interpretations (called system executions) is found to be useful. In fact, Lamport introduced this notion of system executions in a series of papers (see mainly "On interprocess communication" published in Distributed Computing) in which he showed how their properties can be used for arguing about systems and about their low level and high level events. Lamport's system executions however were not defined as logical structures that interpret logical signatures, and so the link between mathematical logic and the constructions of concurrency models was not explicitly established. By exploring the idea that Lamport's original definition of system executions can be developed within the context of logical structures we are able to reach a greater degree of formality and mathematical clarity. In this lecture we define the notion of Tarskian System Executions, and show how they can be used to define different types of

*Departments of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva. Prepared for the course Models for Concurrency 2014.

registers (safe, regular, and atomic). We first revise the notions of signature and interpretations.

In order to define a predicate language one has to specify the set of symbols of that language, which is technically known as its *signature*. Given such a signature and the resulting language L, an interpretation of L is a structure in which the symbols of L acquire their meaning. Different languages will have different signatures, and any signature has typically many possible interpreting structures. We deal here with multi-sorted languages. A signature for a multi-sorted language is a list of the following items.

1. Names of sorts. The sorts are the types of objects that populate the universes (interpretations) of the signature language.

2. Names of variables. The signature can connect variables to specific sorts. For example, if we have a sort named *Event*, we can stipulate that $e, e_0, e_1, \ldots$ are all *Event* variables. This allows one to form shorter quantified formulas. For example, $\exists e \varphi(e)$ would mean: there exists an *Event* e such that $\varphi(e)$ holds. We allow to use sorts as predicates, and then we can write that statement as $\exists x(\mathit{Event}(x) \wedge \varphi(x))$. Sometimes we can even write $\exists x \in \mathit{Event}\ \varphi(x)$.

3. The signature lists predicates (name of relations) and for each predicate its arity is given. The arity of a predicate is the number of "places" or entries that the predicate take. The signature can also stipulate the sort of the k-th entry of a predicate. For example, we may have a binary predicate $<$, that is, a relation symbol of arity 2, and we may require that both of its arguments are of sort *Event*.

4. The signature lists names of functions, and their arities. The signature can also stipulate the sorts of each function entry and the sort of values taken by the function. For example, we may have in our system execution signature a unary function called *Val* which gives for every read or write event e its data value $\mathit{Val}(e)$.

5. The signature lists constants and their sorts.

Given any signature, there is a standard inductive definition of the resulting *language*, which is the set of all expressions and formulas built with quantifiers and connectives from the elements of the signature.

An interpretation M of a signature consists of:

1. A universe, namely a set A of elements.

2. For each sort S in the signature a set $S^M \subseteq A$ which represents the members of M of sort S.

3. For every relation symbol R of arity k in the signature, a relation $R^M \subseteq A^k$. In fact, if the signature stipulates that the i-th entry of R is of sort S_i then we have $R^M \subseteq S_0^M \times \cdots \times S_{k-1}^M$.

4. For every function symbol F in the signature, of arity k, F^M is a function from A^k to A. Again, it has to respect the stipulations made by the signature about the domain and value sorts of the function.

5. For every constant c of sort S in the signature, $c^M \in S^M$.

Note: The equality symbol $=$ appears in every signature and its interpretation in every structure is the identity relation.

System executions

In this course we shall be interested in very specific kind of Tarskian structures which we call system executions, borrowing the term introduced by Lamport[1]. There are two basic types of system executions: those in which time appears explicitly as some "time" sort to represent the moments or instants of time, and those in which only the temporal precedence relation $<$ (or the symbol $\rightarrow$ that Lamport uses) appear without any supporting time line. We say "moment based" for the former type, and "precedence based" for the latter. When we just say system execution, we usually mean precedence based without any representation of the temporal domain.

Definition 1.1 *A (precedence based) system execution signature is a signature that contains the following items (and possibly more).*

1. There are two sorts: *Event* and *Atemporal*. *Event* and *Atemporal* are disjoint.

2. There is a binary relation symbol $<$ defined on *Event*. It is called the temporal precedence relation. In some of my lectures and sometimes in writing, I use $e_1 \rightarrow e_2$ instead of $e_1 < e_2$, and $\rightarrow=$ instead of $\leq$. A possible problem with the $<$ notation is that it may be confused

[1] You may say "Tarskian system executions" and "Lamport system executions" if you want to be more precise, but in these lectures the term "system executions" always refers to Tarskian system executions as defined here

with the ordering relation on the natural or real numbers, but usually the context is so clear that such a confusion will not take place. The $\rightarrow$ notation will be used in connection with the message passing framework to denote the causal relation rather than the temporal precedence relation.

3. The signature contains is a unary predicate denoted *terminating* on *Event*. Intuitively, *terminating*(e) says that event e has a bounded duration.

4. There are possibly other sorts (and subsorts), predicates, function names, and constants in the signature.

Definition 1.2 *A (precedence based) system execution is an interpretation M of a precedence based system execution signature such that:*

1. *Relation $<^M$ is a partial ordering of Event^M which satisfies the Russell–Wiener property.*

2. *Lamport's finiteness property holds for the terminating events. That is, for every terminating event e there exists a finite set X of events such that if a is any event not in X then $e <^M a$.*

3. *If $a <^M b$ then a is terminating (that is, terminating(a)).*

Moment based system executions

Definition 1.3 *A moment based system execution signature is a signature that contains the following items (and possibly more).*

1. There are three sorts: *Event*, *Atemporal*, and *Moment*. *Event* and *Atemporal* are disjoint, and *Moment* is a subsort of *Atemporal*.

2. There is a binary relation symbol $<$ defined on *Moment*. (This is the global-clock ordering relation.) We also write $x \leq y$ as a shorthand of $x < y \vee x = y$.

3. There are two functions *Left_End* and *Right_End* from *Event* into *Moment*. We think of event e as extended in time and represented by the interval $[$*Left_End*(e), *Right_End*$(e))$.

4. *terminating* is a unary predicate on *Event*.

5. There is a constant ∞ in sort *Moment*. (∞ will be the right end point of non-terminating intervals.)

6. There are possibly other predicates, functions, and constants in the signature.

Definition 1.4 *A moment based system execution is an interpretation M of the signature defined above so that:*

1. *$Moment^M$ is linearly ordered by $<^M$. (In most of our applications, the order-type of $Moment^M$ is that of the natural numbers with an additional end point.) ∞^M is its end-point (the last member of $Moment^M$).*

2. *The following holds in M:*

   (a) *For every event e, $Left\_End(e) \leq Right\_End(e)$.*

   (b) *A precedence relation $<$ is defined on the events as follows: for every events e_1 and e_2,*

   $$e_1 < e_2 \text{ iff } Right\_End(e_1) < Left\_End(e_2). \quad (1)$$

   So we use the same symbol, $<$, for two purposes. $m_1 < m_2$ denotes that moment m_1 is before moment m_2, and $e_1 < e_2$ to denote that event e_1 is earlier than event e_2. In my experience this does not create any confusion.

   (c) *For every event e, $Left\_End(e) < \infty$. For every event e, $terminating(e)$ iff $Right\_End(e) < \infty$.*

3. *For every terminating event e there exists a finite set X of events such that if a is any event not in X then $e < a$. (So $Event^M$ is countable.)*

The notion of *reduct* between structures is used to explain the connection between moment based and precedence based system executions. We say that signature L_1 is a sub-signature of L_2 if every item of L_1 is also an item of L_2. So any sort of L_1 is a sort of L_2 (but possibly some sorts of L_2 are not in L_1), every predicate of L_1 is also a predicate of L_2 (and they have the same arity and the same specification of sorts) etc. If M_2 is an interpretation of sort L_2, then a *reduct* of L_2 (to the L_1 signature) is the structure M_1 obtained from M_2 by disregarding the interpretations of those items of L_2 that are not in L_1. So the universe of M_1 consists of all members of the universe of M_2 that are of types that are in L_1. The predicates and

function symbols of L_1 have the same interpretation in M_1 as they have in M_2.

It follows now that if M_2 is a moment based system execution, then the reduct M_1 of M_2 to the signature of precedence based system executions is a precedence based system execution. So the universe of M_1 is obtained by dropping the time axis $Moment^{M_2}$ from the universe of M_2, and defining the precedence relation on the events by the formula (1) above. (The functions *Right_End*, *Left_End*, and the constant ∞ are also removed.)

Theorem 1.5 *Every precedence based system execution is a reduct of some moment based system execution.*

In fact we have essentially proved this theorem in the Time chapter when we proved the representation theorem. We recall this theorem and the relevant definition.

Definition 1.6 (Lamport) Global-time structures*: Let $\mathcal{S} = (E, <, T)$ be a structure where E is a non-empty set (the universe), $<$ is a partial ordering of E, and T a unary predicate on E represented as a subset. Members of E are called events, T is a subset of E called the set of terminating events, and $<$ is called the precedence relation of $\mathcal{S}$.*

Suppose that relation $\not>$ is defined by $a \not> b$ iff $\neg(b < a)$. Then $\mathcal{S}$ is called a global-time structure *iff:*

1. *$\forall a, b, c, d\ (a < b \not> c < d \longrightarrow a < d)$. (This is the Russell–Wiener property. So $<$ is an interval ordering.)*

2. *Every antichain is finite, and the finite predecessors property holds: that is for any $a \in E$ the set $\{x \in E \mid x < a\}$ is finite.*

3. *If $\neg T(e)$ then there is no $x \in E$ such that $e < x$. In other words, if an event e is followed by some event x, then e must be terminating.*

4. *For any terminating $a \in E$, $\{x \in E \mid x \not> a\}$ is finite. (The finiteness property of Lamport, for terminating events only: the set of events that follow a terminating event is co-finite (a subset is co-finite if its complement is finite).)*

Definition 1.7 *[Representations] Let $(L, <_L)$ be a linear ordering and $\mathcal{S} = (E, <_E, T)$ be a global-time structure. We say that μ is a* representation *of $\mathcal{S}$ in L iff μ is a function defined on E which takes values that are intervals of*

L and in which terminating events are represented by bounded intervals, and nonterminating events are represented by right unbounded intervals. That is:

1. *For each $e \in E$, $\mu(e)$ is an interval in L, and*

$$e_1 <_E e_2 \text{ iff } \mu(e_1) \prec_L \mu(e_2).$$

2. *If $T(e)$ holds, then $\mu(e)$ is of the form $[a, b]$, but if $\neg T(e)$ holds (that is e is nonterminating) then $\mu(e)$ has the form $[a, \infty) = \{x \in L \mid a \leq_L x\}$.*

Theorem 1.8 *Any global-time model has a representation as intervals of natural numbers. That is, $(L, <_L)$ can be chosen as $(\mathbb{N}, <)$.*

Exercise 1.9 *Prove theorem 1.5. Let M be a precedence based system execution. Show that there is a moment based system execution N so that M is the reduct of N to the language of M.*

2 Example: register specifications

As an example of system executions we describe here how properties of registers can be expressed by means of these structures. A register is a shared memory location, which supports read and write operations, and we shall specify here serial, safe, and regular registers. To enable this specification we shall first define an event based "register signature" and language $L_{registers}$. This signature extends the system execution signature. So sorts *Event* and *Atemporal* are in the register signature, and so are relations $<$ and *terminating*. Sort *Atemporal* contains a sub-sort of data values which are the values written and read on the registers. In addition, we have a sort of addresses which identify the different registers ("shared memory locations"). As predicates we have unary predicates *Read* and *Write*, and then the *Val* function and the *Address* function. The *Val* function is defined on the *Event* sort and is into the data sort, and the *Address* sort is defined on the *Event* sort and is into the sort of addresses. An additional function symbol which is not absolutely necessary but which we find quite useful is ω, a unary function from *Event* into *Event*. The usage of these symbols is explained below.

Here are some statements which can be expressed in this language.

1. Given a memory location X (think of it as a register name), to say that event e is a read or a write on X we can have the following formula

r-w$_X(e)$: $(Read(e) \vee Write(e)) \wedge Address(e) = X)$. So the function *Address* is used to specify the name of the register with which the read/write event is associated.

2. To say that w is a write event on X that precedes all other write events and all read events (an initial write event) we can use the formula: $Write(w) \wedge Address(w) = X \wedge \forall e(\text{r-w}_X(e) \wedge e \neq w \rightarrow w < e)$. Here you can see an additional reason why we use $<$ rather than $\rightarrow$ for the precedence relation, namely since we use $\rightarrow$ for implication.

Exercise 2.1 *Write each of the following English statements as formulas in the register language, as shown for example in the first item.*

1. If x is a read event then $\omega(x)$ is a write event. Formally: $\forall x(Read(x) \rightarrow Write(\omega(x)))$.
2. For every two read events there exists a write event in between them.
3. No read event is overlapping with a write.
4. There is a write event that is overlapping with all other writes.
5. If $x \neq y$ are read events then the value of $\omega(x)$ is different from the value of $\omega(y)$.
6. For every write event w there are infinitely many read events r such that $w = \omega(r)$. (This statement cannot be expressed directly, but you can use the finiteness condition in order to prove that an equivalent statement is expressible.
7. Expand the signature by adding the sort of natural numbers, and express the following: The write events are linearly ordered and there are two (initial) write events that precede all reads. Moreover, the values of read/write events are natural numbers, and the value returned by a read event is between the values of the two rightmost write events that precede it.
8. There is only one register (all read/write events have the same address).

Serial, safe, regular and atomic registers

Informally speaking, a register is serial when all read and write events connected with the register are linearly ordered under the precedence relation $<$, and are such that the value returned by any read event is the value last write event onto that register. In order to formally define these registers, we shall list a series of "axioms" and we will say that any system execution in which these axioms holds represents a serial register. So let X be a specific register name.

Exercise 2.2 *Express the fact that register X is serial.*

Recall that events a and b are said to be concurrent if neither $a < b$ not $b < a$.

Now we deal with a weaker sort of registers: safe registers. Intuitively speaking, a register is safe when every read returns the value of the rightmost write preceding it, provided that the read is not concurrent with any write. So a read that is concurrent with a write will return an arbitrary value (in the type of returned values), possibly not even a value that is written by any write event. But if a read R is not concurrent with any write, then all writes are either before R or after R and there is a write that is before R (an assumed initial write). In that case, if W is the last write before R then the value read by R is the value written by W. More formally, a register is safe if the write events are linearly ordered, there exists a write event that precedes all reads, and the value of any read event that is not concurrent with any write is the value of the last write that precedes it.

Exercise 2.3 *Write a formal statement saying that the write events are serially ordered and the register X is safe. The function ω is not used in this sentence.*

Technically, it is convenient to express the safeness of a register R by using "return" function ω, which associates with each read r of R a write $w = \omega(r)$ on R such that for every read r:

1. It is not the case that $r < \omega(r)$.
2. There is no write w_1 onto R such that $\omega(r) < w_1 < r$.
3. If $\omega(r) < r$, then $\mathit{Value}(r) = \mathit{Value}(\omega(r))$.

Exercise 2.4 *Verify that the existence of a function ω as above is equivalent to the safeness of R as formulated in the previous exercise.*

Clearly, a register R is serial in a system execution iff it is safe and the read/write events are linearly ordered. We expect our registers to be serial, and certainly no one is going to rely on a shared memory system that only ensures safeness of its registers. The term "safe register" is thus misleading since the assurance that these registers offer is less than minimal. They are important though for theoretical reasons, and it is very interesting to learn what can be achieved with such minimal guarantees that safe registers offer. We will see some algorithms that do not require anything more from their registers than that they are safe. The proof that safety of the registers suffices for the correctness of the algorithm deepens our understanding of these algorithms and of concurrency theory in general.

We now define regular registers. A register X is regular (in a system execution that is a structure for the $L_{registers}$ language) if the read events are linearly ordered, there is a write event that precede all other read/write events on that register, and for every read event r there is a write events w so that $Val(r) = Val(w)$ and the following two properties hold:

1. It is not the case that $r < w$, and
2. there is no write w' on that register such that $w < w' < r$.

Exercise 2.5 *Write a sentence that expresses the property that register X is regular.*

Now we define atomic registers. Let M be a system execution in which the language $L_{registers}$ or a richer language is interpreted. Register X is *atomic* in M if there exists a relation $<^*$ on the events in M so that the following holds.

1. $<^M \subseteq <^*$. That is, $<^*$ extends $<^M$: if $a <^M b$ then $a <^* b$ for every events a and b.
2. If we replace the relation $<^M$ (which is the interpretation of $<$ in M) with the relation $<^*$, then a system execution results in which register X is serial. Specifically, this means that:
   (a) The read/write events on register X are linearly ordered in the $<^*$ relation.

(b) For any read r of register X there is a write event w on X so that $w <^* r$, $Val(r) = Val(w)$, and there is no write event w' on X with $w <^* w' <^* r$.

The definition of atomic registers is, as you see, very different from the definitions of the other types of registers. It is obtained by quantifying over the class of relations. We say "register X is atomic if *there is a relation* $<^*$ so that.... We say that this is a "second-order definition". Namely the quantification is over relations rather than over the members of the universe of M. We shall continue to investigate atomicity and its generalization (linearizability) later in our course.