

Linear State Machines Formal Model

Legalese (Dustin Wehr, Meng Wong, ...)

October 15, 2017

Click most terms (in [this color](#)) to jump to their first underlined usage.

Contents

1	Basics: events, time, and traces	1
2	Simple Contracts	3
2.1	Execution	4
3	Infinite state: Global Variables and Event Parameters	5
3.1	Execution	6
4	May-Later and Must-Later	6

1 Basics: events, time, and traces

Every contract specifies a time unit; it is the smallest unit of time that one writes constraints about or does arithmetic with. We expect it will most often be days. A time stamp is a natural number that we think of as being in units [time unit](#).

An event is composed of an action, a role, a [time stamp](#), and optionally some parameters (but parameters will not be introduced until Section [4](#)). The [actions](#) and [roles](#) are fixed finite sets. In this first version of L4, there is exactly one participant of each [role](#).

All events are modelled as actions, and a special [role](#) Env is used to model events that have no agent ([role](#)).

A trace is a sequence of events. The time stamps of the events must be nondecreasing. Thus, within the smallest unit of time, any number of events can happen; however, they are always strictly ordered. The idea here is that we want events to be strictly ordered for simplicity and to minimize the size of the space of execution traces, but if we made the time stamps strictly increasing, we would need to be working at a level of granularity for time that is at least one level smaller than the smallest unit of time that would appear in an informal version of the contract (at least when time unit = days, since contracts that use days as their minimum unit generally do not require that all events happen on different days).

A contract has a fixed finite number of states, one of which is designated the start state, and which includes at least the following:

- fulfilled
- breached(X) for each nonempty subset X of the roles. There is also an action breaches(X) for each such X , and breachEvent(X, t) is defined as the event $\langle \text{breaches}(X), \text{Env}, t \rangle$

Between any two events in a trace, the contract is in some global state which consists of at least a time stamp for the current time and a state (in Section 3, global variables will be added).

A contract has a finite directed edge-labeled multigraph¹ which we might call its skeleton; the nodes are the states, and each directed edge, which we will call a transition, is labeled with an action. Some notation:

- For r a role, an r -transition is a transition whose role is r .
- For a an action, an a -action (a -transition) is an event (transition) whose action is a .
- For s a state, the incoming s -transitions (outgoing s -transitions) are the edges coming into (going out of) s .

Every transition is one of the following three types. They will be explained in more detail in the next section.

- A may next transition is a permission.
- A relievable must next transition will be the most-used kind of obligation.
- A must next transition is the strongest possible obligation.

¹By this I mean there may be multiple edges from one node to another, but they must have different labels.

Since the environment **Env** cannot breach a contract or be *obligated* to do anything, no **Env**-transition can be a **must next transition** or a **relievable must next transition**. That completes the definition of the finite directed graph **skeleton** of a **contract**.

2 Simple Contracts

We obtain the first complete, but limited, model of contracts, called simple contracts, by elaborating on the definitions from the previous section.

Each **transition** τ is associated with a transition guard $\text{transGuard}_\tau(\cdot)$. In the case of **simple contracts**, it is just a relation on **time stamps**. A **transition** τ is enabled upon entering a **global state** with **time stamp** t iff $\text{transGuard}_\tau(t)$ is true.²

The **transition guards** must satisfy the following conditions, which would be statically verified in a language for **simple contracts**:

unambiguous absolute obligation condition: For every **time stamp** t , if some **must next transition** evaluates to true (at t) then every other **transition guard** evaluates to false (at t).

choiceless relievable obligations condition: For every **role** r and **time stamp** t , if one of r 's **relievable must next transitions** evaluates to true (at t) then any other **relievable must next transitions** for r evaluate to false (at t).

We say that a **transition** τ and an **event** $E = \langle a, r, t \rangle$ are compatible iff they have the same **action** a and the same **role** r . This definition will be modified in Section 3 when we add **event** parameters.

Each **transition** τ is also associated with a deadline function $\text{deadline}_\tau(\cdot)$, which gives a **time stamp** for a deadline when:

- an **enabled may next transition** (a kind of permission) expires.
- an **enabled must next transition** (the strong form of obligation) causes a breach by role_τ ³ if a **compatible event** has not been performed by the deadline.

²Currently, LSM examples are written assuming the **transition guards** of a **state**'s **transitions** get evaluated only once upon entering the **state**. It would also be reasonable to guess that they get evaluated once per **time unit** while the **contract** is in that state. This is not ideal.

³Which recall, in this formal model means a transition to the state $\text{breached}(\{\text{role}_\tau\})$

- an **enabled relievable must next transition** (the weak form of obligation) causes a possibly-joint breach by role_τ if a **compatible event** has not been performed by the deadline **and** no other permitted **event** is performed by the deadline.

For **simple contracts**, a **deadline function** is just a function from **time stamps** to $\text{timeunit} \cup \text{timestamps}$. If d is such a function, and a state is entered at **time stamp** t , then:

- If $d(t) \in \text{timestamps}$, the deadline is $d(t)$.
- If $d(t) \in \text{timeunit}$, the deadline is $t + d(t)$.

That completes the definition of **simple contract**. In the next section, we give the complete definition of how a **simple contract** executes on a **trace**.

2.1 Execution

A **simple contract** of course starts in its **start state**. Let E_1, E_2, \dots be a finite or infinite **trace** (recall: a sequence of **events**), as defined in Section 1. Let G_i be the **global state** that follows E_i for each i .

G_0 is $\langle \text{startstate}, 0 \rangle$.

Let $i \geq 0$, and assume execution is defined up to entering G_i . To reduce notational clutter, let us use the aliases:

$$G = \langle s, t \rangle = G_i \quad E = E_i \quad G' = \langle s', t' \rangle = G_{i+1}$$

Case 1: There is some **enabled must next transition** τ in G . If there is any other **enabled transition**, then this **contract** (not just this **trace**) violates the **unambiguous absolute obligation condition**, and so is invalid.⁴

- If E is **compatible** with τ and E happens within τ 's deadline, then the next state must be target_τ .⁵ This means A fulfilled the obligation created by τ .
- Otherwise, E must be $\text{breachEvent}(\text{role}_\tau, \text{deadline}_\tau(t) + 1)$.

Case 2: There is no **enabled must next transition** in G . From the set of **enabled may next transitions** of s **and** the set of **enabled relievable must next**

⁴Recall that a language (tool) for **simple contracts** will verify that such a thing can't happen.

⁵i.e. if $t' \leq \text{deadline}_\tau(t)$ then $s' = \text{target}_\tau$.

transitions in G , compute the deadline for each, and discard the **transitions** whose deadline has passed by the time E happens;⁶ let T_p be the resulting set of **transitions**. From the set of **enabled relivable must next transitions** in G , compute the deadline for each, and discard the **transitions** whose deadline is not the unique minimal **time stamp** t^* within that set; let T_o be the resulting set, and let R be $\{\text{role}_\tau \mid \tau \in T_o\}$. Then E is either:

- An event compatible with T_p .
- $\text{breachEvent}(R, t^*)$. This means that all of the **roles** whose **relivable must next transition** expires first are jointly responsible for the breach.

3 Infinite state: Global Variables and Event Parameters

Add to the definition of **contract**:

- A fixed finite set of typed **global variables**. The **global variables** are ordered, so we may describe their collective types as a single tuple GVarTypes . Add to the definition of **global state** an assignment of values to the **global variables**.
- An assignment of types to the **actions**. This allows **events** to have parameters. We refer to such a type as an **action-parameters domain**, and the specific action-parameters domain for **action** a is ParamTypes_a .

events receive the following modifications:

- Each **action** a additionally has a **global variables transform**, denoted transform_a , which is a function from $\text{GVarTypes} \times \text{ParamTypes}_a$ to GVarTypes .
- And the definition of a -transition is extended:
 - The **transition guard** attached to an a -transition may now depend on the values of the **global variables**; i.e. it is now a relation on $\text{timestamp} \times \text{GVarTypes}$.
 - Each a -transition additionally gets an **action-schema constructor** called actionSchema_a . This is a function from GVarTypes to a set of **events** all of whose **action** is a . We call such a set of actions an **action schema**.

The **unambiguous absolute obligation condition** is updated: Let T be the set of **transition guards** of the **must next transitions** of some **state** s . For any **global**

⁶i.e. discard τ if $\text{deadline}_\tau(t) > t'$.

variables assignment σ and **time stamp** t , at most one of the **transition guards** in T evaluates to true. **Todo**⁷

Note (probably to move to some other section or document): it will often be the case in a language for **contracts** that we simultaneously define an **action** a and a **state** JH_a (for “ a Just Happened”, to fit its literal meaning). In this case, the **incoming** JH_a -transitions are exactly the set of a -transitions. As a convenience, a language for **contracts** will allow the **outgoing** JH_a -transitions to depend directly on a ’s parameters (that is, for the **transition guard** and **action-schema constructor** to depend on a ’s parameters). This is merely a convenience because, as we will see when we define execution, one can achieve the same effect by introducing new **global variables** that are only used by a and JH_a ; a uses transform_a (recall, its **global variables transform**) to save its parameter values to those new **global variables**, so that the **outgoing** JH_a -transitions can then refer to them.

3.1 Execution

4 May-Later and Must-Later

This section does not actually change the definition of **contract**. Instead, it defines, essentially, an often-useful **contract** structure that is likely to be supported with custom syntax in a language for **contracts**.

We have so far been noncommittal about what types are available for **global variables** and **action-parameters domains**. We will see later that the types strongly affect expressivity. As a special case, the reader should convince themselves that any **contract** that uses only boolean (or other finite domain) types can be simulated by a **simple contract** (using a much larger number of **states**).

⁷Later will be: For any **global variables** assignment σ and **time stamp** t that makes s ’s precondition true, at most one of the **transition guards** in T evaluates to true.