L4 / Linear State Machines Formal Model

Legalese.com

January 25, 2018

I recommend joining #dsl on our Slack and introduce yourself before delving into this. Most of the L4 documentation needs updating/improving, including this document, and Dustin or Meng will be much more motivated to prioritize that if they know there are people waiting for it.

Contents

1 Who this is for and how to read it

Please note that we have not yet taken the time to make this document as widely accessible as it will be eventually, because the contents are still changing frequently.

This document defines the programming language-independent mathematical model, (very) tentatively called "linear state machines," that we use to define the semantics of our formal contracts language L4. It is written especially for computer scientists and mathematicians, especially Formal Verification experts, who aren't interested in, or are turned off by, the finer details of L4. For example, consider the component of the LSM model called a "global vars transform" introduced in Section **??**. It is a mathematical function, which if its domain is infinite means it's an infinite object, which obviously won't work for a programming language. In L4, there is an imperative-looking language for specifying such functions.

The next three sections contain complete formal contract model definitions, with Section **??** extending the model defined in Section **??**, and Section

?? extending the model defined in Section ??. Section ?? is currently the most complete writeup of the L4 semantics. Our intention was to have each of those sections be independent of the others, but we've only succeeded partially in doing that.

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

2 Events, Time, Traces, Finite State Contracts

This section defines a complete-but-limited model of contracts, called simple contracts, and also gives definitions that will be used for the full definition of contracts in Section ??.

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 or hours. A time stamp is a natural number that we think of as being in units time unit.

An event E is composed of an action action_E, a role role_E, a time stamp $\mathsf{timestamp}_E$, and optionally some parameters $\mathsf{actionParams}_E$ (but parameters will not be introduced until Section ??). The actions and roles are fixed finite sets. In this first version of the L4 mathematical model, there is exactly one participant of each role.

All events in the real world are modelled as instances of actions. A special role Env is used to model actions that have no subject. Each event is instantaneous (a real world event with duration is modelled by two such instantaneous events, for the start and end), and its time stamp t means that the modelled real-world event happened between t and $t+1$ time units since the start of the contract. There can be multiple events in a contract execution with time stamp t, and the formal model cannot assign more-precise time units to them, but the formal model does have a total order of all the events with timestamp t. When we need to model two real-world events as truly-simultaneous, we use one event to model their cooccurrence (section with example to be added below).

There can be multiple events in a contract execution with timestamp t, and the formal model can't assign more-precise time units to them, but the formal model does have a total order of all the events with timestamp t. When we need to model two truly-simultaneous informal events, we use one event that models their cooccurrence.

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 sections, one of which is designated the start section, 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 $\langle \mathsf{breaches}(X), \mathsf{Env}, t\rangle$

Between any two events in a trace, the contract is in some global state G which consists of at least a section $\mathsf{section}_G$ and a time stamp $\mathsf{entranceTime}_G$ (in Section ??, global variables will be added).

A contract has a finite directed edge-labeled multigraph[1] which we might call its map; the nodes are the sections, and each directed edge, which we will call an action rule, is labeled with an action. The map is the part of the contract that is easy to visualize. Some notation:

- For r a role, an r-action rule is an action rule whose role is r.
- For a an action, an a-event (respectively a-action rule) is an event (respectively action rule) whose action is a.
- For s a section, the incoming s-action rules (outgoing s-action rules) are the action rules (edges) coming into (going out of) s.

Every action rule is one of the following four types. They will be explained in more detail in the next section.

- A may-next action rule defines permitted events.

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

- A relievable must-next action rule defines the most used kind of obligated events. These are obligations that are relieved by the performance of a permitted event *by some other* agent.
- A must-next action rule defines the strongest kind of obligated events.
- An Env action rule defines a transition that is initiated by the environment.

Note that the events defined by relievable must-next action rules and must-next action rules are also considered permitted events. That completes the definition of the finite directed graph "map" view of a contract.

We say that an action rule c 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 **??** when we add event parameters.

Each action rule c is also associated with a relation $\mathsf{connectionGuard}_c(\cdot)$ called its action rule guard.[2] For simple contracts, it is just a relation on time stamps, and an action rule c is enabled upon entering a global state with time stamp t iff $\mathsf{connectionGuard}_c(t)$ is true.[3]

Each action rule c is also associated with a deadline function $\mathsf{deadline}_c(\cdot)$, which yields a deadline. $\mathsf{deadline}_c(t)$ is either a time stamp after t, or the special element $\underline{\infty}$. The deadline for an action rule is when:

- an enabled may-next action rule (a kind of permission) expires[4].
- an enabled must-next action rule (the strong form of obligation) causes a breach by role_c[5] if a compatible event has not been performed by the deadline.
- an enabled relievable must-next action rule (the weak form of obligation) causes a possibly-joint breach by role_c if a compatible event has not been performed by its deadline **and** no other permitted event is performed by its deadline.

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

[2] But note that in L4 programs, the relation may often be the trivial always-true relation.

[3] Currently, LSM examples are written assuming the action rule guards of a section s's action rules get evaluated only once upon entering the section. 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.

[4] Todo: expires should probably be a defined term.

[5] Which recall, in this formal model means a transition to the state $\mathsf{breached}(\{\mathsf{role}_c\})$

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

The action rule guards must satisfy the following conditions, which would be statically verified in a contract-definition language. We give the simple contracts definitions here, but these conditions will be used in Section **??** as well.

unambiguous absolute obligation condition: For every time stamp t, if some action rule guard of a must-next action rule evaluates to true (at t) then every other action rule 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 action rules's action rule guards evaluates to true (at t) then any other relievable must-next action rules for r evaluate to false (at t).~~

breach or somewhere to go condition: If it is possible for all the enabled non-Env action rules to expire simultaneously, without causing a breach (which entails that there are no enabled must-next action rules or relievable must-next action rules) then there must be an Env action rule with deadline ∞.

2.1 Execution for simple contracts

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

G_0 is $\langle \mathsf{startsection}, 0 \rangle$. Let $i \geq 0$, and assume execution is defined up to entering G_i. To reduce notational clutter, let us use the aliases:

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

Case 1: There is some enabled must-next action rule c in G. If there is any other enabled action rule, then this contract (not just this trace) violates the unambiguous absolute obligation condition, and so is invalid.[6]

- If E is compatible with c and E happens within c's deadline, then the next state must be target_c.[7] This means E fulfilled the obligation created by c.

[6]Recall that a language (tool) for simple contracts will verify that such a thing can't happen.

[7]i.e. if $t' \leq \mathsf{deadline}_c(t)$ then $s' = \mathsf{target}_c$.

- Otherwise, role_e must be r and E must be $\mathsf{breachEvent}(r, \mathsf{deadline}_c(t) + 1)$.

Case 2: **This is rewritten case requires a modification to the breach or somewhere to go condition.**

The are no enabled absolute (strong) obligations. This means that any enabled permitted action, enabled weakly-obligated action, or enabled Env action can occur in the next event. If any of those actions occur next, while the associated deadline function is true, then nobody breaches the contract, and there is nothing more to say. So assume otherwise.

If there is at least one permitted action that has deadline ∞, then there also cannot be a breach, but the contract may now be stuck forever. So assume that doesn't happen either. Then at least one party breaches the contract.

Suppose that a participant p has at least one enabled weak obligation. For each enabled rule r, there is a time t_r at which all of p's permitted actions are expired forever. If the breach happens at t_r or before, then p must be one of the breaching parties.

3 Infinite State with Global Variables

We introduce a set of basic datatypes $\mathbb{T}$, which includes at least $\mathbb{B}, \mathbb{N}$, and $\mathbb{Z}$. Add to the definition of contract a fixed finite set of typed global vars. The global vars are ordered, so we may describe their collective types as a single tuple gvartypes $\in \mathbb{T}^*$.

Add to the definition of global state an assignment of values to the global vars. We'll call such an assignment a global vars assignment. A particular global vars assignment initvals for the values of the global vars in the unique start section is required for a contract; thus, our a technical definition of a contract is fully-instantiated, without parameters. Thus, for example, there is no contract representation of *the* Y-Combinator SAFE startup financing agreement, but there is a contract representation of every fully instantiated signed instance of it. This is not a restriction: any contract-definition language, such as L4, will really be a contract-template definition language. Making contract parameters part of the mathematical model at this point would only serve to make the model more cumbersome.[8]

[8] Later, if we need to write in LaTeX about composing contracts, we may introduce a

The event definition receives the following generalizations:

- Each action a additionally has a global vars transform, denoted $\mathsf{transform}_a$, which is a function from gvartypes $\times$ timestamps to gvartypes.
- The definition of the action rule guard of an a-action rule is generalized: it may now depend on the values of the global vars; i.e. it is now a relation on timestamps $\times$ gvartypes.

Now a action rule guard is a relation on timestamps $\times$ gvartypes, and an s-action rule c is <u>enabled</u> upon entering a global state $\langle s, t, \tau \rangle$ iff $\mathsf{connectionGuard}_c(t, \tau)$ is true.

The three named conditions on action rule guards are updated as follows. For every section s:

<u>unique unrelievable obligation condition</u>: For every global state G whose (local) section is s, if the action rule guard of one of s's must-next action rules evaluates to true (on G) then every other action rule guard of s evaluates to false.

~~role-unique relievable obligations condition: For every role r and global state G whose (local) section is s, if the action rule guard of one of s's relievable must-next action rules with role r evaluates to true (on G) then the action rule guard of every other of s's relievable must-next action rules with role r evaluates to false.~~

<u>breach or somewhere to go condition</u>: If it is possible for all the enabled non-Env action rules to expire simultaneously, without causing a breach (which entails that there are no enabled must-next action rules or relievable must-next action rules) then there must be an Env-action rule with deadline ∞.

Note (probably to move to some other section or document): it will often be the case in a contract-definition language that we simultaneously define an action a and a section JH_a (for "a Just Happened", to fit its literal meaning). In this case, the incoming JH_a-action rules are exactly the set of a-action rules. As a convenience, a contract-definition language will likely allow the outgoing JH_a-action rules to depend directly on a's parameters (that is, for the action rule guard 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 vars that are only used by a and JH_a; a uses $\mathsf{transform}_a$ (recall, its global vars transform) to save its parameter values to those new global vars, so that the outgoing JH_a-action rules can then refer to them.

contract-template mathematical model.

3.1 Execution

Since Subsection **??** is short, we'll repeat essentially the entire definition of execution for simple contracts here, rather than say how to modify it.

Let $E_1, E_2, \ldots$ be a finite or infinite trace (recall: a sequence of events), as defined in Section **??**. Let G_i be the global state that follows E_i for each i. A contract starts in its start section, with initial global vars assignment given by initvals.

G_0 is $\langle \mathsf{startsection}, 0, \mathsf{initvals} \rangle$. Let $i \geq 0$, and assume execution is defined up to entering G_i. To reduce notational clutter, let us use the aliases:

$$G = \langle s, t, \sigma \rangle = G_i \qquad E = \langle a, r, t' \rangle = E_i \qquad G' = \langle s', t', \sigma' \rangle = G_{i+1}$$

Case 1: There is some enabled must-next action rule c in G. If there is any other enabled action rule, then this contract (not just this trace) violates the unique unrelievable obligation condition, and so is invalid.[9]

- If E is compatible with c and E happens within c's deadline ($t' \leq \mathsf{deadline}_c(t)$), then the next state s' must be target_c, and σ' must be $\mathsf{transform}_a(t, \sigma)$. This means E fulfilled the obligation created by c.
- Otherwise, role_e must be r and E must be $\mathsf{breachEvent}(r, \mathsf{deadline}_c(t) + 1)$ and $\sigma' = \sigma$.

Case 2: There is no enabled must-next action rule in G. From the set of enabled may-next action rules of s **and** the set of enabled relievable must-next action rules in G, compute the deadline for each, and discard the action rules whose deadline has passed by the time E happens[10]; let T_p be the resulting set of action rules. From the set of enabled relievable must-next action rules in G, compute the deadline for each, and discard the action rules *whose deadline is not the unique minimal* time stamp t^* *within that set*; let T_o be the resulting set, and let R be $\{\mathsf{role}_c \,|\, c \in T_o\}$. Then E is either:

- An event compatible with some action rule e in T_p. And in this case the next state s' must be target_c, and σ' must be $\mathsf{transform}_a(t, \sigma)$
- $\mathsf{breachEvent}(R, t^*)$.[11] This means that all of the roles whose enabled relievable must-next action rule expire earliest (at t^*) are jointly responsible for the breach.

[9] Recall that a language (tool) for contracts will verify that such a thing can't happen.
[10] i.e. discard c if $\mathsf{deadline}_c(t) > t'$.
[11] Obviously not possible if R is empty

The breach or somewhere to go condition ensures that one of those two cases will apply. In particular, it implies that at least one of T_p or R is nonempty.

4 Event Parameters and Schema

Add to the definition of contract an assignment of types ($\mathbb{T}$-tuples) 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 $\mathsf{paramtypes}_a$.

Each a-action rule c gets assigned an event schema called $\mathsf{eventschema}_c$. An event schema is a set of events that have the same action. We may think of an event schema as a function from gvartypes $\times$ timestamps to a set of a-events (for some fixed a). Equivalently, it is a relation on gvartypes $\times$ timestamps $\times$ a-events, and that is likely how it will be represented in a contract-definition language.

Non-singleton event schema are most useful for an infinite or large choice of actions (and, in the case of Env-events, for infinite or large nondeterminism).

event schema make it necessary to extend the definition of compatible from its previous type event $\times$ actionrule to (globalstate $\times$ event) $\times$ actionrule. We say that an action rule c is compatible with $\langle G, E\rangle = \langle\langle s, t, \sigma\rangle, \langle a, r, t', \tau\rangle\rangle$ iff c is an outgoing s-action rule with action a and role r, and E is in $\mathsf{eventschema}_c(\sigma, t)$.

The three named conditions on action rule guards are the same as before, but we add one more. We now have both event schema and action rule guards as ways of constraining when an action rule can be traversed. To reduce that redundancy we require, for every section s:

nonempty event schema for enabled action rules : For every global state G whose (local) section is s, any enabled s-action rule c must have $\mathsf{eventschema}_c$ nonempty (at G).

4.1 Execution

Again, we elaborate the previous definition, from Subsection **??**, of execution of a contract on a trace, but we repeat all the parts from before.

Let $E_1, E_2, \ldots$ be a finite or infinite trace. Let G_i be the global state that follows E_i for each i. A contract starts in its start section, with initial global vars assignment given by initvals.

G_0 is $\langle \mathsf{startsection}, 0, \mathsf{initvals}\rangle$. Let $i \geq 0$, and assume execution is defined up to entering G_i. To reduce notational clutter, let us use the following aliases, and note that we have added a forth component τ to the event; τ must be of type $\mathsf{paramtypes}_a$.

$$G = \langle s, t, \sigma\rangle = G_i \qquad E = \langle a, r, t', \tau\rangle = E_i \qquad G' = \langle s', t', \sigma'\rangle = G_{i+1}$$

Case 1: There is some enabled must-next action rule c in G. If there is any other enabled action rule, then this contract (not just this trace) violates the unique unrelievable obligation condition, and so is invalid.[12]

- If E is compatible with c and E happens within c's deadline ($t' \leq \mathsf{deadline}_c(t)$), then the next state s' must be target_c, and σ' must be $\mathsf{transform}_a(t, \sigma)$. This means E fulfilled the obligation created by c.
- Otherwise, role_e must be r and E must be $\mathsf{breachEvent}(r, \mathsf{deadline}_c(t) + 1)$ and $\sigma' = \sigma$.

Case 2: There is no enabled must-next action rule in G. From the set of enabled may-next action rules of s **and** the set of enabled relievable must-next action rules in G, compute the deadline for each, and discard the action rules whose deadline has passed by the time E happens[13]; let T_p be the resulting set of action rules. From the set of enabled relievable must-next action rules in G, compute the deadline for each, and discard the action rules *whose deadline is not the unique minimal* time stamp t^* *within that set*; let T_o be the resulting set, and let R be $\{\mathsf{role}_c \mid c \in T_o\}$. Then E is either:

- An event compatible with some action rule e in T_p. And in this case the next state s' must be target_c, and σ' must be $\mathsf{transform}_a(t, \sigma)$
- $\mathsf{breachEvent}(R, t^*)$.[14] This means that all of the roles whose enabled relievable must-next action rule expire earliest (at t^*) are jointly responsible for the breach.

The breach or somewhere to go condition ensures that one of those two cases will apply. In particular, it implies that at least one of T_p or R is nonempty.

[12] Recall that a language (tool) for contracts will verify that such a thing can't happen.
[13] i.e. discard c if $\mathsf{deadline}_c(t) > t'$.
[14] Obviously not possible if R is empty

5 May-Later and Must-Later

VERY WIP

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

We have so far been noncommittal about what types are available for global vars.We will see later that the types strongly affect expressiveness. 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 sections).