L4 / Linear State Machines Formal Model

Legalese.com

November 23, 2017

NTS and todo:

- State $\to$ Section and Edge $\to$ Connection
- Introduce a language for defining nonsingleton event schema? e.g. some $\vec{x} : \tau$ such that $R(\vec{x})$. **No**. That can be part of L4, but in the mathematical model it would just be messy.

Contents

1 How to Read this Document

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 that we will use to define the semantics of our formal contracts language L4. This document also serves as the specification for the formal contract datatype that our natural language generation, formal verification, and visualization software will use.

The the next three sections contain complete formal contract model definitions, with Section 3 extending the model defined in Section 2, and Section 4 extending the model defined in Section 3. Section 4 is currently the most complete writeup of the L4 semantics.

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 3.

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 (but parameters will not be introduced until Section 3) $\mathsf{actionParams}_E$. 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 are modelled as actions**, and a special role Env is used to model actions that have no subject.

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

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

- A may-next connection defines permitted events.
- A relievable must-next connection 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 connection defines the strongest kind of obligated events.

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

We say that a connection 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 3 when we add event parameters.

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

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

Each connection 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 a connection is when:

- an enabled may-next connection (a kind of permission) expires[4].
- an enabled must-next connection (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 connection (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:

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

The connection 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 3 as well.

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

[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 connection guards of a section s's connections 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 $\mathsf{breached}(\{\mathsf{role}_c\})$

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

breach or somewhere to go condition: If it is possible for all the enabled non-Env connections to expire simultaneously, without causing a breach (which entails that there are no enabled must-next connections or relievable must-next connections) then there must be an Env-connection 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 2. 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 connection c in G. If there is any other enabled connection, 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.
- Otherwise, role_e must be r and E must be $\mathsf{breachEvent}(r, \mathsf{deadline}_c(t) + 1)$.

Case 2: **This is correct, but obtuse – over-concise.** There is no enabled must-next connection in G. From the set of enabled may-next connections of s **and** the set of enabled relievable must-next connections in G, compute the deadline for each, and discard the connections whose deadline has passed by the time E happens;[8] let T_p be the resulting set of connections (the p is for permission). Separately, from the set of enabled relievable must-next

[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 $s' = \mathsf{target}_c$.

[8] i.e. discard c if $\mathsf{deadline}_c(t) > t'$.

connections in G, compute the deadline for each, and discard the connections *that do not expire until after the unique minimal expiry* time stamp t^* *within the set*; let T_o be the resulting set (the o is for obligation), and let R be $\{\mathsf{role}_c \mid c \in T_o\}$. Then E is either:

- An event compatible with some connection in T_p.
- $\mathsf{breachEvent}(R, t^*)$.[9] In this case means that all of the roles whose enabled relievable must-next connection 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.

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.[10]

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.

[9] Obviously not possible if R is empty

[10] Later, if we need to write in LaTeX about composing contracts, we may introduce a contract-template mathematical model.

- The definition of the connection guard of an a-connection 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 connection guard is a relation on timestamps $\times$ gvartypes, and an s-connection c is enabled upon entering a global state $\langle s, t, \tau \rangle$ iff $\mathsf{connectionGuard}_c(t, \tau)$ is true.

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

unique unrelievable obligation condition: For every global state G whose (local) section is s, if the connection guard of one of s's must-next connections evaluates to true (on G) then every other connection 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 connection guard of one of s's relievable must-next connections with role r evaluates to true (on G) then the connection guard of every other of s's relievable must-next connections with role r evaluates to false.

breach or somewhere to go condition: If it is possible for all the enabled non-Env connections to expire simultaneously, without causing a breach (which entails that there are no enabled must-next connections or relievable must-next connections) then there must be an Env-connection 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-connections are exactly the set of a-connections. As a convenience, a contract-definition language will likely allow the outgoing JH_a-connections to depend directly on a's parameters (that is, for the connection 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-connections can then refer to them.

3.1 Execution

Since Subsection 2.1 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 2. 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 connection c in G. If there is any other enabled connection, then this contract (not just this trace) violates the unique unrelievable obligation condition, and so is invalid.[11]

- 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 connection in G. From the set of enabled may-next connections of s **and** the set of enabled relievable must-next connections in G, compute the deadline for each, and discard the connections whose deadline has passed by the time E happens[12]; let T_p be the resulting set of connections. From the set of enabled relievable must-next connections in G, compute the deadline for each, and discard the connections *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 connection e in T_p. And in this case the next state s' must be target_c, and σ' must be $\mathsf{transform}_a(t, \sigma)$
- $\mathsf{breachEvent}(R, t^*)$.[13] This means that all of the roles whose enabled relievable must-next connection expire earliest (at t^*) are jointly responsible for the breach.

[11] Recall that a language (tool) for contracts will verify that such a thing can't happen.
[12] i.e. discard c if $\mathsf{deadline}_c(t) > t'$.
[13] 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-connection 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 $\times$ connection to (globalstate $\times$ event) $\times$ connection. We say that a connection 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-connection with action a and role r, and E is in $\mathsf{eventschema}_c(\sigma, t)$.

The three named conditions on connection guards are the same as before, but we add one more. We now have both event schema and connection guards as ways of constraining when an connection can be traversed. To reduce that redundancy we require, for every section s:
nonempty event schema for enabled connections : For every global state G whose (local) section is s, any enabled s-connection c must have $\mathsf{eventschema}_c$ nonempty (at G).

4.1 Execution

Again, we elaborate the previous definition, from Subsection 3.1, 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 connection c in G. If there is any other enabled connection, then this contract (not just this trace) violates the unique unrelievable obligation condition, and so is invalid.[14]

- 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 connection in G. From the set of enabled may-next connections of s **and** the set of enabled relievable must-next connections in G, compute the deadline for each, and discard the connections whose deadline has passed by the time E happens[15]; let T_p be the resulting set of connections. From the set of enabled relievable must-next connections in G, compute the deadline for each, and discard the connections *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 connection e in T_p. And in this case the next state s' must be target_c, and σ' must be $\mathsf{transform}_a(t, \sigma)$
- $\mathsf{breachEvent}(R, t^*)$.[16] This means that all of the roles whose enabled relievable must-next connection 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.

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

5 May-Later and Must-Later

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).