

DRAFT

Legal State Machines, L4, and Formal Verification of Contracts

Report on Computational Law Research by [Legalese](#)*

April 20, 2018

Abstract

This report is intended for industry and academics in Computational Law. However, by publication time it should be readable by anyone with an undergraduate level background in computer science or mathematics. We recommend joining [the #dsl channel](#) on [our Slack workspace](#)¹ and introducing yourself if you're planning on spending more than half an hour with this document.

The primary focus of this report is the definition of the unopinionated, unpretentious programming language-independent mathematical model for computational legal contracts that we've developed after a comprehensive review of the literature and months of research. The model, called *Legal State Machines* (LSMs), provides the formal semantics for our prototype open source computational legal contracts DSL L4, but it is intended to be a *necessary substructure of the semantics of any computational legal contracts language that is worth a damn* (and we eagerly invite disputes).² In programming language theory jargon, LSMs are a denotational semantics.

Legalese is not advocating for the adoption of L4 at this time. We would strongly prefer to join forces with another open source computational contracts DSL project. Our main opinion on this matter is that whatever DSL we align ourselves with, it should have a fragment that fully supports LSMs.

*Contact: dustin.wehr@gmail.com or collective@legalese.com

¹<https://legalese.slack.com>

²L4's typesystem (Section 5.1), though we are quite proud of it, is an example of a feature that does not meet this high standard. It is plausible that the complication it introduces, when in the presence of other optional language features that (the current version of) L4 does not have, makes its inclusion unjustified.

Contents

1	Introduction	2
2	Time, Actors, and Events	3
3	Legal State Machines	4
3.1	Reducing the Abstraction By One Level	7
3.2	Final Specification of LSM	9
4	Formal Verification for LSM	10
4.1	Satisfiability Modulo Theories (SMT) Technology	10
4.2	Symbolic Execution	10
4.3	Exhaustive Model Checking for finite or tamely-infinite state spaces .	10
4.4	Unbounded-Trace Formal Verification with Pre/Postconditions and Invariants	11
4.5	Unexplored: Hard Unbounded-Trace Formal Verification with Inter- active Theorem Proving	11
5	L4: An Experimental but Practical LSM-compatible DSL	11
5.1	Type Checking with Subtyping and Intersection Types	11
6	Related Work	11

1 Introduction

We expect the Computational Law community will develop a number of independent, open source, computational contract DSLs, to suit different tastes and focuses, but we hope that the bulk of the work done by the community will be effectively reusable, particularly in statute and contract libraries, formal verification, and visualization. For this reason, not only have we made our DSL L4 completely free and open source, we have also ensured that none of our work on formal verification of contracts is dependent on the fine details of L4 - only on the much simpler mathematical model of Legal State Machines that one can use L4 (or your own DSL!) to construct.

Sections 2 and 3 define Legal State Machines. Section 4 documents our progress on static analysis for **LSMs**.

Hovering over (resp. clicking on) most terms in **sf font** should show you a popup of (resp. take you to) where the term is defined, where the term is styled like [this](#). This might not work in all PDF viewers.

2 Time, Actors, and Events

We will always be working with a fixed minimal [timeunit](#), which will be one of days, hours, minutes, seconds, etc. It is a parameter of the Legal State Machine (LSM), and should be set to the smallest unit of time that one writes constraints about, or does arithmetic with, in the text of the legal contract one is modelling. A [timestamp](#) is just a nonnegative real number³ that we *think of* as being in units [timeunit](#), which marks the time since the designated start of the LSM execution, which is always 0 by definition. It is worth emphasizing that timestamps are distinct from both DateTimes (some standard for calendar dates with optional within-day times) and TimeDeltas (i.e. durations), both of which are important datatypes in DSLs such as L4. We have found that there is no advantage, and significant disadvantage (when it comes to formal verification), to having DateTimes or TimeDeltas in the mathematical model.

Fix a set \mathbb{D} of basic datatypes, or *sorts*, which includes at least `bool`. The LSM-compatible languages we use will usually include `timestamp`, \mathbb{Z} , and \mathbb{R} as well. These datatypes should be definable types of SMT-LIB. It is important to note that SMT-LIB itself allows for rich datatypes, including recursive datatypes, but also that a computational contracts DSL such as L4 or Ergo can include types beyond those easily definable in SMT-LIB (see Section 5.1).

Fix a finite set of symbols [actor](#), which includes:

- The parties to the contract.
- Any “oracles” that send information to the contract from the environment.
- The special symbol [Code](#), for events that are initiated by the code of the contract.

Before publication of this document, we will likely replace the finite set of party-actors with a finite set of *roles*, and allow for an unbounded number of party-actors in each role; that seems to be necessary to model many blockchain smart contracts in a natural way.

Fix a finite set of symbols [event](#), and for each such e a parameter type assignment $\text{Dom}_e \in \mathbb{D}^*$. Furthermore, partition `event` into three kinds of events:

- party-events, which are actions done by a party-to-the-contract,
- oracle events, which provide information from the environment, and
- deadline events, which are transitions mandated by the contract.

An [event instance](#) is a tuple $\langle e, a, t, \sigma \rangle$ where e is an `event`, a is an `actor`, t is a `timestamp`, and $\sigma \in \text{Dom}_e$. The actor for a deadline event is always `Code`.

³See a few paragraph below for why it is \mathbb{R} and not \mathbb{N} .

DRAFT

Event instances are instantaneous,⁴ occurring at a particular **timestamp**; a real world event with duration is modelled by two such instantaneous **event-instances**, for the start and end of the real world event. That convention is quite flexible; it easily allows modelling overlapping real-world events, for example. We will see in the next section that a sequence of **event-instances** that constitutes a valid execution of an **LSM** requires strictly increasing time stamps. For example, if the **timeunit** is days, then three real-world events that happen in some sequence on the second day would happen at **timestamps** $1, 1 + \epsilon_1, 1 + \epsilon_1 + \epsilon_2$, for some $\epsilon_1, \epsilon_2 > 0$. When we need to model two real-world events as truly-simultaneous, we use one event instance to model their cooccurrence (**todo: example**).

For our intended domain of legal contracts, we are not aware of any cogent criticism of requiring instantaneous event instances with strictly increasing timestamps; and we welcome attempts. An earlier version of the model, in fact, did not require that timestamps are *strictly* increasing, used discrete time, and had what we believe was a very satisfying⁵ justification. However, the justification requires at least another paragraph, and probably several more to adequately defend it. Meanwhile, it offered no advantages in examples, and had one clear disadvantage for formal verification, where the use of integer variables is costly for SMT solvers.⁶

3 Legal State Machines

A Legal State Machine (**LSM**) first of all fixes the definitions of the terms introduced in Section 2: \mathbb{D} , **timeunit**, **actor**, and **event**. It also includes a finite set of symbols situation that must contain at least the symbols:

- fulfilled
- breached_X for each nonempty subset X of $\text{actor} \setminus \{\text{Code}\}$.⁷

An **LSM** M also has an ordered finite set of symbols statevars, and an assignment VDom of a datatype from \mathbb{D} to each. Since the **statevars** are ordered, we can take VDom to be an element of \mathbb{D}^* . M also includes an initial setting **initvals** of its **statevars**.

⁴We might relax this before publication, after discussion with others in the Computational Law community.

⁵Or “elegant”, as unscrupulous researchers put it.

⁶The best explanation we have for this is not simple. It starts with noting that real arithmetic is decidable (real closed fields), but even quantifier free integer arithmetic is undecidable (diophantine equations). This does not necessarily mean that simple uses of integer variables will be costly, but in practice, as of April 2018, it seems to, at least to us outsiders. We are not aware of any particularly-useful decidable restriction of quantifier free combined real/integer arithmetic, and the currently-implemented heuristics, at least in Z3, are easily fooled.

⁷These are breaches and oracle errors analogous to undifferentiated unhandled exceptions in software. Some well-drafted computational contracts might avoid using them completely.

DRAFT

The state space of M is the product set

$$\text{situation} \times \text{VDom} \times \text{timestamp}$$

and a state is an element of the state space.

The remainder and bulk of the definition of an LSM is a mapping from situation to *situation handlers*, and a mapping from event to *event handlers*. An event-handler for event e consists of:

- a destination situation.
- a function statetransform_e of type

$$\text{timestamp}^2 \times \text{VDom} \times \text{Dom}_e \rightarrow \text{VDom}$$

where the two timestamp arguments provided to statetransform_e will always be the timestamps of the previous and next event-instances.

Each situation gets a situation-handler, which for now is just a finite set of event-rules that satisfy the condition unambiguous default condition given below. An event-rule is one of three types: a *party rule*, *oracle rule*, or *deadline rule*. Every event-rule governs the applicability of a unique event by a unique actor.⁸ Every event-rule r has a relation enabled-guard_r on

$$\text{timestamp} \times \text{VDom}$$

where the timestamp argument is the timestamp of the previous event-instance. Frequently in our examples, enabled-guard_r is just the trivial relation true. r is enabled upon entering its parent situation at the timestamp t of the previous event-instance iff enabled-guard_r is true when evaluated at t and the current statevar assignment.

A deadline event rule r governing the applicability of a deadline event e has an additional *deadline function* deadline_r of type

$$\text{timestamp} \times \text{VDom} \rightarrow \text{timestamp}$$

where the timestamp argument is the timestamp of the previous event-instance. r also has a *parameter setter* params_r of type

$$\text{timestamp}^2 \times \text{VDom} \rightarrow \text{Dom}_e$$

where the two timestamp arguments are the timestamps of the previous and next event-instances.

Since deadline event-rules cause an event to occur automatically when the rule activates, we would need to either specify what happens when two such rules activate

⁸In L4, we offer syntax for concisely expressing a set of such rules that apply to different elements of event and actor.

DRAFT

at the same time, or else ensure that can't happen. We take the latter approach.⁹ For now, we adopt a constraint that is stronger than necessary but especially simple:¹⁰ unambiguous default condition: if a situation-handler has multiple event-rules, their enabled-guards must be disjoint relations.

Each party and oracle event rule r governing the applicability of a party or oracle event e has an additional *parameter constraint relation* param-constraint_r on

$$\text{timestamp}^2 \times \text{VDom} \times \text{Dom}_e$$

where the two `timestamp` arguments are the timestamps of the previous and next event-instances. Note that a *parameter setter* is a special case of a *parameter constraint relation*. Because that special case is used fairly frequently, in L4 we allow party and oracle event rules to use the *parameter setter* syntax of deadline event rules instead of their own *parameter constraint relation* syntax.

That completes the definition of a Legal State Machine.

We now define the *well-formed event sequences* of an LSM M , which are a superset of the *traces* of M defined next.

Definition 1 (event-rule compatible with event-instance). An event-rule r is compatible with an event-instance $\langle e, a, t, \sigma \rangle$ iff e and a are the event and actor that r governs the applicability of.

Definition 2 (well-formed event sequence). Fix an LSM M .

A well-formed event sequence of M is a sequence of event-instances E_0, E_1, \dots with strictly-increasing timestamps such that, if $\langle e_i, a_i, t_i, \sigma_i \rangle$ is E_i , then

- The start-situation s_0 of M has an event-rule compatible with E_0
- Either the destination situation s_{i+1} of e_i has an event-rule compatible with E_{i+1} , or else E_i is the final element of the sequence and s_i is fulfilled or breached _{X} for some $X \subseteq \text{actor}$.

Execution of LSMs

Let $\tau = E_1, E_2, \dots$ be a (finite or infinite) well-formed event sequence of M . The starting state G_0 is always $\langle \text{start-situation}, 0, \text{initvals} \rangle$. Let $i \geq 0$ be arbitrary. Assume the sequence is a valid trace up to entering $G_i = \langle s, t, \pi \rangle$. Let E_i be $\langle e, a, t', \sigma \rangle$. We now define the valid values of $G_{i+1} = \langle s', t', \pi' \rangle$:

⁹Because we see no natural way to pick one over the other. Note that the event-rules are not ordered.

¹⁰A closer-to-minimal constraint is: when the enabled-guards of two deadline event-rules are simultaneously true, their deadline functions cannot yield the same timestamp (and then the earlier of the two deadlines is used). We have not yet experienced any desire for the extra leniency, but in case we do, it would be easy to allow it.

DRAFT

- If E_i is a party (resp. oracle) event, then it must be compatible with some party (resp. oracle) event-rule r of s that is enabled in G_i such that $\text{param-constraint}_r$ is true at $\langle t, t', \pi, \sigma \rangle$.
- If E_i is a deadline event, then it must be compatible with the unique¹¹ deadline event-rule r of s that is enabled in G_i such that $\text{deadline}_r(t, \pi) = t'$.
- $\pi' = \text{statetransform}_e(t, t', \pi, \sigma)$.

Any well-formed event sequence where G_i, E_i satisfy the above requirements for all i is a valid trace for M .

3.1 Reducing the Abstraction By One Level

So far, LSM is not a complete language, in the sense that it does not have an abstract syntax tree. In particular, we specified that certain components are mathematical *functions* or *relations*, rather than expressions that define such functions or relations. To recap, those components are as follows, where now we adopt the common convention of writing the types of relations as functions to `bool`.

Initial abstract components of LSM :

- $\text{statetransform}_e : \text{timestamp}^2 \times \text{VDom} \times \text{Dom}_e \rightarrow \text{VDom}$ for each event e .
- $\text{enabled-guard}_r : \text{timestamp} \times \text{VDom} \rightarrow \text{bool}$ for each event-rule r .
- For each party or oracle event-rule r that governs an event e :
 - $\text{param-constraint}_r : \text{timestamp}^2 \times \text{VDom} \times \text{Dom}_e \rightarrow \text{bool}$
- For each deadline event-rule r that governs an event e .
 - $\text{deadline}_r : \text{timestamp} \times \text{VDom} \rightarrow \text{timestamp}$
 - $\text{params}_r : \text{timestamp}^2 \times \text{VDom} \rightarrow \text{Dom}_e$

Despite the lack of concreteness, we saw that there is enough detail that execution can be defined precisely. We *could* stop there, but that would mean leaving out of the coming sections some useful details of formal verification routines that are very likely to be needed in any LSM-compatible DSL.

In this section, we reduce the *description* of the abstraction level significantly, while maintaining the flexibility of being able to define LSM-compatible languages that range from finite state machines to Turing complete languages.¹² Recall that that the set of basic datatypes or *sorts* \mathbb{D} is a parameter to LSM. After this section, we will have a notion of *class of LSM* that depends only on \mathbb{D} and a set \mathcal{F} of functions on the sorts in \mathbb{D} ; that is, each function in such a set \mathcal{F} is of type $S_1 \times \dots \times S_k \rightarrow S_0$

¹¹by the unambiguous default condition

¹²Technically, [beyond Turing complete languages](#), but we don't know of any practical uses of such languages!

DRAFT

for some $k \geq 0$ and some $S_0, \dots, S_k \in \mathbb{D}$. Note that, for the sake of this report, we take individual elements of the sorts \mathbb{D} to be 0-ary functions in \mathcal{F} .

Assume $\text{timestamp} \in \mathbb{D}$.¹³ Surprisingly little is needed in the way of additional definitions. Since $\text{VDom} \in \mathbb{D}^*$ already, the functions enabled-guard_r , $\text{param-constraint}_r$, and deadline_r are already of the required \mathcal{F} -form. The remaining two categories of functions params_r and statetransform_e in the bullet-list above simply get replaced by their point-wise components

- For each deadline event-rule r that governs an event e , and each sort $S_i \in \text{Dom}_e$, a function $\text{params}_r^i : \text{timestamp}^2 \times \text{VDom} \rightarrow S_i$
- For each event e , and each sort $S_i \in \text{VDom}$, a function $\text{statetransform}_e^i : \text{timestamp}^2 \times \text{VDom} \times \text{Dom}_e \rightarrow S_i$.

We go one step further by introducing some minimal structure into the statetransforms , which has a role in [symbolic execution](#).

Definition 3 (statement, statement-implementation). For event e , an [e-statement](#) is one of:

- $x \leftarrow f$ for some $x \in \text{statevars}$ of sort S ¹⁴ and some function f of type $\text{timestamp}^2 \times \text{VDom} \times \text{Dom}_e \rightarrow S$.
- if f then U_1 else U_2 for some function f of type $\text{timestamp}^2 \times \text{VDom} \times \text{Dom}_e \rightarrow \text{bool}$ and finite sets of e -statements U_1, U_2 .

A [statement-implementation](#) of statetransform_e is a set of e -statements that satisfies the [unambiguous statevar-update condition](#), which says: Consider the rooted tree formed by a set of e -statements¹⁵. Any well-typed setting of the statetransform_e parameters yields a value of true or false in the “test” part f of each conditional node if f then U_1 else U_2 . Consider the subtree formed by dropping the appropriate “branch” U_1 (if test is false) or U_2 (if test is true) of each such node, at every level. Then any statevar may occur at most once in that subtree.

We now officially modify the specification of **LSM** to say that each statetransform_e is **statement-implementation** (instead of just being a set of functions of types $\{\text{timestamp}^2 \times \text{VDom} \times \text{Dom}_e \rightarrow \text{VDom}(x) \mid x \in \text{statevars}\}$).

¹³Technically this implies that an extra constraint will be needed to define finite-state machines: roughly, that no statevars of type timestamp are allowed, and the functions in \mathcal{F} cannot depend on their timestamp arguments, even if timestamp appears in the function type.

¹⁴i.e. $\text{VDom}(x) = S$

¹⁵A set of e -statements, including the top-level set, is an internal node whose children are the individual statements. An assignment $x \leftarrow f$ is a leaf node. And a conditional if f then U_1 else U_2 is an internal node with two children U_1 and U_2 .

3.2 Final Specification of LSM

Since the start of Section 3, we've elaborated some details of the initial definition of LSM, to bring it closer to being a useful subject for formal verification. Here we give just the concise final specification, without explanation of the semantics.

Let \mathbb{D} be a set of sorts (i.e. datatypes, i.e. sets). Let \mathcal{F} be a set of functions each of type $S_1 \times \dots \times S_k \rightarrow S_0$ for some $k \geq 0$ and some $S_0, \dots, S_k \in \mathbb{D}$. Then \mathcal{F} determines a class of LSMs that we denote $\text{LSM}(\mathcal{F})$, which are defined as follows. For notational simplicity, and to reduce redundancy, we assume that every sort in \mathbb{D} is in the type of some function in \mathcal{F} , so that we may uniquely determine \mathbb{D} from \mathcal{F} . Let $\text{sorts}(\mathcal{F})$ be that unique determination.

Definition 4 ($\text{LSM}(\mathcal{F})$). Let \mathbb{D} be $\text{sorts}(\mathcal{F})$. An $\text{LSM}(\mathcal{F})$ model M is given the following components. To reduce clutter, we omit the superscript M except at the component's introduction:

- timeunit^M in $\{\text{days, hours, minutes, seconds, } \dots\}$, which we write unsuperscripted in the remainder.
- Finite sets $\text{actor}^M, \text{event}^M, \text{situation}^M$, and statevar^M , which we write unsuperscripted in the remainder.
- A mapping VDom^M from statevar to \mathbb{D} .
- For each $e \in \text{event}$, a type for its parameters $\text{Dom}_e^M \in \mathbb{D}^*$.
- For each $e \in \text{event}$, a statement-implementation statetransform_e , which is a set of e -statements M that satisfies the **unambiguous statevar-update condition**.
- For each $s \in \text{situation}$, a situation-handler handler_s^M , which is a finite subset of event-rule^M that satisfies the **unambiguous default condition**.
- A finite set event-rule^M , with two partitions:
 - *party rules, oracle rules, and deadline rules*
 - $\{\text{handler}_s^M \mid s \in \text{situation}\}$
- An \mathcal{F} function $\text{enabled-guard}_r^M : \text{timestamp} \times \text{VDom} \rightarrow \text{bool}$ for each event-rule r .
- For each party or oracle event-rule r that governs an event e , an \mathcal{F} -function $\text{param-constraint}_r^M : \text{timestamp}^2 \times \text{VDom} \times \text{Dom}_e \rightarrow \text{bool}$
- For each deadline event-rule r that governs an event e .
 - An \mathcal{F} -function $\text{deadline}_r^M : \text{timestamp} \times \text{VDom} \rightarrow \text{timestamp}$
 - For each sort $S_i \in \text{Dom}_e$, an \mathcal{F} -function $(\text{params}_r^i)^M : \text{timestamp}^2 \times \text{VDom} \rightarrow S_i$

4 Formal Verification for LSM

4.1 Satisfiability Modulo Theories (SMT) Technology

Familiarity with SMT is not a prerequisite for this document, but if you are unfamiliar and interested, Microsoft’s [Z3 tutorial](#) is a fine place to start.

4.2 Symbolic Execution

In model checking for expressive models (say, with at least nonlinear integer arithmetic available), usually¹⁶ an infinite-state model is approximated by a finite or tamely-infinite state model, and correctness properties of the approximation model are checked exhaustively, or exhaustively up to a certain maximum computation path length. In Section 4.3 we will consider some cases where approximation is not necessary.

Symbolic execution is a technique for avoiding some of the approximation, especially for avoiding having to approximate unbounded datatypes with bounded ones (e.g. \mathbb{R} approximated by `float`). We do not do this for the sake of more accurate/faithful correctness theorems. Indeed, most of the time software is executed with bounded numeric datatypes anyway, and even when not, no fixed computer can actually compute with arbitrarily large numbers. Rather, we use symbolic execution because, if there is not actually complex math going on (e.g. any cryptographic functions) in a program, but only the use of functions that puts us outside decidable theories, then we should be able to save a lot of time by analyzing computation paths in axiomatically-defined batches.

When we symbolically execute an LSM, we have a choice about whether to use some, all, or none of the model’s `initvals` of its `statevars`. Generally, the more of them used, the faster `state space` exploration will be. For example, in a loan agreement, we could treat an interest rate as an arbitrary element of $(0, 1)$. We would then be proving correctness of the agreement for interest rates that we’ll never use, which of course is perfectly fine if the analysis finishes in a reasonable amount of time. Alternatively, we could use fixed values of the interest rate only, and whenever we use a new fixed interest rate, we simply rerun symbolic execution.

4.3 Exhaustive Model Checking for finite or tamely-infinite state spaces

Suppose the only `statevars` in an LSM M are boolean, and suppose that for every event-rule r none of `enabled-guardr`, `enabled-guardr`, `deadliner`, or `param-constraintr`

¹⁶The term “model checking” is used rather inconsistently.

DRAFT

depend on their `timestamp` arguments. Then M is equivalent for formal verification purposes to a kind of compressed¹⁷ deterministic finite state machine (FSM). If there are no `statevars`, then M is equivalent for formal verification purposes to a normal FSM. Many important properties about FSMs are decidable. If the specification of such a model M is given entirely in terms of a `state space` invariant, then full formal verification can be checked exhaustively by well-known methods.

We may relax the constraint on the functions/predicates that take `timestamp` arguments somewhat, which results in a computation model similar to [Timed Automata](#) with a single clock (but we have not yet mapped the correspondence carefully!).

4.4 Unbounded-Trace Formal Verification with Pre/Postconditions and Invariants

4.5 Unexplored: Hard Unbounded-Trace Formal Verification with Interactive Theorem Proving

5 L4: An Experimental but Practical LSM-compatible DSL

5.1 Type Checking with Subtyping and Intersection Types

6 Related Work

This is thoroughly covered by [Meng's book chapter](#) and Appendix B of the [CodeX whitepaper](#) *Developing a Legal Specification Protocol: Technological Considerations and Requirements*.

¹⁷via the boolean `statevars`