# Legal State Machines, L4, and Formal Verification of Contracts

Report on Computational Law Research by Legalese*

April 25, 2018

(todo: Closure of $\mathcal{F}$ under composition)
(todo: Kill $\mathsf{sorts}_e$ in favour of the one-sort-per-name requirement)
(todo: Maybe kill $\mathsf{sorts}_{\mathsf{state}}$ for the same reason)

**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.

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

[1] https://legalese.slack.com

[2] L4's typesystem (Section 6.1), 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.

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.

# Contents

# 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 and 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 number[3] that we *think of* as being in units timeunit. It denotes 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, clock times, time zones) and TimeDeltas (i.e. durations, which are similar to timeunit aside from coming in more than one unit), 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 almost always include timestamp as well, and will usually include $\mathbb{Z}$ and $\mathbb{R}$. We require that these datatypes are definable types of SMT-LIB. It is important to note that SMT-LIB itself allows for rich datatypes, including recursive datatypes[4], but also that a computational contracts DSL such as L4 or Ergo can include types beyond those easily definable in SMT-LIB (see Section 6.1).

**Note 1.** We will skip specifying that various symbol sets are disjoint. Any two symbol sets that you might expect to be disjoint, are required to be disjoint.

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

[4] Though then quantifier free validity is undecidable, so the solver is incomplete.

**Note 2.** Every typed symbol is used with a unique sort (i.e. element of $\mathbb{D}$), regardless of scope.

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. In our implementation of SAFE in L4, we have an oracle that decides whether a liquidation, equity, or dissolution event has occurred. Generally, we put an event's announcement in the hands of an oracle when every party has an incentive to lie about it.
- 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, since 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 $\mathsf{sorts}_e \in \mathbb{D}^*$. 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 $\sigma \in \mathsf{sorts}_e$. The actor for a deadline event is always Code.

Event instances are instantaneous,[5] 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 satisfying[6] justification. However, the justification requires at least

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

[6] Or "elegant", as unscrupulous researchers put it.

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.[7]

# 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 actor$\backslash\{$Code$\}$.[8]

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

The state space of $M$ is the product set

$$\text{situation} \times \text{sorts}_{\text{state}} \times \text{timestamp}$$

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:

- dest$_e$ $\in$ situation.
- a function statetransform$_e$ of type

$$\text{timestamp}^2 \times \text{sorts}_{\text{state}} \times \text{sorts}_e \to \text{sorts}_{\text{state}}$$

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 deadline condition given below. The set event-rule

[7] The 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.

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

is partitioned into: party-rules, oracle-rules, and deadline-rules. Every event-rule governs the applicability of a unique event by a unique actor.[9] Every event-rule $r$ has a relation enabled-guard$_r$ on

$$\text{timestamp} \times \text{sorts}_{\text{state}}$$

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.

**Note 3** (redundancy of enabled-guard$_r$ for party/oracle rules)**.** With respect to the execution semantics (Section 3.1), it is easy to eliminate enabled-guard$_r$ for party and oracle event-rules $r$ (though not for deadline event-rules), by conjoining enabled-guard$_r$ to param-constraint$_r$. We will do this for some formal verification purposes where the simplification of the model outweighs the cost (in development time) of interpreting errors and traces. From the practical software engineering perspective, on the other hand, we have found it is very natural to split the constraint on when an oracle/party event-rule can apply into

- The (usually-maximal) part that depends only on the current state. This is enabled-guard$_r$.
- The part that depends on the next event-instance. This is param-constraint$_r$.

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

$$\text{timestamp} \times \text{sorts}_{\text{state}} \to \text{timestamp}$$

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

$$\text{timestamp} \times \text{sorts}_{\text{state}} \to \text{sorts}_e$$

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 at the same time, or else ensure that can't happen. We take the latter approach.[10]

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

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

For now, we adopt a constraint that is stronger than necessary but especially simple:[11] unambiguous deadline condition: if a situation-handler has multiple deadline event-rules, then their enabled-guards must be disjoint relations.

As a very basic liveness condition, we want that an LSM can't get stuck in a state where no party or oracle event-rules will ever be in applicable again, and where there is no deadline event-rule that will trigger. We call this the never-stuck condition.

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

$$\mathsf{timestamp}^2 \times \mathsf{sorts}_{\mathsf{state}} \times \mathsf{sorts}_e$$

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 $E_0, E_1, \ldots$ 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 $X \subseteq$ actor.

## 3.1 Execution of LSMs

Let $\tau = E_1, E_2, \ldots$ be a (finite or infinite) well-formed event sequence of $M$. The starting state $G_0$ is always $\langle$start-situation, 0, 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 $G_{i+1} = \langle s', t', \pi' \rangle$:

[11] A 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.

- If $E_i$ is a party (resp. oracle) event-instance, then it must be compatible with some party (resp. oracle) event-rule $r$ of $s$ that is enabled in $G_i$ such that $\mathsf{param\text{-}constraint}_r$ is true at $\langle t, t', \pi, \sigma \rangle$.
- If $E_i$ is a deadline event-instance, then it must be compatible with the unique[12] deadline event-rule $r$ of $s$ that is enabled in $G_i$ such that $\mathsf{deadline}_r(t, \pi) = t'$. Moreover, $\sigma$ must be $\mathsf{psetter}_r(t, \pi)$.
- $\pi' = \mathsf{statetransform}_e(t, t', \pi, \sigma)$.

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

## 3.2 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** :

- $\mathsf{statetransform}_e : \mathsf{timestamp}^2 \times \mathsf{sorts}_{\mathsf{state}} \times \mathsf{sorts}_e \to \mathsf{sorts}_{\mathsf{state}}$ for each event $e$.
- $\mathsf{enabled\text{-}guard}_r : \mathsf{timestamp} \times \mathsf{sorts}_{\mathsf{state}} \to \mathsf{bool}$ for each event-rule $r$.
- For each party or oracle event-rule $r$ that governs an event $e$:
  - $\mathsf{param\text{-}constraint}_r : \mathsf{timestamp}^2 \times \mathsf{sorts}_{\mathsf{state}} \times \mathsf{sorts}_e \to \mathsf{bool}$
- For each deadline event-rule $r$ that governs an event $e$.
  - $\mathsf{deadline}_r : \mathsf{timestamp} \times \mathsf{sorts}_{\mathsf{state}} \to \mathsf{timestamp}$
  - $\mathsf{psetter}_r : \mathsf{timestamp} \times \mathsf{sorts}_{\mathsf{state}} \to \mathsf{sorts}_e$

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.[13] 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 $S_1 \times \cdots \times S_k \to S_0$

[12] by the unambiguous deadline condition

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

for some $k \geq 0$ and some $S_0, \ldots, S_k \in \mathbb{D}$. 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 $\mathsf{timestamp} \in \mathbb{D}$.[14] Surprisingly little is needed in the way of additional definitions. Since $\mathsf{sorts}_{\mathsf{state}} \in \mathbb{D}^*$ already, the functions $\mathsf{enabled\text{-}guard}_r$, $\mathsf{param\text{-}constraint}_r$, and $\mathsf{deadline}_r$ are already of the required $\mathcal{F}$-form. The remaining two categories of functions $\mathsf{psetter}_r$ and $\mathsf{statetransform}_e$ 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 event-rule-parameter $x_i$ of sort $S_i \in \mathsf{sorts}_e$, a function $\mathsf{psetter}_r^{x_i} : \mathsf{timestamp} \times \mathsf{sorts}_{\mathsf{state}} \to S_i$
- For each event $e$, and each sort $S_i \in \mathsf{sorts}_{\mathsf{state}}$, a function $\mathsf{statetransform}_e^i : \mathsf{timestamp}^2 \times \mathsf{sorts}_{\mathsf{state}} \times \mathsf{sorts}_e \to S_i$.

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

**Definition 3** (statement, conditional-tree)**.** For event $e$, an *e*-statement is one of:

- $x \leftarrow f$ for some $x \in$ statevars of sort $S$[15] and some function $f$ of type $\mathsf{timestamp}^2 \times \mathsf{sorts}_{\mathsf{state}} \times \mathsf{sorts}_e \to S$.
- if $f$ then $U_1$ else $U_2$ for some function $f$ of type $\mathsf{timestamp}^2 \times \mathsf{sorts}_{\mathsf{state}} \times \mathsf{sorts}_e \to$ bool and finite sets of $e$-statements $U_1, U_2$.

A conditional-tree of $\mathsf{statetransform}_e$ is a set of $e$-statements that satisfies the unambiguous statevar-update which says: Consider the rooted tree formed by the set of $e$-statements $\mathsf{statetransform}_e$[16]. Any well-typed setting of the $\mathsf{statetransform}_e$ 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 $\mathsf{statetransform}_e$ is a conditional-tree (instead of just being a set of functions of types $\{\mathsf{timestamp}^2 \times \mathsf{sorts}_{\mathsf{state}} \times \mathsf{sorts}_e \to \mathsf{sorts}_{\mathsf{state}}(x) \mid x \in \mathsf{statevars}\}$).

[14] Technically 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.

[15] i.e. $\mathsf{sorts}_{\mathsf{state}}(x) = S$

[16] The tree: A set of $e$-statements is an internal node whose children are the individual statements, with the top-level set $\mathsf{statetransform}_e$ itself being the root of the tree. An assignment $x \leftarrow f$ is a leaf node. A conditional if $f$ then $U_1$ else $U_2$ is a second kind of internal node with two children $U_1$ and $U_2$ (which, observe, are themselves internal nodes of the first kind).

## 3.3 Basic Specification: LSM($\mathcal{F}$)

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 $S_1 \times \cdots \times S_k \to S_0$ for some $k \geq 0$ and some $S_0, \ldots, S_k \in \mathbb{D}$. Then $\mathcal{F}$ determines a class of LSMs that we denote 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 sorts($\mathcal{F}$) be that unique determination. We also assume timestamp $\in \mathbb{D}$.

Backpeddling on "without explanation of the semantics" for a moment: Note/recall that when a function in the following definition has a type starting with timestamp$^2$, that is the timestamp of the most recent event-instance and the timestamp of a candidate next event-instance. When the type starts with just timestamp, it is of the most recent event-instance.

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

- timeunit$^M$ in {days, hours, minutes, seconds, ...}.
- Finite sets actor$^M$, event$^M$, situation$^M$, and statevar$^M$.
- A mapping $\text{sorts}_{\text{state}}{}^M$ from statevar to $\mathbb{D}$.
- For each $e \in$ event, a type for its parameters $\text{sorts}_e^M \in \mathbb{D}^*$.
- For each $e \in$ event, a conditional-tree statetransform$_e$, which is a set of $e$-statements$^M$ that satisfies the unambiguous statevar-update condition.
- For each $s \in$ situation, a situation-handler $\text{handler}_s^M$, which is a finite subset of event-rule$^M$ that satisfies the unambiguous deadline 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 $\text{enabled-guard}_r^M$ : timestamp $\times$ $\text{sorts}_{\text{state}} \to$ bool for each event-rule $r$.
- For each party or oracle event-rule $r$ that governs an event $e$:
  - An $\mathcal{F}$-function $\text{param-constraint}_r^M$ : timestamp$^2$ $\times$ $\text{sorts}_{\text{state}} \times \text{sorts}_e \to$ bool
- For each deadline event-rule $r$ that governs an event $e$.
  - An $\mathcal{F}$-function $\text{deadline}_r^M$ : timestamp $\times$ $\text{sorts}_{\text{state}} \to$ timestamp
  - For each $r$-parameter $x_i$ of sort $S_i \in \text{sorts}_e$, an $\mathcal{F}$-function $(\text{psetter}_r^i)^M$ : timestamp $\times$ $\text{sorts}_{\text{state}} \to S_i$

# 4 Automated 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 Checking the Basic Correctness Conditions

We will define here how we use an SMT solver to verify the unambiguous deadline condition, never-stuck condition, and unambiguous statevar-update condition.

## 4.3 LSM($\mathcal{F}$) Extension: Adding Pre/Postconditions and Invariants

## 4.4 Symbolic Execution

In model checking for expressive models (say, with at least nonlinear integer arithmetic available), usually[17] 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 4.5 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.

### 4.4.1 Initial values, fixed or abstract?

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

[17] The term "model checking" is used rather inconsistently.

correctness of the agreement for interests 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.

### 4.4.2 Timeless Normal Form for LSM($\mathcal{F}$)

We define a normal form for LSM($\mathcal{F}$) which is not literally "timeless", but removes the special significance of timestamp from the model. This is just to make the symbolic execution algorithm simpler.

First, and unrelated to the following timestamp-related transformations (but with the same goal of simplifying symbolic execution), eliminate enabled-guard$_r$ for each party/oracle event-rule $r$, as described in Note 3.

- Introduce a new statevar $t_{\mathsf{last}}$ of sort timestamp with initial value 0.[18]
- Add a parameter $t_{\mathsf{next}}$ of sort timestamp to every event[19]
- Add $t_{\mathsf{last}} \leftarrow t_{\mathsf{next}}$ to statetransform$_e$ for each $e$.
- Replace psetter$^i_r$ with psetter$^i_r(t_{\mathsf{last}}, \cdot)$. Move deadline$_r(t_{\mathsf{next}}, \cdot)$ into psetter$^{t_{\mathsf{next}}}_r$ for each deadline event-rule $r$.
- Add $>_{ts}$ :timestamp $\times$ timestamp $\to$ bool to $\mathcal{F}$.
- Conjoin ($\wedge$) $t_{\mathsf{next}} >_{ts} t_{\mathsf{last}}$ to param-constraint$_r$ for each party/oracle event-rule $r$.

We obtain the following slightly simplified LSM definition:

**Definition 5** (timeless-LSM($\mathcal{F}$))**.** Let $\mathbb{D}$ be sorts($\mathcal{F}$). The relation $>_{ts}$ :timestamp $\times$ timestamp $\to$ bool must be in $\mathcal{F}$ (and so timestamp must be in $\mathbb{D}$). A timeless-LSM($\mathcal{F}$) model $M$ is given by the following components.

- timeunit in {days, hours, minutes, seconds, ...}.
- Finite sets actor, event, situation, and statevar.
- A mapping sorts$_{\mathsf{state}}$ from statevar to $\mathbb{D}$.
- For each $e \in$ event, types for its parameters sorts$_e \in \mathbb{D}^*$.
- For each $e \in$ event, a conditional-tree statetransform$_e$, which is a set of $e$-statements that satisfies the unambiguous statevar-update condition.
- For each $s \in$ situation, a situation-handler handler$_s$, which is a finite subset of event-rule that satisfies the unambiguous deadline condition.
- A finite set event-rule, with two partitions:
  - party-rules, oracle-rules, and deadline-rules
  - {handler$_s$ | $s \in$ situation}
- For each party or oracle event-rule $r$ that governs an event $e$

[18] i.e. sorts$_{\mathsf{state}}(t_{\mathsf{last}}) =$ timestamp and initvals$(t_{\mathsf{last}}) = 0$.

[19] i.e. sorts$_e(t_{\mathsf{next}}) =$ timestamp for every event $e$.

  - An $\mathcal{F}$-function $\mathsf{param\text{-}constraint}_r : \mathsf{sorts}_{\mathsf{state}} \times \mathsf{sorts}_e \to \mathsf{bool}$
- For each deadline $\mathsf{event\text{-}rule}$ $r$ that $\mathsf{governs}$ an $\mathsf{event}$ $e$.
  - An $\mathcal{F}$ function $\mathsf{enabled\text{-}guard}_r : \mathsf{sorts}_{\mathsf{state}} \to \mathsf{bool}$ for each $\mathsf{event\text{-}rule}$ $r$.
  - For each sort $S_i \in \mathsf{sorts}_e$, an $\mathcal{F}$-function $\mathsf{psetter}_e^i : \mathsf{sorts}_{\mathsf{state}} \to S_i$

### 4.4.3 Symbolic Execution for LSM($\mathcal{F}$)

WARNING: This section is not ready for reading yet

$v$ ranges over $\mathsf{sexec\text{-}variables}$.

The set of $\mathcal{F}$-terms is defined by:

- $\mathsf{sexec\text{-}variables}$ are $\mathcal{F}$-terms.
- If $t_1, \ldots, t_k$ are $\mathcal{F}$-terms and $f$ is the function symbol for some $k$-ary function in $\mathcal{F}$, then $f(t_1, \ldots, t_k)$ is a $\mathcal{F}$-term.

$\psi$ ranges over $\mathcal{F}$-terms of sort $\mathsf{bool}$.

Introduce symbols

- $q_i$ to stand for the $i$-th $\mathsf{statevar}$.
- $x_i^e$ to stand for the $i$-th parameter of $\mathsf{event}$ $e$.
- $x_i^r$ to stand for the $i$-th parameter of $\mathsf{event\text{-}rule}$ $r$.

Throughout, $\Gamma$ is a full mapping from $\mathsf{statevar}$ to $\mathcal{F}$-terms of the correct type. Initially it is $\mathsf{initvals}$.

$P$ is an $\mathcal{F}$-$\mathsf{term}$ of sort $\mathsf{bool}$.

$$\frac{s \in \mathsf{situation} \quad \mathsf{event\text{-}rule} r \in \mathsf{handler}_s \quad \Gamma; P \vdash r[\Gamma] \rightsquigarrow \mathsf{rv}}{\Gamma;\ P \vdash \mathsf{handler}_s \rightsquigarrow \mathsf{rv}}$$

$$\frac{\begin{array}{c}\vec{v} \leftarrow fresh(x^e) \\ P' = P \wedge \psi[\Gamma, x^e \mapsto \vec{v}] \quad P' \text{ consistent} \quad \Gamma;\ P' \vdash \mathsf{handler}_e[\Gamma, x^e \mapsto \vec{v}] \rightsquigarrow \mathsf{rv}\end{array}}{\Gamma;\ P \vdash R \text{ may } (e \text{ where } \psi) \rightsquigarrow \mathsf{rv}}$$

$$\frac{\begin{array}{c}P' = P \wedge \psi[\Gamma] \\ P' \text{ consistent} \quad \Gamma;\ P' \vdash e(\mathsf{psetter}_e^1[\Gamma], \ldots, \mathsf{psetter}_e^k[\Gamma] \rightsquigarrow \mathsf{rv}\end{array}}{\Gamma;\ P \vdash e(\mathsf{psetter}_e^1, \ldots, \mathsf{psetter}_e^k) \text{ provided } \psi \rightsquigarrow \mathsf{rv}}$$

$$\frac{\begin{array}{c}e \in \mathsf{event} \\ X \text{ a set of } e\text{-}\mathsf{statements} \\ (\mathsf{if}\, \psi\, \mathsf{then}\, X_1\, \mathsf{else}\, X_2) \in X \quad P \wedge \psi \text{ consistent} \quad \Gamma; P \wedge \psi \vdash X \cup X_1 \rightsquigarrow \mathsf{rv}\end{array}}{\Gamma;\ P \vdash X \rightsquigarrow \mathsf{rv}}$$

$$\frac{\begin{array}{c} e \in \mathsf{event} \\ X \text{ a set of } e\text{-}\mathsf{statements} \\ (\mathsf{if}\,\psi\,\mathsf{then}\,X_1\,\mathsf{else}\,X_2) \in X \end{array} \quad P \wedge \neg\psi \text{ consistent} \quad \Gamma; P \wedge \neg\psi \vdash X \cup X_2 \rightsquigarrow \mathsf{rv}}{\Gamma;\ P \vdash X \rightsquigarrow \mathsf{rv}}$$

$$\frac{\begin{array}{c} X \text{ a set of (only) assignment } e\text{-}\mathsf{statement}\mathcal{F}\text{-}\mathsf{terms} \text{ from } \mathsf{handler}_e \\ \Gamma' \text{ is } \Gamma \text{ updated by } X[\Gamma] \end{array} \quad \Gamma'; P \vdash \mathsf{dest}_e \rightsquigarrow \mathsf{rv}}{\Gamma;\ P \vdash X \rightsquigarrow \mathsf{rv}}$$

## 4.5 Exhaustive Model Checking for Finite and Tamely-Infinite State Spaces

Suppose the only statevars in an LSM $M$ are bool or enums, and suppose that for every event-rule $r$ and event $e$ none of $\mathsf{enabled\text{-}guard}_r$, $\mathsf{deadline}_r$, $\mathsf{param\text{-}constraint}_r$, $\mathsf{statetransform}_e$, or $\mathsf{psetter}_r$ depend on their timestamp arguments. Then $M$ is equivalent for formal verification purposes to a kind of compressed[20] 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. For example, 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, allowing constraints that are boolean combinations of atomic formulas of the form

$$(\text{linear fn of } t,t') \ (\leq \mid < \mid =) \ (\text{linear fn of } t,t')$$

where $t$ is the current timestamp and $t'$ is the next. DateTime and TimeDelta literals can be used also, in the higher-level DSL). This results in a computation model close or equivalent to Timed Automata with two clocks, one of which is never reset and the other that is reset after each event-instance. We have not yet proved such a reduction, so take it with a grain of salt.

[20] via the boolean statevars

### 4.6 Unbounded-Trace Formal Verification with Pre/Postconditions and Invariants

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

## 5 Extensions

### 5.1 Unbounded **actor** Set

This is to accommodate smart contracts (aka dapps). It is a **very** tentative proposal.

We introduce to the definition of LSM($\mathcal{F}$) another finite set role, which always include a special symbol unknown. unknown is the role of a party (i.e. ethereum address, in the ethereum dapp context) who has never interacted with the contract before, and who was not known to the parties who initiated the contract. It is thus used at most once by any one party. An event-instance allowed by an event-rule governing unknown has the side effect of assigning a non-unknown role to the party that initiates the event-instance.

## 6 L4: An Experimental but Practical LSM-compatible DSL

### 6.1 Type Checking with Subtyping and Intersection Types

## 7 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.*