

DRAFT

Legal State Machines, the language L4, and Formal Verification of Legal Contracts

Report on Open Source Computational Law Research by [Legalese](https://legalese.com)[‡]

July 1, 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](#)¹ on [our Slack workspace](#)² and introducing yourself if you're planning on spending more than an hour with this document.

The primary focus of this report is the definition of the 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, (tentatively) called *Legal State Machines* (LSMs), provides the formal semantics for our prototype open source computational legal contracts DSL L4. But more importantly, 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).³ In programming language theory jargon, LSMs are a denotational semantics. We will argue that a single-threaded, non-distributed, event-driven computation model is best suited for legal contracts, despite not being the first thing one thinks of for multi-agent computation.

*<https://legalese.com>

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

¹<https://legalese.slack.com/messages/C0SB9HZ1S/>

²<https://legalese.slack.com>

³L4's typesystem (Section 7.2), 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	How to read	3
2	Introduction	3
3	Time, Actors, and Events	3
4	Legal State Machines	5
4.1	LSM Execution	8
4.2	Reducing the Abstraction By One Level	8
4.3	Basic Specification: $\text{LSM}(\mathcal{F})$	10
4.4	$\text{LSM}(\mathcal{F})$ Execution	11
5	Automated Formal Verification for LSM	12
5.1	Satisfiability Modulo Theories (SMT) Technology	12
5.2	Checking the Basic Correctness Conditions	12
5.3	$\text{LSM}(\mathcal{F})$ Extension: Adding Pre/Postconditions and Invariants	13
5.4	Symbolic Execution	13
5.4.1	Initial values, fixed or not?	13
5.4.2	Timeless Normal Form for $\text{LSM}(\mathcal{F})$	14
5.4.3	A Minimal timeless- $\text{LSM}(\mathcal{F})$ -Compatible Language	15
5.4.4	Symbolic Execution for timeless- $\text{LSM}(\mathcal{F})$	16
5.5	Exhaustive Model Checking for Finite and Tamely-Infinite State Spaces	18
5.5.1	Finite Automata / Buchi Automata	18
5.5.2	Timed Automata	18
5.6	Unbounded-Trace Formal Verification with Pre/Postconditions and Invariants	19
5.7	Unexplored: Hard Unbounded-Trace Formal Verification with Inter- active Theorem Proving	19
6	Extensions	19
6.1	Uninitialized <code>statevars</code>	19
6.2	Unbounded <code>actor</code> Set	20
7	L4: An Experimental but Practical LSM-compatible DSL	20
7.1	The Largest Set of Examples	20
7.2	Type Checking with Subtyping and Intersection Types	20
8	Related Work	20

9	Dustin's todo	20
9.1	ASAP	20
9.2	Priority	21
9.3	Low Priority / Questionable	21

1 How to read

2 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 3 and 4 define Legal State Machines. Section 5 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.

3 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 number⁴ 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 [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

⁴See a few paragraph below for why it is \mathbb{R} and not \mathbb{N} .

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 datatypes⁵, but also that a computational contracts DSL such as L4 or Ergo can include types beyond those easily definable in SMT-LIB (see Section 7.2).

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. Also, every typed symbol has a dedicated type (i.e. element of \mathbb{D} for various variables, function types $\mathbb{D}^* \rightarrow \mathbb{D}$ for function symbols), regardless of its scope. We will thus (eventually) not have as much type-assignment notation as is common in programming language research. A user-facing DSL such as L4 or Ergo will lift this restriction.

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

- The parties to the contract, who announce action-events, which are the elements of the event stream most responsible for driving a contract.
- Any “oracles” that send information to the contract from the environment. In our implementation of Y-Combinator’s SAFE in L4, we have an oracle that decides whether a liquidation, equity, or dissolution event has occurred, in case the company fails to announce it.
- The special symbol [Code](#), for events that are initiated by the code of the contract.

Note 2. 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 $\text{type}_e \in \mathbb{D}^*$. 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 $\sigma \in \text{type}_e$. The **actor** for a **deadline event** is always **Code**.

Event instances are instantaneous,⁶ 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

⁵Though then quantifier free validity is undecidable, so the prover is incomplete.

⁶We might relax this before publication, after discussion with others in the Computational Law community.

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 satisfying⁷ 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 provers.⁸

4 Legal State Machines

A Legal State Machine ([LSM](#)) first of all fixes the definitions of the terms introduced in Section 3: `℔`, `timeunit`, `actor`, and `event`. It also includes a finite set of symbols [situation](#) that must contain at least the symbols:

- [fulfilled](#), for handled contract termination.
- [breached_A](#) for each nonempty subset A of `actor \ {Code}`.⁹

Let [terminated](#) be the union of those symbols.

An LSM M also has an ordered finite set of symbols [statevars](#), and an assignment [type_{state}](#) of a datatype from `℔` to each. Since the `statevars` are ordered, we can take [type_{state}](#) to be an element of \mathbb{D}^* . M also includes an initial setting [initvals](#) of its `statevars`, and designates one of the `situations` its [start-situation](#).

The [state space](#) of M is the product set

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

and a [state](#) is an element of the `state space`.

⁷Or “elegant”, as unscrupulous researchers put it.

⁸The 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.

⁹These are breaches and oracle errors analogous to undifferentiated, unhandled exceptions in normal programming languages. Some well-drafted computational contracts might avoid using them completely.

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 ∈ *situation*, the destination *situation* that e always transitions to.
- a function statetransform_e of type

$$\text{timestamp}^2 \times \text{type}_{\text{state}} \times \text{type}_e \rightarrow \text{type}_{\text{state}}$$

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 is partitioned into: *party-rules*, *oracle-rules*, and *deadline-rules*. Every *event-rule* governs the applicability of a unique event (*party-events*, *oracle-events*, and *deadline-events*) by a unique actor.¹⁰ Every *event-rule* r has a relation enabled-guard_r on

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

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* \mathfrak{t} of the previous *event-instance* iff enabled-guard_r is true when evaluated at \mathfrak{t} and the current statevar assignment.

Note 3 (redundancy of enabled-guard_r for party/oracle rules). With respect to the execution semantics (Section 4.1), 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

$$\text{timestamp} \times \text{type}_{\text{state}} \rightarrow \text{timestamp}$$

¹⁰In L4, we offer syntax for concisely expressing a set of such rules that apply to different elements of *event* and *actor*.

where the **timestamp** argument is the **timestamp** of the previous event-instance. r also has a *parameter setter* [psetter_r](#) of type

$$\text{timestamp} \times \text{type}_{\text{state}} \rightarrow \text{type}_e$$

where the **timestamp** argument is the **timestamp** of the previous event-instances. The **event** parameters are completely determined by the rule and the current state since a deadline **event-instance** is created by the contract itself.

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.¹¹ For now, we adopt a constraint that is stronger than necessary but especially simple:¹² [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

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

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

¹¹Because we see no natural way to pick one over the other. Note that the **event-rules** are not ordered.

¹²A 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.

- 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 _{A} for some nonempty $A \subseteq \text{actor}$.

4.1 LSM Execution

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 $G_i = \langle s, \mathbf{t}, \pi \rangle$. Let E_i be $\langle e, a, \mathbf{t}', \sigma \rangle$. We now define the valid values of $G_{i+1} = \langle s', \mathbf{t}', \pi' \rangle$:

- 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 $\text{param-constraint}_r$ is true at $\langle \mathbf{t}, \mathbf{t}', \pi, \sigma \rangle$.
- If E_i is a deadline event-instance, then it must be compatible with the unique¹³ deadline event-rule r of s that is enabled in G_i such that $\text{deadline}_r(\mathbf{t}, \pi) = \mathbf{t}'$. Moreover, σ must be $\text{psetter}_r(t, \pi)$.
- $s' = \text{dest}_e$.
- $\pi' = \text{statetransform}_e(\mathbf{t}, \mathbf{t}', \pi, \sigma)$.

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

4.2 Reducing the Abstraction By One Level

So far, LSM is far from a *formal language* in that it does not have anything like an abstract syntax tree with finitely-many node categories. This is intentional. 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 turn to the common convention of writing the types of relations as functions to `bool`.

Initial abstract components of LSM :

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

¹³by the unambiguous deadline condition

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.¹⁴ We do not go as far as to define a language; that will happen in Section 5.4.3.

Recall 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 $S_1 \times \dots \times S_k \rightarrow S_0$ for some $k \geq 0$ and some $S_0, \dots, S_k \in \mathbb{D}$.

Definition 3 (\mathcal{F}^*). Take \mathcal{F}^* to be the closure of \mathcal{F} under well-typed composition, with the normal meaning. So, it is the superset of \mathcal{F} that contains functions such as the following function of type $S_1 \times S_2 \rightarrow S_1$:

$$a_1 \in S_1, a_2 \in S_2 \mapsto f_1(a_1, f_2(a_2))$$

where $f_1, f_2 \in \mathcal{F}$ and f_2 has type $S_2 \rightarrow S_1$ and f_1 has type $S_1 \times S_1 \rightarrow S_1$.

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}^* ; doing so simply reduces the description length of some definitions.

Surprisingly little is needed in the way of additional definitions. Assume `timestamp` $\in \mathbb{D}$.¹⁶ Since `typestate` $\in \mathbb{D}^*$ already, the functions `enabled-guardr`, `param-constraintr`, and `deadliner` are already elements of \mathcal{F}^* (which is what we want). The remaining two categories of functions in the bullet-list above, `psetterr` and `statetransforme`, simply get replaced by their point-wise components:

- For each deadline event-rule r that governs an event e , and each event parameter x_i^e of sort $S_i \in \text{type}_e$, a function `psetterri` : `timestamp` \times `typestate` $\rightarrow S_i$ from \mathcal{F}^* .¹⁷
- For each event e , and each sort $S_i \in \text{type}_{\text{state}}$, a function `statetransformei` : `timestamp`² \times `typestate` \times `typee` $\rightarrow S_i$ from \mathcal{F}^* .

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

Definition 4 ([statement](#), [conditional-tree](#)). For event e , an [e-statement](#) is one of:

¹⁴Technically, [beyond Turing complete languages](#)¹⁵, but we don't know of any practical uses of such languages!

¹⁶Technically 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.

¹⁷Or we will sometimes write `psetterrxie` when it helps more than clutters.

- $q \leftarrow f$ for some $q \in \text{statevars}$ of sort S^{18} and some function f of type $\text{timestamp}^2 \times \text{type}_{\text{state}} \times \text{type}_e \rightarrow S$ in \mathcal{F}^* .
- if f then U_1 else U_2 for some function f of type $\text{timestamp}^2 \times \text{type}_{\text{state}} \times \text{type}_e \rightarrow \text{bool}$ in \mathcal{F}^* and finite sets of e -statements U_1, U_2 .

A conditional-tree of statetransform_e is a set of e -statements that satisfies the [unambiguous statevar-update condition](#), which says: Consider the rooted tree formed by the set of e -statements $\text{statetransform}_e^{19}$. 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 a **conditional-tree**, instead of being a set of functions of types $\{\text{timestamp}^2 \times \text{type}_{\text{state}} \times \text{type}_e \rightarrow \text{type}_{\text{state}}(q) \mid q \in \text{statevars}\}$.

Note 4. We will give a specific elaboration of the definition of execution from Section 4.1 soon, but you can probably infer it already. Just note that our use of sets of **statements** instead of lists of **statements** is intentional; their order doesn’t matter.

4.3 Basic Specification: $\text{LSM}(\mathcal{F})$

Since the start of Section 4, 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 $S_1 \times \cdots \times S_k \rightarrow S_0$ 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 below. 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. We also assume $\text{timestamp} \in \mathbb{D}$.

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 , it is “expecting” the **timestamp** of the most recent **event-instance** followed by the **timestamp** of a candidate next **event-instance**. When the type starts with just **timestamp**, the function is expecting just the **timestamp** of the most recent **event-instance**.

¹⁸i.e. $\text{type}_{\text{state}}(x) = S$

¹⁹The tree: A set of e -statements is an internal node whose children are the individual **statements**, with the top-level set 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).

Definition 5 (LSM(\mathcal{F})). Let \mathbb{D} be $\text{sorts}(\mathcal{F})$. An $\text{LSM}(\mathcal{F})$ model is given by the following components.

- `timeunit` in $\{\text{days, hours, minutes, seconds, } \dots\}$.
- Finite sets `actor`, `event`, `situation`, and `statevar`.
- A mapping `typestate` from `statevar` to \mathbb{D} .
- For each $e \in \text{event}$:
 - a subset of `actor`; the `actors` allowed to initiate an `event-instance` of type e .
 - a destination `situation` `deste`.
 - types for its parameters `typee` $\in \mathbb{D}^*$.
 - a conditional-tree, `statetransforme`, which is a set of e -statements that satisfies the `unambiguous statevar-update condition`. The functions used on the right side of assignment statements are from \mathcal{F}^* .
- A finite set `event-rule`, with two partitions:
 - party-rules, oracle-rules, and deadline-rules
 - $\{\text{handler}_s \subseteq \text{event-rule} \mid s \in \text{situation}\}$ where each `handlers` satisfies the `unambiguous deadline condition`.
- An \mathcal{F}^* function `enabled-guardr` : $\text{timestamp} \times \text{type}_{\text{state}} \rightarrow \text{bool}$ for each `event-rule` r .
- For each party or oracle `event-rule` r that governs an `event` e :
 - An \mathcal{F}^* -function `param-constraintr` : $\text{timestamp}^2 \times \text{type}_{\text{state}} \times \text{type}_e \rightarrow \text{bool}$
- For each `deadline event-rule` r that governs an `event` e .
 - An \mathcal{F}^* -function `deadliner` : $\text{timestamp} \times \text{type}_{\text{state}} \rightarrow \text{timestamp}$
 - For each r -parameter x_i of sort $S_i \in \text{type}_e$, an \mathcal{F}^* -function `psetterri` : $\text{timestamp} \times \text{type}_{\text{state}} \rightarrow S_i$

4.4 LSM(\mathcal{F}) Execution

We only need to expand the line

$$\pi' = \text{statetransform}_e(\mathbf{t}, \mathbf{t}', \pi, \sigma)$$

from Section 4.1, but for the sake of a useful reference, we repeat the entire (short) definition here.

Fix a M in $\text{LSM}(\mathcal{F})$. Let $\tau = E_1, E_2, \dots$ be a 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 $G_i = \langle s, \mathbf{t}, \pi \rangle$. Let E_i be $\langle e, a, \mathbf{t}', \sigma \rangle$. We now define the valid values of $G_{i+1} = \langle s', \mathbf{t}', \pi' \rangle$:

- 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 `param-constraintr` is true at $\langle \mathbf{t}, \mathbf{t}', \pi, \sigma \rangle$.

- If E_i is a deadline **event-instance**, then it must be compatible with the unique²⁰ deadline **event-rule** r of s that is enabled in G_i such that $\text{deadline}_r(\mathbf{t}, \pi) = \mathbf{t}'$. Moreover, σ must be $\text{psetter}_r(\mathbf{t}, \pi)$.
- $\pi' = \text{statetransform}_e(\mathbf{t}, \mathbf{t}', \pi, \sigma)$. The evaluation of statetransform_e , and in general of any set of e -statements U , $\text{exec}(U, \mathbf{t}, \mathbf{t}', \pi, \sigma)$, works as follows.
 - If U contains a conditional e of the form **statement if f then U_1 else U_2** , let b be the truth value $f(\mathbf{t}, \mathbf{t}', \pi, \sigma)$. If b is true then

$$\text{exec}(U, \mathbf{t}, \mathbf{t}', \pi, \sigma) = \text{exec}(U - e \cup U_1, \mathbf{t}, \mathbf{t}', \pi, \sigma)$$

else if b is false then

$$\text{exec}(U, \mathbf{t}, \mathbf{t}', \pi, \sigma) = \text{exec}(U - e \cup U_2, \mathbf{t}, \mathbf{t}', \pi, \sigma)$$

- if U contains no conditional **statement**, then let $q_1 \leftarrow f_1, \dots, q_k \leftarrow f_k$ be its contents. So $\{q_1, \dots, q_k\}$ is a subset of **statevar** and $\{f_1, \dots, f_k\}$ is a subset of \mathcal{F}^* . Then π' is identical to π except at q_i , where

$$\pi'(q_i) = f_i(\mathbf{t}, \mathbf{t}', \pi, \sigma)$$

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

5 Automated Formal Verification for LSM

5.1 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](https://rise4fun.com/z3/tutorial)²¹ is a fine place to start.

5.2 Checking the Basic Correctness Conditions

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

²⁰by the unambiguous deadline condition

²¹<https://rise4fun.com/z3/tutorial>

5.3 **LSM(\mathcal{F}) Extension: Adding Pre/Postconditions and Invariants**

5.4 **Symbolic Execution**

In model checking for expressive models (say, with at least nonlinear integer arithmetic available), usually²² 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 5.5 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.

5.4.1 **Initial values, fixed or not?**

First, observe that due to the nondeterminacy in the occurrence of **event-instances** (which recall can have parameters), using fixed **initvals** is not like using fixed inputs in formal verification of software (which would then just be testing, unless the software interacts with an environment, in which case the situation is similar to ours). 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 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 (for a new instance of the contract), we simply rerun symbolic execution.

²²The term “model checking” seems to be used rather inconsistently, at least in the computational law literature.

5.4.2 Timeless Normal Form for $\text{LSM}(\mathcal{F})$

We define a normal form for $\text{LSM}(\mathcal{F})$ 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), we eliminate **enabled-guard**_{*r*} for each party/oracle **event-rule** *r*, as described in Note 3. Next:

- Introduce a new **statevar** \mathbf{t}_{last} of sort **timestamp** with initial value 0.
- Add a parameter \mathbf{t}_{next} of sort **timestamp** to every **event**.
- Add $\mathbf{t}_{\text{last}} \leftarrow \mathbf{t}_{\text{next}}$ to statetransform_e for each *e*.
- Add $>_{ts} : \text{timestamp} \times \text{timestamp} \rightarrow \text{bool}$ to \mathcal{F} .
- For each party/oracle **event-rule** *r*:

$$\text{param-constraint}'_r(\vec{q}, \vec{x}) := \text{param-constraint}_r(\mathbf{t}_{\text{last}}, \mathbf{t}_{\text{next}}, \vec{q}, \vec{x}) \wedge \mathbf{t}_{\text{next}} >_{ts} \mathbf{t}_{\text{last}}$$

- For each **deadline event-rule** *r*:
 - For existing event parameters *x* (i.e. not \mathbf{t}_{next}), psetter_r^x does not change, since its domain type $\text{timestamp} \times \text{type}_{\text{state}}$ is already equivalent to $\text{type}_{\text{state}}'$.
 - $\text{psetter}'_r \mathbf{t}_{\text{next}}(\vec{q}) := \text{deadline}_r(\mathbf{t}_{\text{next}}, \vec{q})$.

Those transformations take us to the following slightly simplified $\text{LSM}(\mathcal{F})$ definition:

Definition 6 (**timeless-LSM**(\mathcal{F})). Let \mathbb{D} be $\text{sorts}(\mathcal{F})$. The relation $>_{ts} : \text{timestamp} \times \text{timestamp} \rightarrow \text{bool}$ must be in \mathcal{F} (and so **timestamp** must be in \mathbb{D}). A **timeless-LSM**(\mathcal{F}) model is given by the following components.

- **timeunit** in {days, hours, minutes, seconds, ...}.
- Finite sets **actor**, **event**, **situation**, and **statevar**.
- A mapping $\text{type}_{\text{state}}$ from **statevar** to \mathbb{D} .
- For each *e* ∈ **event**:
 - a subset of **actor**; the **actors** allowed to initiate an **event-instance** of type *e*.
 - a destination **situation** dest_e .
 - types for its parameters $\text{type}_e \in \mathbb{D}^*$.
 - a **conditional-tree**, statetransform_e , which is a set of *e*-statements that satisfies the **unambiguous statevar-update condition**. The functions used on the right side of assignment statements are from \mathcal{F}^* .
- A finite set **event-rule**, with two partitions:
 - **party-rules**, **oracle-rules**, and **deadline-rules**
 - $\{\text{handler}_s \subseteq \text{event-rule} \mid s \in \text{situation}\}$ where each handler_s satisfies the **unambiguous deadline condition**.
- For each party or oracle **event-rule** *r* that governs an **event** *e*

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

5.4.3 A Minimal **timeless-LSM(\mathcal{F})-Compatible Language**

Here we define a language (syntax) for **timeless-LSM(\mathcal{F})** that is *just enough* to exhibit the symbolic execution algorithm in Section 5.4.4 in a style that is typical of research in programming languages. It is not strictly speaking *necessary* to introduce such a syntax, but it is probably simpler.

Introduce source language symbols, each with a dedicated sort in \mathbb{D} (as per Note 1):

- \underline{q}_i for the i -th **statevar**.
- \underline{x}_i^e for the i -th parameter of **event** e .
- **e-params** for the tuple of e 's parameters.

We previously used the metavariable f for elements of \mathcal{F} or \mathcal{F}^* . We now use it also for elements of a set of function symbols, **\mathcal{F} -fn-symbol**, where there is one such symbol for each element of \mathcal{F} . Each $f \in \mathcal{F}\text{-fn-symbol}$ inherits the function type of the corresponding function in \mathcal{F} , but in a (perhaps misguided) effort to not obscure meaning, we don't introduce notation for that, nor for the sorts of \underline{q}_i and \underline{x}_i^e . This extends Note 1 to function symbols.

The following definitions of **src- \mathcal{F} -term** and **smt- \mathcal{F} -term** are essentially just complete term languages for \mathcal{F}^* , but differing in the symbols they use for variables.

Definition 7 (**src- \mathcal{F} -term** and their types). The set **src- \mathcal{F} -term**, and the type of an **src- \mathcal{F} -term**, are defined by:

- Every **statevar** \underline{q}_i and event parameter \underline{x}_i^e is in **src- \mathcal{F} -term**. The type of each is of course its dedicated sort.
- If $t_1, \dots, t_k \in \text{src-}\mathcal{F}\text{-term}$ and have types $S_1, \dots, S_k \in \mathbb{D}$ and $f \in \mathcal{F}\text{-fn-symbol}$ of type $S_1 \times \dots \times S_k \rightarrow S_0$, then $f(t_1, \dots, t_k) \in \text{src-}\mathcal{F}\text{-term}$ and has type S_0 .

We introduce a set **smt-var**, containing a countable set of variable names for each sort in \mathbb{D} . These are the existential variables that the SMT prover will try to solve for.

Definition 8 (**smt- \mathcal{F} -term** and their types). The set **smt- \mathcal{F} -term**, and the type of an **smt- \mathcal{F} -term**, are defined by:

- Every **smt-var** is in **smt- \mathcal{F} -term**. An **smt-var**'s type is of course its dedicated sort.
- Same as in the **src- \mathcal{F} -term** defn.
If $t_1, \dots, t_k \in \text{smt-}\mathcal{F}\text{-term}$ and have types $S_1, \dots, S_k \in \mathbb{D}$ and $f \in \mathcal{F}\text{-fn-symbol}$ of type $S_1 \times \dots \times S_k \rightarrow S_0$, then $f(t_1, \dots, t_k) \in \text{smt-}\mathcal{F}\text{-term}$ and has type S_0 .

This is not finished! Still need to introduce the syntax for event rules, for example.

5.4.4 Symbolic Execution for timeless-LSM(\mathcal{F})

Metavariables used in the algorithm:

- e, s, r range over event, situation, event-rule, respectively.
- v ranges over **smt-vars**.
- P ranges over **smt- \mathcal{F} -terms** of type **bool**. It is usually called a “path constraint” in symbolic execution algorithms.
- ψ ranges over **src- \mathcal{F} -terms** of type **bool**. t ranges over **src- \mathcal{F} -terms** in general.
- U ranges over sets of **e -statements** (all elements share the same e).
- π ranges over certain **statevar**-substitutions, namely over full mappings from **statevar** to **smt- \mathcal{F} -terms** of the correct type.
- Define **e -subst** to be the set of mappings from **e -params** to **smt- \mathcal{F} -terms** of the correct type. Then:
 σ ranges over $\bigcup_{e \in \text{event}} e\text{-subst}$

We define two “judgements,” both written with \vdash , but differing in the number of arguments they take. Together they define a many-output function, whose return values (**rv** below) are whatever information about the symbolic path taken that you wish to collect. Let \mathbb{I} be the type of that information.

- $\pi; P \vdash \langle \text{thing} \rangle \rightsquigarrow \text{rv}$ is defined for $\langle \text{thing} \rangle$ a **situation**, **situation-handler**, or an **event-rule**.
- $\sigma; \pi; P \vdash \langle \text{thing} \rangle \rightsquigarrow \text{rv}$ is defined for $\langle \text{thing} \rangle$ a set of **statements**. Recall that an **event-handler** is a set of **statements**.

\mathbb{I} should include:

- The set of **situations** visited, the set of **events** and **event-rules** used, and the set of **statements** evaluated, if you want to use symbolic execution to compute code coverage.
- The number of times each **statevar** is “written to” in an assignment **statement**, if you want to check write count bounds given in the source code (not a part of **LSM**, but a feature in **L4**).
- The entire **trace**, if you are using symbolic execution to debug your contract, or if you are using it to speed up the process of writing test cases.
- The **trace** length, if you want to check bounds on that given in the source code.

Let **info**(\cdot, \cdot, \cdot) be your function for extracting the \mathbb{I} from the data you have at the end of a consistent symbolic execution path (currently π, P, s). We will generalize that later, most importantly for extracting information from consistent (reachable) execution paths that contain errors (e.g. failed assertions, which we haven’t added to the model yet).

Definition 9 (\mathcal{F} -consistent, i.e. \mathcal{F} -satisfiable). Let $\mathcal{L}_{\mathcal{F}}$ be the many-sorted first order language with a sort symbol for each set in $\text{sorts}(\mathcal{F})$ and a function symbol for each function in \mathcal{F} . The types of the function symbols are of course the types of the corresponding functions. Let $T_{\mathcal{F}}$ denote the first order theory of $\mathcal{L}_{\mathcal{F}}$ interpreted by \mathcal{F} . We say that an $\mathcal{L}_{\mathcal{F}}$ -formula B is \mathcal{F} -consistent (or \mathcal{F} -satisfiable) if there is a well-typed assignment of objects from $\bigcup \mathbb{D}$ to the free variables of B that makes the formula true according to some model of $T_{\mathcal{F}}$.²³

Now it is time for the proof.sty part. :-)

Roles most ignored atm! Consider eliminating them from timeless-LSM(\mathcal{F}).

Every situation other than those in `terminated` has a non-empty situation-handler:

$$\frac{s \in \text{terminated}}{\pi; P \vdash s \rightsquigarrow \text{info}(\pi, P, s)} \quad \frac{s \notin \text{terminated} \quad \pi; P \vdash \text{handler}_s \rightsquigarrow \text{rv}}{\pi; P \vdash s \rightsquigarrow \text{rv}}$$

From a situation-handler (which recall is just a set of event-rules), each event-rule is tried:

$$\frac{r \in \text{handler}_s \quad \pi; P \vdash r \rightsquigarrow \text{rv}}{\pi; P \vdash \text{handler}_s \rightsquigarrow \text{rv}}$$

In the following rule for party/oracle event-rules, the as-yet not-explicitly-defined *fresh* is an impure function that, given the ordered set of some event e 's parameters (which recall by Note 1 have globally assigned sorts), returns an e -subst mapping them to fresh (not used before) `smt-vars` of the corresponding sorts.

$$\frac{\sigma \leftarrow \text{fresh}(x^e) \quad P' = P \wedge \psi[\sigma, \pi] \quad P' \text{ consistent} \quad \sigma; \pi; P' \vdash \text{handler}_e \rightsquigarrow \text{rv}}{\pi; P \vdash R \text{ may } (e \text{ where } \psi) \rightsquigarrow \text{rv}}$$

$$\frac{P' = P \wedge \psi[\pi] \quad P' \text{ consistent} \quad \sigma = e\text{-params} \mapsto \text{psetter}_e^1(\pi), \dots, \text{psetter}_e^k(\pi) \quad \sigma; \pi; P' \vdash \text{handler}_e \rightsquigarrow \text{rv}}{\pi; P \vdash \text{if } \psi \text{ then fire } e(\text{psetter}_e^1, \dots, \text{psetter}_e^k) \rightsquigarrow \text{rv}}$$

A conditional e -statement results in following one or two paths via the following two rules.

$$\frac{P' = P \wedge \psi[\sigma, \pi] \quad P' \text{ consistent} \quad \sigma; \pi; P' \vdash U \cup X_1 \rightsquigarrow \text{rv}}{\sigma; \pi; P \vdash U \cup \{\text{if } \psi \text{ then } X_1 \text{ else } X_2\} \rightsquigarrow \text{rv}}$$

$$\frac{P' = P \wedge (\neg \psi)[\sigma, \pi] \quad P' \text{ consistent} \quad \sigma; \pi; P' \vdash U \cup X_2 \rightsquigarrow \text{rv}}{\sigma; \pi; P \vdash U \cup \{\text{if } \psi \text{ then } X_1 \text{ else } X_2\} \rightsquigarrow \text{rv}}$$

²³Some SMTLib theories have more than one standard model due to how undefinedness is handled, or due to the allowance of uninterpreted function symbols.

The algorithm arrives at the need to use the following rule after repeated use of the two `if · then · else ·` rules. **Note:** There is currently no rule for single assignment statements $q_i \leftarrow t$ because we have omitted the obvious-but-tedious definition of the “updated by” function used in the following rule.

$$\frac{\pi' = \pi \text{ updated by } \{q_{i_1} \leftarrow t_{i_1}[\sigma, \pi], \dots, q_{i_m} \leftarrow t_{i_m}[\sigma, \pi]\} \quad \pi'; P \vdash \text{dest}_e \rightsquigarrow \text{rv}}{\sigma; \pi; P \vdash \{q_{i_1} \leftarrow t_{i_1}, \dots, q_{i_m} \leftarrow t_{i_m}\} \rightsquigarrow \text{rv}}$$

5.5 Exhaustive Model Checking for Finite and Tamely-Infinite State Spaces

The main topic of this section will be a reduction to a Timed Automata model checking problem for a restricted class of $\text{LSM}(\mathcal{F})$, but we will begin with the even more restricted class whose model checking problem is equivalent to model checking finite automata.

5.5.1 Finite Automata / Buchi Automata

Suppose the only `statevars` in an `LSM` M are typed with small finite sorts such as `bool` or `enums`,²⁴ and suppose that for every `event-rule` r and `event` e none of `enabled-guard` _{r} , `deadline` _{r} , `param-constraint` _{r} , `statetransform` _{e} , or `psetter` _{r} depend on their `timestamp` arguments. Then M is equivalent for formal verification purposes to a kind of compressed²⁵ deterministic finite state machine (FSM), or when infinite `traces` are allowed, to a compressed Buchi Automaton. Let’s consider the general case where infinite `traces` are allowed. Then any properties expressible in the logic [SIS](#)²⁶, which includes safety and liveness properties, are decidable about M .

Since most legal contracts involve time at least a little, and since model checking for the more-general Timed Automata model is also tractable, we will skip giving the reduction for this special case in favour of the one given in the next section.

5.5.2 Timed Automata

We may relax the constraint from the previous section on the functions/predicates that take `timestamp` arguments somewhat, allowing constraints that are boolean combinations of atomic formulas of the form

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

²⁴Technically the following is true for *any* finite sorts, but then the representation as a finite state machine becomes prohibitively expensive.

²⁵via the boolean `statevars`

²⁶https://www.cs.toronto.edu/~wehr/buchis_theorem_relating_finite_automata_on_streams_and_SIS.pdf

where t is the current `timestamp` and t' is the next. `DateTime` and `TimeDelta` literals can be used also, in the higher-level DSL, by compiling them down to expressions about `timestamps`. What we arrive at is a computation model that is close or equivalent to [Timed Automata](https://drum.lib.umd.edu/bitstream/handle/1903/15232/Fontana_umd_0117E_15027.pdf?sequence=1&isAllowed=y)²⁷ with two clocks, one of which is never reset (for the time since the start of the contract) and the other that is reset after each `event-instance`.

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

WARNING: This section is very hastily written.

We use SMT provers (possibly falling back to general first-order theorem provers in cases where the SMT prover gives up) in a second way, distinct from symbolic execution. Consider that almost all of the computation in an LSM happens in the `statetransforms`. We may avoid symbolic execution's limitation to bounded traces by employing preconditions (including the already-available `enabled-guardr`, and `param-constraintr`) and postconditions in various places. The tradeoff is that we lose a certain completeness property of symbolic execution; it never considers behaviours that cannot occur in real executions. In contrast, in the approach of this section, we prove stronger properties about components of the contract (individual `statetransforms`, especially) than are actually needed for correctness. At this point, we're overdue for an example.

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

6 Extensions

6.1 Uninitialized statevars

It is very common in computational contracts, and programming in general, to have state variables that are not meaningful until they are written to the first time, and then are meaningful forever after. L4 introduces a significant extension of the minimal type system that LSMs have (see Section 7.2), which allows one to introduce an `Option` or `Maybe` type operator. However, it is wrong and inconvenient to treat an uninitialized `T` variable as type `Maybe T` if it never becomes `None` again after first getting a value of the form `Some x`.

We therefore propose an extension to $\text{LSM}(\mathcal{F})$ where `initvals` can omit some `statevars`. This requires an additional correctness condition to be checked using the methods of Section 5:

²⁷https://drum.lib.umd.edu/bitstream/handle/1903/15232/Fontana_umd_0117E_15027.pdf?sequence=1&isAllowed=y

[no read-before-write condition](#) - If `statevar q` is omitted from `initvals`, then there is no computation path in which it is read before it is written.

6.2 Unbounded actor Set

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

We introduce to the definition of $\text{LSM}(\mathcal{F})$ 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`.

7 L4: An Experimental but Practical LSM-compatible DSL

7.1 The Largest Set of Examples

We follow, more seriously than any similar project, the tenant that DSL development should be guided by usage.

7.2 Type Checking with Subtyping and Intersection Types

8 Related Work

This is thoroughly covered by [Meng’s book chapter](#)²⁸ and Appendix B of the [CodeX whitepaper](#)²⁹ *Developing a Legal Specification Protocol: Technological Considerations and Requirements*.

9 Dustin’s todo

9.1 ASAP

Before it’s done, absence of these could make some sections of the text unnecessarily hard to read.

²⁸<https://github.com/legalese/complaw-deeptech/blob/master/doc/chapter-201707.org>

²⁹<https://conferences.law.stanford.edu/compkworking201709/wp-content/uploads/sites/40/2017/07/WhitePaperDraftfordistroApril32018.pdf>

- **How to read** - this should be priority 1...
- **A brief defn of substitution for Section 5.4.4.**

9.2 Priority

Before it's done, leaves undocumented some progress we've made.

- Add **assertions, invariants, preconditions, postconditions**. These are super useful when combined with symbolic execution.
- L4 section, obviously.

9.3 Low Priority / Questionable

- **Consider changing** $\text{timestamp}^2 \times \text{type}_{\text{state}} \times \text{type}_e \rightarrow \text{type}_{\text{state}}$ to

$$(\text{timestamp} \times \text{type}_{\text{state}}) \times (\text{timestamp} \times \text{type}_e) \rightarrow \text{type}_{\text{state}}$$

since it better reflects the “Timeless” reduction.

- **Backlinks into “Metavariables used in the algorithm”**
- **Eliminate the word “sort” and replace with “atomic type”?**