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

Report on Computational Law Research by Legalese*

April 16, 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 half and hour with this document.

The primary focus of this report is the definition of the programming language-independent mathematical model for computational legal contracts that we've settled on after a comprehensive review of the literature and many months of research. The model, tentatively called *Legal Abstract State Machines* (LASMs), 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).[3] In programming language theory jargon, LASMs are a denotational semantics.

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

[1]https://legalese.slack.com/messages/C0SB9HZ1S/

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

[3]L4'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 L4 does not have, makes its inclusion unjustified. In fact, it would be hard to add L4's typesystem to the definition of LASMs, as LASMs do not even have a term language!

DRAFT

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 depends strongly on L4 - only on the much simpler mathematical model of Legal Abstract State Machines that one can use L4 (or your own DSL!) to construct.

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

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 Legal Abstract State Machines (LASMs), 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[4] that we *think of* as being in units timeunit, which marks the time since the designated start of the LASM 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, which includes at least Bool, $\mathbb{Z}$, and $\mathbb{R}$. 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.

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

DRAFT

Fix a finite set of symbols event, and for each such e a parameter type assignment $\mathsf{param\text{-}types}_e \in \mathbb{D}^*$. 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 $\sigma \in \mathsf{param\text{-}types}_e$. 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 LASM 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 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]

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

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

[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

3 Legal Abstract State Machines

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

- fulfilled
- $\mathsf{breached}_X$ for each nonempty subset X of $\mathsf{actor} \backslash \{\mathsf{Code}\}$.[8]

An LASM M also has an ordered finite set of symbols statevars, and an assignment statevar-domains of a datatype from $\mathbb{D}$ to each. Since the statevars are ordered, we can take statevar-domains 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

$$\mathsf{situation} \times \mathsf{statevar\text{-}domains} \times \mathsf{timestamp}$$

and a state is an element of the state space.

The remainder and bulk of the definition of an LASM 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 $\mathsf{statetransform}_e$ of type

$$\mathsf{timestamp}^2 \times \mathsf{statevar\text{-}domains} \times \mathsf{param\text{-}types}_e \to \mathsf{statevar\text{-}domains}$$

where the two timestamp arguments provided $\mathsf{statetransform}_e$ will always be the timestamps of the previous and next event-instances.

A situation-handler is a finite set of *event rules*, where 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.[9] Every event-rule r has a relation $\mathsf{enabled\text{-}guard}_r$ on

$$\mathsf{timestamp} \times \mathsf{statevar\text{-}domains}$$

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.

[9]In 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. 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

$$\mathsf{timestamp} \times \mathsf{statevar\text{-}domains} \to \mathsf{timestamp}$$

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

$$\mathsf{timestamp}^2 \times \mathsf{statevar\text{-}domains} \to \mathsf{param\text{-}types}_e$$

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

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

$$\mathsf{timestamp}^2 \times \mathsf{statevar\text{-}domains} \times \mathsf{param\text{-}types}_e$$

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 Abstract State Machine.

We now define the *well-formed event sequences* of an LASM 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 LASM 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.

Execution of LASMs

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 $\langle \mathsf{start\text{-}situation}, 0, \mathsf{initvals} \rangle$. 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 $G_{i+1} = \langle s', t', \pi' \rangle$:

- 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 $\mathsf{param\text{-}constraint}_r$ is true at $\langle t, t', \pi, \sigma \rangle$.
- If E_i is a deadline event, then it must be compatible with the unique deadline event-rule r of s that is enabled in G_i such that $\mathsf{deadline}_r(t, \pi) = t'$.[10]
- $\pi' = \mathsf{statetransform}_e(t, t', \pi, \sigma)$.

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

4 Formal Verification of LASMs

4.1 Satisfiability Modulo Theories (SMT) Technology

4.2 Symbolic Execution

Symbolic execution is model checking for languages with unbounded datatypes.

4.3 Exhaustive Model Checking for bounded or nearly-bounded state spaces

Suppose the only statevars in an LASM M are boolean, and suppose that for every event-rule r none of enabled-guard$r$, $\mathsf{enabled\text{-}guard}_r$, $\mathsf{deadline}_r$, or

[10] Unique by the (todo: udt)

param-constraint$_r$ depend on their timestamp arguments. Then M is equivalent for formal verification purposes to a kind of compressed[11] 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 A Prototype Computational Contracts DSL: L4

5.1 Type Checking with Subtyping and Intersection Types

6 Related Work

[11] via the boolean statevars