DRAFT

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

Report on Computational Law Research

April 16, 2018

Abstract

This report is intended for industry and academics in Computational Law. It should, however, 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 a half hour with this document.

The primary focus of this report is the definition of the programming language-independent mathematical model for 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). In programming language theory jargon, LASMs are a denotational semantics.

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

DRAFT

Contents

1 Introduction

We expect the Computational Law community will develop a number of independent, open source, user-facing DSLs to suit different tastes and focuses, but we hope that the community will be able to share the bulk of our work, particularly in statute and contract libraries, formal verification, and visualization. For this reason, not only is Legalese's DSL L4 completely free and open source, but also none of Legalese's work on formal verification of contracts depends strongly on L4 - only on the much simpler mathematical LASM models 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.

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[3] 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 $\mathbb{R}$. Fix a finite set of labels actor, which includes:

- The parties to the contract.
- Any "oracles" that send information to the contract from the environment.
- The special symbol Code, for events that are initiated by the code of the contract.

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

Fix a finite set of labels 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.
- deadline events, which are transitions mandated by the contract, and

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 always Code.

Event instances are instantaneous,[4] occurring at a particular timestamp; a real world event with duration is modelled by two such instantaneous event-instances, for the start and end of the real world event. That convention is

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

[4]We may relax this before publication.

DRAFT

quite flexible; it easily allows modelling overlapping real-world events, for example. We will see that a sequence of event-instances that constitutes a valid execution of a contract requires strictly increasing time stamps. When we need to model two real-world events as truly-simultaneous, we use one event instance to model their cooccurrence (todo: 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 $\epsilon_1, \epsilon_2 > 0$.

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, and used discrete time, with what we believe was a very satisfying[5] justification. However, the justification requires at least another paragraph, and probably several more to adequately defend it. Meanwhile, it offered no advantages in examples, and had one clear disadvantage for formal verification, where the use of integer variables is costly for SMT solvers.[6]

3 Legal 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 an initial setting initvals of its statevars, and a finite set of labels named situation which must contain the symbols

- fulfilled
- $\mathsf{breached}_X$ for each nonempty subset X of actor.

An LASM M also has an ordered finite set of 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 $\mathbb{D}^*$. The *state space* of M

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

[6] 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.

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

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

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 r has a relation $\mathsf{enabled\text{-}guard}_r$ on $\mathsf{timestamp} \times \mathsf{statevar\text{-}domains}$, where the timestamp argument is the timestamp of the previous event-instance. Frequently $\mathsf{enabled\text{-}guard}_r$ is just the trivial relation true. r is enabled upon entering its parent situation at the timestamp t of the previous event-instance iff $\mathsf{enabled\text{-}guard}_r$ 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* $\mathsf{deadline}_\mathsf{r}$ of type

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

where the timestamp argument is the timestamp of the previous event-instance, and a *parameter setter* $\mathsf{params}_\mathsf{r}$ 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* $\mathsf{param\text{-}constraint}_r$ 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 (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 governing e_{i+1} and a_{i+1} , or E_i is the final element of the sequence and s_i is a *breach* or *fulfilled* situation.

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 trace is valid 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 $\mathsf{deadline}_r(t, \pi) = t'$.[7]
- $\pi' = \mathsf{statetransform}_e(t, t', \pi, \sigma)$.
- If s' is $fulfilled$ or $breached_X$ for some $X \subseteq \mathsf{actor}$, then E_i must be the final event in the trace.

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

[7]Unique by the (todo: udt)

4 Formal Verification of LASMs

4.1 Satisfiability Modulo Theories (SMT) Technology

4.2 Type Checking with Subtyping and Intersection Types

4.3 Symbolic Execution

4.4 Unbounded Formal Verification with Pre/Postconditions and Invariants

4.5 Unexplored: Hard Unbounded Formal Verification with Interactive Theorem Proving

5 A Prototype Computational Contracts DSL: L4