DRAFT

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

Report on Computational Law Research by Legalese*

April 27, 2018

Abstract

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

The primary focus of this report is the definition of the 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 *Contract State Machines* (CSMs), 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, CSMs are a denotational semantics.

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

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

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

[2]L4's typesystem (Section 7.1), though we are quite proud of it, is an example of a feature that does not meet this high standard. It is plausible that the complication it introduces, when in the presence of other optional language features that (the current version of) L4 does not have, makes its inclusion unjustified.

Contents

1 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 CSMs.

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 (CSM), 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 CSM 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 CSM-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 7.1).

[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 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 $\mathbb{D}^* \to \mathbb{D}$ 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.
- 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 $\mathsf{type}_e \in \mathbb{D}^*$. 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 $\sigma \in \mathsf{type}_e$. 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 CSM 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

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

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]

4 Legal State Machines

A Legal State Machine (CSM) first of all fixes the definitions of the terms introduced in Section 3: $\mathbb{D}$, timeunit, actor, and event. It also includes a finite set of symbols situation that must contain at least the symbols:

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

Let terminated be the union of those symbols, i.e. $\{\mathsf{fulfilled}\} \cup \{\mathsf{breached}_A \mid \text{nonempty } A \subseteq \mathsf{actor}\}$.

An CSM M also has an ordered finite set of symbols statevars, and an assignment $\mathsf{type}_{\mathsf{state}}$ of a datatype from $\mathbb{D}$ to each. Since the statevars are ordered, we can take $\mathsf{type}_{\mathsf{state}}$ 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

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

and a state is an element of the state space.

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

[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 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.

DRAFT

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

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

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 by a unique actor.[9] Every event-rule r has a relation enabled-guard$_r$ on

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

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 2 (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$_\mathsf{r}$ of type

$$\mathsf{timestamp} \times \mathsf{type}_{\mathsf{state}} \to \mathsf{timestamp}$$

where the timestamp argument is the timestamp of the previous event-instance. r also has a *parameter setter* psetter$_\mathsf{r}$ of type

$$\mathsf{timestamp} \times \mathsf{type}_{\mathsf{state}} \to \mathsf{type}_e$$

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

DRAFT

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] 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 CSM 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* $\mathsf{param\text{-}constraint}_r$ on

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

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 CSM 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 CSM 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 $\mathsf{breached}_A$ for some nonempty $A \subseteq \mathsf{actor}$.

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

[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.

4.1 CSM Execution

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 $\langle \textsf{start-situation}, 0, \textsf{initvals} \rangle$. 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-instance, then it must be compatible with some party (resp. oracle) event-rule r of s that is enabled in G_i such that $\textsf{param-constraint}_r$ 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 $\textsf{deadline}_r(t, \pi) = t'$. Moreover, σ must be $\textsf{psetter}_r(t, \pi)$.
- $\pi' = \textsf{statetransform}_e(t, t', \pi, \sigma)$.

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

4.2 Reducing the Abstraction By One Level

So far, CSM 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 CSM :

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

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 CSM-compatible DSL.

[12] by the unambiguous deadline condition

In this section, we reduce the *description* of the abstraction level significantly, while maintaining the flexibility of being able to define CSM-compatible languages that range from finite state machines to Turing complete languages.[13] 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 CSM. After this section, we will have a notion of *class of CSM* 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 $S_1 \times \cdots \times S_k \to S_0$ for some $k \geq 0$ and some $S_0, \ldots, S_k \in \mathbb{D}$.

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 $S_1 \times S_2 \to S_1$:

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

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

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 $\mathsf{timestamp} \in \mathbb{D}$.[14] Since $\mathsf{type}_{\mathsf{state}} \in \mathbb{D}^*$ already, the functions $\mathsf{enabled\text{-}guard}_r$, $\mathsf{param\text{-}constraint}_r$, and $\mathsf{deadline}_r$ are already elements of $\mathcal{F}^*$ (which is what we want). The remaining two categories of functions in the bullet-list above, $\mathsf{psetter}_r$ and $\mathsf{statetransform}_e$, simply get replaced by their point-wise components:

- For each deadline event-rule r that governs an event e, and each e-parameter x_i of sort $S_i \in \mathsf{type}_e$, a function $\mathsf{psetter}_r^{x_i} : \mathsf{timestamp} \times \mathsf{type}_{\mathsf{state}} \to S_i$ from $\mathcal{F}^*$.
- For each event e, and each sort $S_i \in \mathsf{type}_{\mathsf{state}}$, a function $\mathsf{statetransform}_e^i : \mathsf{timestamp}^2 \times \mathsf{type}_{\mathsf{state}} \times \mathsf{type}_e \to S_i$ 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:

- $q \leftarrow f$ for some $q \in \mathsf{statevars}$ of sort S[15] and some function f of type $\mathsf{timestamp}^2 \times \mathsf{type}_{\mathsf{state}} \times \mathsf{type}_e \to S$ in $\mathcal{F}^*$.

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

[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. $\mathsf{type}_{\mathsf{state}}(x) = S$

- $\mathsf{if}\, f\, \mathsf{then}\, U_1\, \mathsf{else}\, U_2$ for some function f of type $\mathsf{timestamp}^2 \times \mathsf{type}_{\mathsf{state}} \times \mathsf{type}_e \rightarrow \mathsf{bool}$ in $\mathcal{F}^*$ 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 condition, 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 $\mathsf{if}\, f\, \mathsf{then}\, U_1\, \mathsf{else}\, U_2$. 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 CSM to say that each $\mathsf{statetransform}_e$ is a conditional-tree, instead of being a set of functions of types $\{\mathsf{timestamp}^2 \times \mathsf{type}_{\mathsf{state}} \times \mathsf{type}_e \rightarrow \mathsf{type}_{\mathsf{state}}(q) \mid q \in \mathsf{statevars}\})$.

Note 3. 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: CSM($\mathcal{F}$)

Since the start of Section 4, we've elaborated some details of the initial definition of CSM, 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 $S_0, \ldots, S_k \in \mathbb{D}$. Then $\mathcal{F}$ determines a class of CSMs that we denote $\mathsf{CSM}(\mathcal{F})$, 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 $\mathsf{sorts}(\mathcal{F})$ be that unique determination. We also assume $\mathsf{timestamp} \in \mathbb{D}$.

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 $\mathsf{timestamp}^2$, 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.

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

[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 $\mathsf{if}\, f\, \mathsf{then}\, U_1\, \mathsf{else}\, U_2$ 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).

- $\mathsf{timeunit}$ in {days, hours, minutes, seconds, ...}.
- Finite sets $\mathsf{actor}, \mathsf{event}, \mathsf{situation}$, and $\mathsf{statevar}$.
- A mapping $\mathsf{type}_{\mathsf{state}}$ from $\mathsf{statevar}$ to $\mathbb{D}$.
- For each $e \in \mathsf{event}$, a type for its parameters $\mathsf{type}_e \in \mathbb{D}^*$.
- For each $e \in \mathsf{event}$, a conditional-tree $\mathsf{statetransform}_e$, 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 $\mathcal{F}^*$.
- For each $s \in \mathsf{situation}$, a situation-handler $\mathsf{handler}_s$, 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*
  - $\{\mathsf{handler}_s \mid s \in \mathsf{situation}\}$
- An $\mathcal{F}^*$ function $\mathsf{enabled\text{-}guard}_r : \mathsf{timestamp} \times \mathsf{type}_{\mathsf{state}} \to \mathsf{bool}$ for each event-rule r.
- For each party or oracle event-rule r that governs an event e:
  - An $\mathcal{F}^*$-function $\mathsf{param\text{-}constraint}_r : \mathsf{timestamp}^2 \times \mathsf{type}_{\mathsf{state}} \times \mathsf{type}_e \to \mathsf{bool}$
- For each deadline event-rule r that governs an event e.
  - An $\mathcal{F}^*$-function $\mathsf{deadline}_r : \mathsf{timestamp} \times \mathsf{type}_{\mathsf{state}} \to \mathsf{timestamp}$
  - For each r-parameter x_i of sort $S_i \in \mathsf{type}_e$, an $\mathcal{F}^*$-function $\mathsf{psetter}^i_r : \mathsf{timestamp} \times \mathsf{type}_{\mathsf{state}} \to S_i$

4.4 CSM($\mathcal{F}$) Execution

We only need to expand the line

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

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

Fix a M in CSM($\mathcal{F}$). Let $\tau = E_1, E_2, \ldots$ be a 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-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[17] deadline event-rule r of s that is enabled in G_i such that $\mathsf{deadline}_r(t, \pi) = t'$. Moreover, σ must be $\mathsf{psetter}_r(t, \pi)$.

[17] by the unambiguous deadline condition

- $\pi' =$ statetransform$_e(t, t', \pi, \sigma)$. The evaluation of statetransform$_e$, and in general of any set of e-statements U, $\mathsf{exec(U, t, t', \pi, \sigma)}$, 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 $f(t, t', \pi, \sigma)$. If b is true then

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

    else if b is false then

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

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

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

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

5 Automated Formal Verification for CSM

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 is a fine place to start.

5.2 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.

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

5.4 Symbolic Execution

In model checking for expressive models (say, with at least nonlinear integer arithmetic available), usually[18] an infinite-state model is approximated by a finite or tamely-infinite state model, and correctness properties of the approximation model

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

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 abstract?

When we symbolically execute an CSM, we have a choice about whether to use some, all, or none of the model's $\mathsf{initvals}$ of its $\mathsf{statevars}$. Generally, the more of them used, the faster $\mathsf{state\ space}$ 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, we simply rerun symbolic execution.

5.4.2 Timeless Normal Form for $\mathsf{CSM}(\mathcal{F})$

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

First, and unrelated to the following $\mathsf{timestamp}$-related transformations (but with the same goal of simplifying symbolic execution), eliminate $\mathsf{enabled\text{-}guard}_r$ for each party/oracle $\mathsf{event\text{-}rule}$ r, as described in Note 2.

- Introduce a new $\mathsf{statevar}$ t_{last} of sort $\mathsf{timestamp}$ with initial value 0.[19]
- Add a parameter t_{next} of sort $\mathsf{timestamp}$ to every event[20]
- Add $t_{\mathsf{last}} \leftarrow t_{\mathsf{next}}$ to $\mathsf{statetransform}_e$ for each e.
- Replace $\mathsf{psetter}^i_r$ with $\mathsf{psetter}^i_r(t_{\mathsf{last}}, \cdot)$. Move $\mathsf{deadline}_r(t_{\mathsf{next}}, \cdot)$ into $\mathsf{psetter}^{t_{\mathsf{next}}}_r$ for each deadline $\mathsf{event\text{-}rule}$ r.

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

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

- Add $>_{ts}$:$\mathsf{timestamp} \times \mathsf{timestamp} \to \mathsf{bool}$ to $\mathcal{F}$.
- Conjoin ($\wedge$) $t_{\mathsf{next}} >_{ts} t_{\mathsf{last}}$ to $\mathsf{param\text{-}constraint}_r$ for each party/oracle event-rule r.

We obtain the following slightly simplified CSM definition:

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

- timeunit in {days, hours, minutes, seconds, ...}.
- Finite sets actor, event, situation, and statevar.
- A mapping $\mathsf{type}_{\mathsf{state}}$ from statevar to $\mathbb{D}$.
- For each $e \in$ event, types for its parameters $\mathsf{type}_e \in \mathbb{D}^*$.
- For each $e \in$ event, a conditional-tree $\mathsf{statetransform}_e$, 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 $\mathcal{F}^*$.
- For each $s \in$ situation, a situation-handler $\mathsf{handler}_s$, 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
  - $\{\mathsf{handler}_s \mid s \in \mathsf{situation}\}$
- For each party or oracle event-rule r that governs an event e
  - An $\mathcal{F}^*$-function $\mathsf{param\text{-}constraint}_r : \mathsf{type}_{\mathsf{state}} \times \mathsf{type}_e \to \mathsf{bool}$
- For each deadline event-rule r that governs an event e.
  - An $\mathcal{F}^*$ function $\mathsf{enabled\text{-}guard}_r : \mathsf{type}_{\mathsf{state}} \to \mathsf{bool}$ for each event-rule r.
  - For each sort $S_i \in \mathsf{type}_e$, an $\mathcal{F}^*$-function $\mathsf{psetter}^i_e : \mathsf{type}_{\mathsf{state}} \to S_i$

5.4.3 A Minimal timeless-CSM($\mathcal{F}$)-Compatible Language

Here we define a language (syntax) for timeless-CSM($\mathcal{F}$) 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):

- q_i for the i-th statevar.
- x^e_i for the i-th parameter of event e.

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 $f \in \mathcal{F}$-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 q_i and 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 q_i and event parameter x_i^e is in src-$\mathcal{F}$-term. The type of each is of course its dedicated sort.
- If $t_1, \ldots, t_k \in$ src-$\mathcal{F}$-term and have types $S_1, \ldots, S_k \in \mathbb{D}$ and $f \in \mathcal{F}$-fn-symbol of type $S_1 \times \cdots \times S_k \to S_0$, then $f(t_1, \ldots, t_k) \in$ src-$\mathcal{F}$-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 solver 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 $t_1, \ldots, t_k \in$ smt-$\mathcal{F}$-term and have types $S_1, \ldots, S_k \in \mathbb{D}$ and $f \in \mathcal{F}$-fn-symbol of type $S_1 \times \cdots \times S_k \to S_0$, then $f(t_1, \ldots, t_k) \in$ smt-$\mathcal{F}$-term and has type S_0.

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

WARNING: This section is not ready for reading yet

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.
- X 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 x_e to smt-$\mathcal{F}$-terms of the correct type. Then
  σ ranges over $\bigcup_{e \in \mathsf{event}} e$-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.

- $\pi; P \vdash \langle\text{thing}\rangle \rightsquigarrow \mathsf{rv}$ is defined for $\langle\text{thing}\rangle$ a situation, situation-handler, or an event-rule.
- $\sigma; \pi; P \vdash \langle\text{thing}\rangle \rightsquigarrow \mathsf{rv}$ 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 CSM, 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 $\mathsf{info}(\cdot,\cdot,\cdot)$ 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 $\mathsf{sorts}(\mathcal{F})$ 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 $T_\mathcal{F}$.[21]

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

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

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

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

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

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

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.

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

$$\frac{\begin{array}{cc}P' = P \wedge \psi[\pi] & \sigma = \mathsf{x}_e \mapsto \mathsf{psetter}_e^1, \ldots, \mathsf{psetter}_e^k \\ P' \text{ consistent} & \sigma;\pi;P' \vdash \mathsf{handler}_e \rightsquigarrow \mathsf{rv}\end{array}}{\pi; P \vdash if\ \psi\ then\ fire\ e(\mathsf{psetter}_e^1, \ldots, \mathsf{psetter}_e^k) \rightsquigarrow \mathsf{rv}}$$

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

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

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

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.

$$\frac{\begin{array}{c}\pi' = \pi \text{ updated by } \{q_{i_1} \leftarrow t_{i_1}[\sigma,\pi], \ldots, q_{i_m} \leftarrow t_{i_m}[\sigma,\pi]\} \\ \sigma;\pi';P \vdash \mathsf{dest}_e \rightsquigarrow \mathsf{rv}\end{array}}{\sigma;\pi;P \vdash \{q_{i_1} \leftarrow t_{i_1}, \ldots, q_{i_m} \leftarrow t_{i_m}\} \rightsquigarrow \mathsf{rv}}$$

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

Suppose the only statevars in an CSM M are bool or enums, 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 compressed[22] deterministic finite

[22]via the boolean statevars

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.

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

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

6 Extensions

6.1 Unbounded actor Set

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

We introduce to the definition of CSM($\mathcal{F}$) 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 CSM-compatible DSL

7.1 Type Checking with Subtyping and Intersection Types

8 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.*

9 Dustin's todo

9.1 ASAP

Before it's done, could make some section of the text unnecessarily hard to read.

- How to read - this should be priority 1...
- A brief defn of substitution for Section 5.4.4.
- A note in Section 5.4.4 that P consistent is the result of an SMT query.
- Different metavariable for $\mathsf{timeunit}$, since use t for $\mathsf{src}\text{-}\mathcal{F}\text{-}\mathsf{term}$ later.
- Unify metavariables for sets of statements. Use U not X.

9.2 Priority

Before it's done, leaves undocumented some significant 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 $\mathsf{timestamp}^2 \times \mathsf{type}_{\mathsf{state}} \times \mathsf{type}_e \to \mathsf{type}_{\mathsf{state}}$ to

$$(\mathsf{timestamp} \times \mathsf{type}_{\mathsf{state}}) \times (\mathsf{timestamp} \times \mathsf{type}_e) \to \mathsf{type}_{\mathsf{state}}$$

  since it better reflect the "Timeless" reduction.
- Backlinks into "Metavariables used in the algorithm"
- Eliminate the word "sort" and replace with "atomic type"?