Formalizing a simple loan agreement in L4 and Maude

Joe Watt

March 31, 2023

This is a work in progress.

Contents

1 Introduction

In [2], Flood & Goodenough propose that many financial contracts are inherently computational in nature. They argue that the computational structure of many such contracts can be formalized via deterministic finite automata (DFAs), with states representing various situations. Transitions between these states then correspond to events triggering a change in these situations. This is demonstrated using a simple loan agreement [2], which they formalize directly as a DFA.

While the DFA approach is a useful and important proof of concept, as the authors themselves recognize, there are some limitations with such a formalism, perhaps the most apparent being that the manual encoding of a contract as a DFA is a laborious process, and one that in itself does not produce a machine executable result.

Our first demonstration in this paper is using the L4 language being developed at the Singapore Management University (SMU) to represent the bargain described by Flood & Goodenough [2]. Through this, we show how L4 addresses some shortcomings of the DFA formalization.

Next, we discuss how we implemented an execution engine that allows us to execute contracts written in L4. For this, we designed a Structural Operational Semantics (SOS) for L4, giving us a mathematical description of the execution behavior of contracts. To make this executable, we implemented this mathematical description in the Maude language, which in turn enables us to generate interactive visualizations of the resulting transition system.

Cite Plotkin's seminal paper

2 Shortcomings of the DFA formalization

As aforementioned, DFAs quickly become cumbersome to work with directly when the contracts we are trying to encode grow larger. One source of complexity arises from the fact that the DFA is a low level computational formalism that is arguably far removed from our intuitive understanding of a contract. While we can easily encode real-world events as transitions in a DFA, as done in [2], some legal traditions reason about and specify contracts in terms of normative requirements that need to be fulfilled by various actors [3]. The authors of this paper, representing different traditions, do not entirely agree on this point, but it remains true that a DFA is a relatively mechanistic chain of event and consequence, and is not adept at normative representation.

Contracts also frequently embody some notion of time and deadline. These are examples of domain-specific concepts which are an integral part of contract drafting. Unfortunately, DFAs have no primitive notion of these, so that these must all be encoded manually, with the passage of time reflected as an achieved event, in a manner that is perhaps unnatural to many.

Another source of complexity is the state explosion problem. Complex contracts have large DFA representations and are thus impractical for humans to specify manually at scale. Observe that a simplified visual representation of the

automaton [2, Fig. 1] corresponding to this simple contract already contains more than 20 states and 40 transitions.

One of the main causes of this is the concurrent interleaving of real world events. By this, we mean that many real world events can occur in any order (including at the same time) and perhaps for the sake of simplicity, this has not been accounted for in the DFA formalization. As an example of such a scenario, consider the following. On May 31, 2015, the day before payment 1 is due, the borrower defaults on his representations and warranties. On the next day, when payment 1 becomes due, the borrower diligently repays that payment. One day later, on June 2, 2015, the lender notifies the borrower of his earlier default, which does not get cured after another 2 days. Now all outstanding payments become accelerated, and the borrower pays off the remaining amount of $525 in time, causing the contract to terminate. This sequence of events, when viewed as a word over the event alphabet, is unfortunately not accepted by the automaton.

Closer inspection of the DFA suggests that there is an implicit assumption being made that once an event of default occurs, the borrower will be notified by the lender soon after, with no other events like payments occurring in between.

More formally, it is assumed that the default and notification events occur within the same atomic step. Splitting this up into separate events would increase the number of states in the DFA, especially if we account for concurrency. In fairness, Flood and Goodenough understood some of these limits, and specified in [2][Section 6] of their model agreement that: "In the event of multiple events of default, the first to occur shall take precedence for the purposes of specifying outcomes under this agreement". This provision goes some way toward resolving competing defaults by the expedient of mandating that later events will simply be disregarded.

3 The L4 approach

The L4 language for encoding legal texts is being developed at SMU which addresses some of these shortcomings. As described by its developers, "L4 is a domain-specific specification language that facilitates semantically rigorous formalization of legal expressions found in legislation and contracts." (Mahajan, Strecker & Wong. 2022)

cite properly

Its core approach utilizes a natural language-based structure and syntax which allows the detailed, if artificially structured, expression of the obligations and consequences that need to be set out in a legal specification. The programming language recognizes the logic embedded in these statements and then allows that logic to be represented in a number of different execution software approaches. (Id) It is capable of representing both the private bargains of contract and the public prohibitions and requirements of statutes and regulations. Additional background on L4 and its applications is available at _ .

cite properly

Cite

As described in , L4, as with LegalRuleML , distinguishes between two kinds of rules, namely constitutive and regulative (or prescriptive) rules. The former

Cite WAICOM 2022 and PROLALA 2022 papers

Cite Legal-RuleML

encodes static decision logic, similar to Prolog-style *if-then* rules.

Our focus in this work is on the latter, namely regulative rules, which concern the norms in a legal text. These are a central component in the computational structure of contracts like the loan agreement. They govern under which conditions an actor is allowed, permitted or prohibited from performing an action, as well as the consequences of fulfilling and violating them.

While deontic logic approaches have been popular, our approach to regulative rules in L4 is closer in spirit to [1, 4], in that our treatment of the rules is operational in nature, grounded in the mathematical framework of SOS. This yields a state transition system, similar in spirit to the DFA found in the work of Flood & Goodenough [2].

Insert example, like the one in PROLALA paper

Cite appropriately

Cite the Maude paper as well, and any others that are relevant

3.1 Syntax of regulative rules

Here we describe the natural-language like syntax for specifying regulative rules in L4, as well as a brief discussion of the relevant domain specific concepts. A more detailed discussion with some examples will be provided in the following section, in the context of our encoding of the loan agreement.

Regulative rules in L4 have the following form, with special keywords given in all-caps:

```
RULE rule name
PARTY actor
deontic
deadline
[ DO ] action
[ HENCE hence_0 AND ... AND hence_n ]
[ LEST lest_0 AND ... AND lest_n ]
```

Note that the `deadline`, `DO` keyword, as well as the `HENCE` and `CLAUSES` are optional. We also allow for some permutations of the `deontic`, `deadline` and `action` clauses.

Each clause in the above rule corresponds to the following domain concepts that we have identified:

- `actor`
- `action`

  Actors and actions are currently plain strings with no special meaning attached to them.

- `deontic`

  There are 3 kinds of deontics:

  - `MUST`: Achievement obligation
  - `MAY`: Permission
  - `SHANT`: Prohibition

- `deadline`

  This can take the form:

  - `WITHIN` n `DAY`
  - `ON` n `DAY`

  These are relative to the time when the rule comes into effect and `WITHIN` n `DAY` indicates that the action may occur at any time within the next n days, while `ON` n `DAY` means that it may only occur on the n-th day itself.

  Note that we currently do not support weeks, months, years as well as fixed dates, though we intend to add support for these in the future. Also, all rule are currently required to include a deadline. We intend to relax this in the future.

- `hence_i` (resp `lest_i`)

  `hence_i` (resp `lest_i`) are rule names denoting rules that are triggered when the obligation or prohibition is fulfilled (resp violated). We identify the fulfillment of an obligation with whether the actor performed the prescribed action within the deadline, and violation is identified with the deadline passing before the actor performs the action.

  Fulfillment and violation for prohibitions are defined symmetrically.

  In the case of permissions, the `hence_i` are triggered when the permission is exercised, and the `lest_i` are triggered when it is not exercised by the deadline.

For instance, consider an obligation of the form

```
RULE rule
PARTY actor
MUST DO action
WITHIN 3 DAY
HENCE hence
LEST lest0 AND lest1
```

If `actor` performs `action` on days 0, 1 or 2, we deem the obligation to have been fulfilled, and the timer is stopped. In this case, the rule named `hence` now takes effect and a new timer begins. If instead, 3 days pass without the actor performing the action, we deem the rule to have been violated. Now, the two rules, `lest0` and `lest1`, in the `LEST` clause are triggered.

3.2 The loan agreement in L4

(TODO: Add link)

It should be mentioned that our encoding of the loan agreement is an approximation of the one in the original paper [2] as we currently lack certain features like fixed dates, which are used as deadlines in the original contract. To cope with this, we specify deadlines in terms of days relative to the time

the rule triggers, in place of dates. Other limitations and inadequacies will be pointed out along the way.

The following rule is found at the beginning of our encoding of the loan agreement. We assume that the first rule in an L4 encoding is the starting rule which is triggered at the beginning of the contract.

```
RULE Contract Commencement
PARTY Borrower
MAY WITHIN 1 DAY
request funds
HENCE Remit principal
```

This starting rule says that the actor, `Borrower`, may perform the `request principal` action any time from the time the rule comes into effect up til 1 day later. Note that for simplicity, we assume that permissions are one-off and are used up once the action occurs, so that no repeated actions are allowed. This assumption mirrors that of [1], in which the author notes that one can also allow for permissions that may be exercised more than once. In the future, we hope to lift this restriction.

The `HENCE` clause is then used to indicate that if the permission is exercised, then another rule called `Remit Principal` comes into effect. The lack of a `LEST` clause indicates that no new rule is triggered when the deadline passes without the permission being exercised.

The `Remit principal` rule is then expressed as follows:

```
RULE Remit principal
PARTY Lender
MUST remit principal
     in the amount of 1000
     to Borrower
WITHIN 1 DAY
HENCE Repayment
```

This obliges the lender to send the principal amount to the borrower once the borrower has requested for it. The `Repayment` rule defines the main body of the contract and comes into effect when the `Remit principal` rule is fulfilled, ie when the lender sends the principal to the borrower. Note that L4 allows us to define a rule to be a conjunction of multiple rules so that `Repayment` is defined as:

```
       Repayment
MEANS  Repay in two halves
AND    Avoid default
```

Triggering the composite `Repayment` rule then has the effect of triggering the `Repay in two halves` and `Avoid default` rules in parallel. The former obliges the borrower to make the first repayment, followed by the second, and the latter prohibits him from defaulting on representations and warranties and so on.

It should be noted here that as in _ , we distinguish between the violation of

cite one of Guido's papers that talks about contrary to duty norms and compensability

an obligation/prohibition, and the violation of the contract as a whole. Obligations and prohibitions like `Remit principal`, which do not have a `LEST` clauses are assumed to be non-reparable. This means that when this obligation is violated, we deem the contract to have been breached by the lender. Execution of the contract proceeds no further and to draw a parallel with computer programs, we can view this as the contract (seen as a program) throwing an unrecoverable exception during execution, causing it to terminate in a breach state (ie crash).

In contrast, some rules, like the `Repay in two halves` obligation below, have a nonempty `LEST` clause and so is deemed to be reparable, with reparation mechanisms, or *contrary-to-duty* rules , being encoded in the `LEST` clause. In this case, the violation of the rule results in the rules in the `LEST` clause coming into effect, and the execution of the contract continues, as opposed to terminating in a breach state. Going back to the analogy with computer programs, one can view these contrary-to-duty norms as exception handling mechanisms that activate in an attempt to steer the execution of the contract back along the "happy path".

Cite a Guido paper on CTD norms and compensability

```
RULE Repay in two halves
PARTY Borrower
MUST ON 30 DAY
DO repay first half
   in the amount of 550
   to Lender
HENCE Repay second half
LEST Notify borrower of default
```

In this rule, the `ON 30 DAY` deadline is used to mean that early payments are not allowed, as in the original agreement. While we could have chosen to encode 1 year as 365 days, due to the way in which we handle time and deadlines under the hood, 365 days results in too large of a transition system for our system to handle. Thus, we have instead chosen a more modest deadline of 30 days.

The `Notify borrower of default` grants the lender permission to notify the borrower and encodes the "notice and cure" reparation mechanism. As noted in [2], this indicates the start of the reparation pathway, which gives the borrower a second chance, by obliging him to cure his default. Should he still not make amends, the execution of the contract has reached the point of no return and no further second chances are given. This is embodied by the activation of the `Accelerate outstanding payments` obligation.

```
RULE Accelerate outstanding payments
PARTY Borrower
MUST WITHIN 1 DAY
DO repay outstanding amount
   to Lender
```

Unfortunately, while encoding this rule in L4, we realized that our formalization does not faithfully implement the intuitive reading of its corresponding natural language counterpart. Intuitively, when the execution of the contract reaches this stage, no other obligations are relevant anymore, like the `Avoid`

`default` prohibition. Consequently, one would like to be able to terminate all existing obligations when this comes into effect. While we would have liked to add support for this, we encountered various complications and hence decided to leave this for future work.

4 Operational semantics of Regulative Rules in L4

In order to operationalize contracts for execution, we begin by formalizing a semantics for regulative rules in L4 using the framework of Structural Operational Semantics (SOS) originating from the field of programming language theory. Cite Plotkin

SOS is a mathematical framework which allows one to describe transition systems in terms of states, called configurations, and transition rules that describe how one state can transition to another. These rules are expressed as logical inference rules of the form:

$$\frac{H_1 \quad \dots \quad H_n}{S \xrightarrow{E} S'}$$

Here $S \xrightarrow{E} S'$ is known as a judgement form which expresses that the configuration S evaluates to S' when an event E occurs. With this, the above transition rule is then read as saying that given a configuration S, if all the hypotheses H_i hold, then the event E is allowed to occur, and S is allowed to transition to S' when it does.

Our configurations and transitions are parameterized over a set of rules, called `Rule`. Our semantics bears resemblance to that of timed arc petri nets (TAPNs), which are based on timed transition systems . As with markings in TAPNs, configurations in our semantics help us keep track of active rules and their respective deadlines via timers that count down when a rule is triggered and becomes active. Cite

The key difference is that we also allow for configurations to also be failure states, which we call *breach states*. These breach states indicate the actors who breached the contract and are final in that they have no outgoing transitions.

To formalize this, we first define an *active rule instance* to be a pair of the form (r, t) where $r \in$ `RuleName` is a rule name and $t \in \mathbb{N}$ is the corresponding timer value of the rule. With this, a configuration C is a pair that is either:

1. (`Active`, R)

   where `Active` is used to indicate that the contract is still active (ie not breached) and $R \subseteq$ `RuleName` $\times \mathbb{N}$ is a set of active rule instances.

   We view a contract to be fulfilled if there are no active rule instances remaining, and so we define `Fulfilled` $=$ (`Active`, $\emptyset$).

Note that rules can be viewed as places in a TAPN, so that such a set of active rule instances can then be seen as a marking, with each one being a timed token with value t on the place r.

2. $(\texttt{Breached}, A)$

   is a breach state where `Breached` indicates that the contract has been breached, and $A \subseteq \texttt{Actor}$ identifies the actors who breached it.

Next, we equip configurations with the lattice obtained from the partial order given by:

$$\begin{aligned}
(\texttt{Breached}, A) &\leq (\texttt{Breached}, A') && \text{if} \quad A' \subseteq A \\
(\texttt{Active}, R) &\leq (\texttt{Active}, R') && \text{if} \quad R' \subseteq R \\
(\texttt{Breached}, A) &\leq (\texttt{Active}, R)
\end{aligned}$$

so that we have:

$$\begin{aligned}
(\texttt{Breached}, A) \wedge (\texttt{Breached}, A') &= (\texttt{Breached}, A \cup A') \\
(\texttt{Breached}, A) \wedge (\texttt{Active}, R) &= (\texttt{Breached}, A) \\
(\texttt{Active}, R) \wedge (\texttt{Active}, R') &= (\texttt{Active}, R \cup R')
\end{aligned}$$

Note that the lattice meet $\wedge$ is crucial for our semantics as we will explain later. This operator allows us to combine configurations by taking the union of the corresponding sets of actors (resp active rule instances) if both configurations are breached (resp active). Note that breach states absorb active states.

TODO
Probably best to shift this part on the transition rules up to the top and then define δ_{action} and δ_{tick} properly here.

Following timed transition systems, we define 2 kinds of events, namely action events of the form `actor does action` and discrete tick events indicating the passage of a day. We then have the following transitions:

$$\frac{\text{if the } \texttt{actor does action} \text{ event is allowed to occur in the configuration } C}{C \xrightarrow{\texttt{actor does action}} \delta_{\text{action}}(\texttt{actor does action}, C)} \text{ action}$$

$$\frac{}{C \xrightarrow{\text{tick}} \delta_{\text{tick}}(C)} \text{ tick}$$

where δ_{action} and δ_{tick} are transition functions that compute the effect of action events and tick events on the current state of the contract. The use of the precondition for action transitions is to restrict actions so that we can only take that transition when there is an active rule mentioning it.

More precisely, we say that an action event `actor does action` is allowed to occur in a configuration C if either:

- There is a rule named r with a deadline of the form `WITHIN n DAY` and C contains an active rule instance of the form (r, m) where $m \leq n$.
- There is a rule named r with a deadline of the form `ON n DAY` and C contains an active rule instance of the form $(r, 0)$

The transition functions are defined via primitive recursion using the lattice meet $\wedge$, or in functional programming terms, a `foldMap`, viewing the meet operator as the binary operator of an (idempotent and commutative) monoid.

$$
\begin{aligned}
&\delta_{\text{action}}(\texttt{actor does action}, \texttt{Fulfilled}) \\
=&\delta_{\text{action}}(\texttt{actor does action}, (\texttt{Active}, \emptyset)) \\
=&\texttt{Fulfilled} \\
&\delta_{\text{action}}(\texttt{actor does action}, (\texttt{Breached}, A)) \\
=&(\texttt{Breached}, A) \\
&\delta_{\text{action}}(\texttt{actor does action}, (\texttt{Active}, \{(r, t)\} \cup R)) \\
=&(\texttt{Active}, \text{hences}) \wedge \delta_{\text{action}}(\texttt{actor does action}, (\texttt{Active}, R)) \\
&\text{if deontic}(r) \in \{\texttt{MUST}, \texttt{MAY}\} \text{ and actor}(r) = \texttt{actor} \text{ and action}(r) = \texttt{action}
\end{aligned}
$$

Here hences refers to all the `hence_i` in the (`HENCE` hence_1 `AND ... AND` hence_n) clause of the rule, along with their respective timers. Similarly, lests refers to those in the lest clause.

$$
\begin{aligned}
&\delta_{\text{action}}(\texttt{actor does action}, (\texttt{Active}, \{(r, t)\} \cup R)) \\
=&(\texttt{Breached}, \{\texttt{actor}\}) \wedge \delta_{\text{action}}(\texttt{actor does action}, (\texttt{Active}, R)) \\
&\text{if deontic}(r) = \texttt{SHANT} \text{ and lests} = \emptyset \text{ and actor}(r) = \texttt{actor} \text{ and action}(r) = \texttt{action} \\
&\delta_{\text{action}}(\texttt{actor does action}, (\texttt{Active}, \{(r, t)\} \cup R)) \\
=&(\texttt{Active}, \text{lests}) \wedge \delta_{\text{action}}(\texttt{actor does action}, (\texttt{Active}, R)) \\
&\text{if deontic}(r) = \texttt{SHANT} \text{ and lests} \neq \emptyset \text{ and actor}(r) = \texttt{actor} \text{ and action}(r) = \texttt{action}
\end{aligned}
$$

Fun fact: This `foldMap` looking definition can be foramlized as a `traverse` over an Either, with `Active` $\cong$ `Right` and `Breached` $\cong$ `Left`. With this analogy, the absorption property of the lattice meet ensures that lefts absorb rights, which in turn has the effect of short-circuiting the `traverse` once a non-compensable rule is breached.

δ_{tick} is defined similarly.

5 L4 through Maude

In search for such a suitable formalism for encoding contracts, we have surveyed various approaches, using the simplified contract in [2, Fig 1.] as an example. One of these which we explored is the Maude language and its associated tools.

In this section, we discuss how we implement the above semantics in Maude in order to turn it into executable code. Thereafter, we demonstrate how we can use Maude to help generate a DFA representation of the state space of the contract and visualize it.

TODO
Maybe elaborate more on Maude and talk about how Maude makes for a good *meta* language for designing and modelling DSLs
Discuss the relationship between Maude and the mathematical notation used in defining operational semantics for programming languages.

Maude is a language grounded in rewriting logic, and in some ways resembles functional programming languages like Haskell. Programs in Maude are organized into modules, and we can define algebraic data types and equations that instruct the system on how to evaluate terms of these datatypes.

For instance, we defined a sort for contract states, along with constructors for the various states:

```
sort ContractState .
op Active : Set{ActiveRule} -> ContractState .
op Breached : Set{ActorName} -> ContractState .
op Fulfilled : ContractState .
```

We can then define the signature for the lattice meet operator via:

```
op _^_ : ContractState ContractState -> ContractState
  [assoc comm id: Fulfilled] .
```

using `assoc` and `comm` to indicate that this operator is associative and commutative. `id: Fulfilled` indicates that `Fulfilled` is to be treated as the identity element.

The rest of the semantics of this operator is then specified using equations of the following form, with the comma `,` denoting set union.

```
eq Active empty = Fulfilled .

eq Active activeRules ^ Active activeRules'
= Active (activeRules, activeRules') .

--- Absorption properties.
eq Breached actorNames ^ Active activeRules]
= Breached actorNames .

eq Breached actorNames ^ Breached actorNames'
```

```
= Breached (actorNames, actorNames') .
```

With this, the transition functions δ_{action} and δ_{tick} are formalized with signature

```
op deltaAction : Set{Rule} ActionEvent ContractState -> ContractState .
op deltaTick : Set{Rule} ContractState -> ContractState .
```

and equations resembling those presented in the previous section. Note that these take an extra `Set{Rule}` parameter in our implementation, representing the set of all rules in the contract.

Maude also allows us to define rewriting rules, which is what we used to encode the transition system semantics. In order to encode our transition rules of the form $\frac{H_1 \dots H_n}{C \xrightarrow{E} C'}$ in Maude, we first introduced a new datatype `Configuration` with constructor

```
op Config : ContractState Event -> Configuration .
```

and then encode the rules in the following form:

```
crl [action] :
  Config contractState event
=>
  Config
    (deltaAction rules contractState (actor does action))
    (actor does action)
if condition .

rl [tick] :
  Config contractState event
=>
  Config (deltaTick rules contractState) tick .
```

5.1 Executing and visualizing the loan agreement

Insert DFA of state space here.

6 Limitations and future work

TODO
Shift this discussion of time the previous section on visualizing the DFA representation when that's ready.

The current Maude models are not very performant and have huge state spaces. One of the main reasons is that for simplicity, we chose a naive approach in which configurations in our state space keep track of timers that count down

one day at a time as `tick` transitions occur. This means that an active rule instance arising from a rule with a large deadline like 365 days, can attain any value from 0 to 365 during the execution of the contract. Thus, the introduction of such a rule increases the number of possible states by at least 366, growing exponentially with the number of rules and possible timer values in the contract.

TODO
Mention that various shortcomings have already been mentioned along the way in the previous sections and summarize them quickly, before listing a few more general ones here.

Unfortunately, this considerably bloats up the state space. Again it must be emphasized that this is a proof of concept and these were designed to be as high-level and close to the syntax of L4 as possible. In the future, we intend to optimize this, among other things.

Currently, we don't yet support global variables and more sophisticated control structures like loops. Global variables would be useful to capture the notion of the outstanding amount in the loan agreement contract, which would then vary with the payments made by the borrower. Maude as a formalism supports these, but we currently do not support them in L4. For future work, we would like to provide syntax for these in L4 and translate them to Maude.

Perhaps the biggest downside is that while we have primarily focused our efforts on the moving parts of a contract, ie regulative rules, we have yet to consider static decision logic, ie constitutive rules. What we would like to do is to allow users to define constitutive rules and use them as "if conditions" in regulative rules, so that they can write rules of the form:

```
RULE ruleName
PARTY actor
IF condition
MUST DO action
WITHIN n DAY
```

where the rule is only triggered if `condition` holds.

7 Conclusions

References

[1] Chircop, S., Pace, G., and Schneider, G. *An Automata-Based Formalism for Normative Documents with Real-Time.* 12 2022.

[2] Flood, M. D., and Goodenough, O. R. Contract as automaton: representing a simple financial agreement in computational form. *Artificial Intelligence and Law* (Oct 2021).

[3] Hashmi, M., Governatori, G., and Wynn, M. Normative requirements for regulatory compliance: An abstract formal framework. *Information Systems Frontiers 18* (05 2015).

[4] Sasdelli, D. Normative diagrams. In *Proceedings of the International Workshop on AI Compliance Mechanism (WAICOM 2022)* (12 2022), pp. 39–49.