Formalizing a simple loan agreement in L4 and Maude

Joe Watt

March 22, 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)

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 _ .

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

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].

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.

- `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.

3.2 The loan agreement in L4

(TODO: Add link)

Note 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 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".

6

```
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

This section and the others after it are still heavily in progress.

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
Cite Plotkin

theory.

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

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

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.

In the context of L4, our configurations are pairs of active rules and timers, to help us keep track of which rules are in effect, and their deadlines. Following timed transition systems , we define two kinds of events, namely action events and tick events.

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 begin by providing some background on Maude before we discuss our formalization of the simple loan agreement. Along the way, we explain how we addressed the aforementioned shortcomings. In particular, we show how we can generate an arguably more accurate DFA that accounts for the concurrent interleaving of real world events. Thereafter, we demonstrate how we can use Maude to help us visualize and reason about the contract.

5.1 Background on Maude

Techniques originating from the field of formal methods have been devised to automatically analyze the behavior of computer systems. These often rely on first modelling these systems as *labelled transition systems* (LTS), which can be seen as generalizations of nondeterministic finite automata).

While DFAs and LTSes in general are formalisms that are well suited for computers to analyze and reason about, they are cumbersome for humans to use to encode systems. This is especially so for systems like the simple loan agreement of [2] which involve concurrency, in that there are many possible ways in which events can be interleaved. Modelling concurrent systems like this directly as a transition system is often impractical as the interleaving of events gives rise to numerous states and transitions. Moreover, as previously discussed,

they leave implicit many of the domain specific concepts that people directly reason about.

Maude is a language that allows us to model and analyze such concurrent systems. As with others like it, it comes equipped with a more sophisticated formalism that allow us to more conveniently specify these transition systems.

> **TODO**
> Introduce Maude syntax and give some examples of how it works.
> 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.

5.2 Implementing the operational semantics in Maude

5.3 Visualizing and simulating the contract

Insert DFA of state space here.

6 Related work

7 Limitations and future work

The current Maude models are not very performant and have huge state spaces. One of the main reasons is that we model time naively, with a `tick` transition denoting the passage of 1 unit of time (which can be taken to be 1 day, 1 month etc). We did this because Maude does not come with any built-in support for these notions, so we wanted an easy way to model these. 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.

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.