Formalizing a simple loan agreement in mcrl2

Joe Watt

September 30, 2022

This is a work in progress.

Contents

1 Intro and shortcomings of DFA formalization

1.1 Intro

TODO
Write a better intro and provide more background on [2].

In [2], the authors claim 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, Table 1], which they formalize directly as a DFA.

1.2 Shortcomings

While this approach is a nice proof of concept, there are various shortcomings with such a formalism, perhaps the most apparent being that the manual encoding of a contract as a DFA is a laborious process. A simplified, arguably inaccurate, visual representation of the automaton [2, Fig. 1] corresponding to this simple contract already contains more than 20 states and 40 transitions. Here, we discuss 2 reasons contributing to the complexity involved with a more accurate formalization of the contract as a DFA. We claim that even for such a simple contract, a more accurate formalization as a DFA is too impractical to be carried out by hand.

The first source of complexity is a consequence of DFAs not having an explicit notion of global variables, as well as their inability to perform arithmetic computations. To see this, observe how there are verbose duplication of "Payments … accelerating" states. Note here that when we say "duplication", we do not mean that there are states which play exactly the same role from the point of view of bisimilarity or language acceptance as in the sense of the Myhill-Nerode technique of DFA minimization. What we mean instead is that these states play similar roles, and viewing the DFA as a graph, the subgraphs rooted at these states have similar structure.

Viewing these subgraphs as sub-automata, we see that they both accept similar sequences of events from the point when the borrower defaults and is obliged to make an accelerated repayment of the outstanding amount. Notice that the key difference between them is the "Payment made $\$n$" transition, corresponding to the event that the borrower pays off the outstanding amount of $\$n$. While we would like to collapse both of these subgraphs into a single one, this is not possible as the value of n varies at runtime, with the current state of the contract, ie with the number of payments the borrower has paid off previously. In other words, n can be thought of as a global variable whose actual value is updated at runtime, whenever payment events occur. However, DFAs do not have a notion of global state, nor can they perform sophisticated computations like arithmetic ones, so that these values must be manually computed by hand and then encoded in the DFA. The result of this manual encoding is the aforementioned duplication of states and transitions. Now, if the contract were to contain more than just 2 repayment stages, more manual computations would be required, resulting in even more duplicated states and transitions that would then need to be added to the DFA.

The second source of complexity arises from the concurrent interleaving of real world events. Perhaps for the sake of simplicity, this has not been accounted for in the DFA formalization. As an example of such a scenario that is not handled by the DFA, 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.* One could argue that the real world is inherently concurrent and asynchronous in nature, so that a more realistic encoding of this contract would account for such interleaving of events. Of source, doing so would significantly increase the complexity of the DFA, and consequently, the labor involved with its manual construction.

Therefore, we argue that this evinces that accurately encoding contracts as a DFA directly is too laborious and difficult, even for a contract as simple as the one presented in [2, Fig 1.]. In our view, a more practical formalism should sit at a higher-level than a DFA, providing mechanisms for tackling these 2 issues. Firstly, it should provide global variables, or at least, some way of encoding global state. There should also be operations that allow us to retrieve and update the global state. Secondly, it should be able to conveniently accommodate the concurrency inherent in the real world. Better still if the formalism comes with tools that allow us to compile down to something like automata for visualization and automated reasoning.

2 Our approach

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 `mcrl2` toolset [1], which provides a high-level modeling language based on a process algebra [3].

In this section, we begin by providing some background on `mcrl2`before we discuss our formalization of the simple loan agreement and 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 the `mcrl2` toolset to help us visualize and reason about the contract.

TODO
May also want to cover other related approaches, like Symboleo and how they use nuxmv for verification. Need to investigate how they model that further first.

2.1 Background on mcrl2

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 (NFA). As with finite automata, LTSes can be seen as directed graphs, with nodes representing states that the system can be in, and labelled edges denoting transitions that change the state of a system. Where they differ from finite automata is that they are allowed to have an infinite [1] number of states and transitions between them. They are also not required to come with a notion of initial and final states.

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.

May be good to explain what is "concurrent interleaving" a bit more.

The toolset we use, `mcrl2`, 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. In the case of `mcrl2`, the formalism it uses is a textual specification language [3] based on a variant of the process algebra known as the Algebra of Communicating Processes (ACP). It also comes with a data language that allows for us to define abstract data types built on top of primitive types like booleans and natural numbers. A specification can then be simulated and visualized using the provided tools. For automated reasoning, we can use the provided tools to verify properties expressed in a first-order modal μ-calculus. Witnesses (resp. counter-examples) can then be provided for successful (resp. failed) verifications.

As an example, one can define a binary search tree as such:

```
sort
  Tree = struct
    Empty
    | T(left : Tree, value : Nat, right : Tree);
```

[1] Possibly uncountable

Here we define a `Tree` sort [2] (ie type) of natural numbers with 2 constructors, namely `Empty` and `T`, the second of which has 3 parameters which can be projected via functions `left`, `value` and `right`. Readers familiar with functional programming will recognize that this is reminiscent of algebraic data types.

With this, we can define an insert function with the following equations.

```
map insert : Nat # Tree -> Tree;

var t, t' : Tree; n, m : Nat;

eqn
  insert(n, Empty) = T(Empty, n, Empty);
  (n < m) -> insert(n, T(t, m, t')) = insert(n, t);
  (n > m) -> insert(n, T(t, m, t')) = insert(n, t');
  (n == m) -> insert(n, T(t, m, t')) = T(t, m, t');
```

Note here that `Nat # Tree` denotes the cartesian product `Nat` × `Tree` and that the arrows `(->)` as in `(n < m) -> ...` denote optional boolean guards on the equations. As with functional programming, there is some support for writing them using pattern matching. To evaluate these equations, the toolset treats them as rewriting rules [5], oriented from left to right.

Next, we define, as processes, a binary search tree and a button which when pressed, empties the tree:

```
act
  insert : Nat;
  press, press_r, press_s;

proc
  Tree(t : Tree) =
    sum n : Nat. (
      insert(n) .
      Tree(insert(n, t))
    )
    + press_r . Tree(Empty);

  Button =
    press_s . Button;
```

The `Tree` process is parameterized over a tree `t`, indicating the current state of the tree. It has 2 actions, the first being `insert`, which is parameterized over all $n \in \mathbb{N}$. `insert(n)` indicates that `n` is inserted into the tree and `press_r`, which receives (hence the `_r`) a button press. The button then has a corresponding `press_s` which sends (hence the `_s`) such a press to the tree. These are composed together using the (nondeterministic) choice operator `+` so that the tree may either have something inserted into it or receive a button press at each step. Briefly, `sum n : Nat. ...` is used to indicate a choice over all

[2] The term "sort" comes from multi-sorted logics since sorts in `mcrl2` are interpreted logically as such.

possible natural numbers, so that any natural number can be inserted into the tree. The dot (.) following insert(n) denotes sequential composition with the tail recursion Tree(insert(n, t)), which update the internal state of the tree.

Finally, we can define the initial process as a parallel composition (· || ·) of the Tree process, beginning with an empty tree and the button, Button.

```
init
  allow({insert, press},
  comm({press_r|press_s -> press},
  Tree(Empty) || Button));
```

Since we only want the tree to empty itself when the button is pressed, and not at any other times, we use allow and comm(unicate) to enforce communication. Simply put, this means that the press_s and press_r in both processes must be executed together in the same atomic step. Here, press_r|press_s is a *multi-action* which is an atomic action representing the simultaneous execution of both the individual actions.

One can then use the tools provided by the mcrl2 toolset to automatically transform this specification into a LTS. Note that the resulting transition system contains infinitely many states as each n $\in \mathbb{N}$ gives rise to a transition labelled insert(n). Thus, we see that the mcrl2 toolset provides us with a convenient, high level formalism with which we can concisely express large (infinite in this case) transition systems.

2.2 Formalizing the contract in mcrl2

2.2.1 Generic parameters

As we wanted our specification to be modular, we began by defining the following global constants and functions over which the rest of our specification is parameterized.

```
map
  % The total number of payments involved in the contract.
  total_num_payments : Pos;

  % payment_amt(n) is the amount the borrower has to pay for payment n.
  payment_amt : Pos -> Pos;

  % The principal amount the lender sends the borrower.
  principal_amt : Pos;

  % The initial amount that the borrower owes the lender.
  initial_outstanding_amt : Pos;

var n : Pos;

eqn
  total_num_payments = 2;
  principal_amt = 1000;
```

```
initial_outstanding_amt = 1075;

(n == 1) -> payment_amt(n) = 550;
(n == 2) -> payment_amt(n) = 525;
```

These can be easily changed for instance to account for additional payment stages, or to incorporate interest rate calculations into `payment_amt`. We only require that:

1. `payment_amt` is a partial function defined on the domain:

$$\operatorname{dom}\text{payment\_amt} = \{n \in \mathbb{N} \mid n \leq \text{total\_num\_payments}\}$$

2. The sum of all payment amounts is equal to the initial outstanding amount, that is:

$$\sum_{n \in \operatorname{dom}\text{payment\_amt}} \text{payment\_amt}(n) = \text{initial\_outstanding\_amt}$$

As the rest of our specification makes no further assumptions about the values of these constants and the computation performed by `payment_amt`, changing these do not require any further changes in the structure of the specification. In comparison, if one were to add additional payment stages to the DFA in [2] directly, one will need to manually alter the structure of the DFA by introducing more states and transitions. This highlights one advantage of our approach.

2.2.2 Events and contract stages

To proceed with our formalization, we decided to model events as atomic actions in `mcrl2`. For instance, we have the actions `borrower_default` for borrower defaults and `pay` for payments. Note here that for simplicity, we have ignored multiple kinds of defaults, preferring instead to use a single event to indicate defaults by representations, warranties, covenants and so on. We do however, have a separate action `payment_default` for payment default events. These payment events include normal repayments, accelerated repayments, as well as principal repayments. We also have the actions `fulfilled` and `breached`, which as their names suggest, represent the events where the contract has terminated in a fulfilled or breached state.

We then observe that the contract has 4 key stages, namely:

1. **Initialization**

   The contract begins in this stage, where the borrower may request for the principal amount. Upon such a request, the lender is obliged to send it. Thereafter, the process starts the main and side tracks below simultaneously, before terminating.

2. **Main repay payments track**

   This process handles all the payments that the borrower makes to the lender. Once these are completed, the side track process is terminated and the contract as a whole is fulfilled. Should the borrower default on a payment, the side track is stopped and the accelerated repayment process is started.

3. **Side borrower default track**

   When this process starts running, it waits for the occurrence of a borrower default event. If that happens, and the borrower fails to cure the default, the main track is stopped and the accelerated repayment process is started.

4. **Accelerated repayment**

   As described above, this is triggered when things go wrong. It handles the accelerated repayment stage, in which the borrower is obliged to repay the outstanding amount immediately.

TODO
Draw and insert picture.

These are modelled in `mcrl2` as the processes `Init`, `Main`, `Side` and `Accel` respectively. `Init` is defined as follows:

```
Init =
  dont_request_principal . fulfilled
  + request_principal . (
      payment_default(Principal, Lender, Borrower, principal_amt) .
      breached
      + pay(Principal, Lender, Borrower, principal_amt) .
        start_main_s . start_side_s
    );
```

At the start of the contract, the borrower has the option of requesting for the principal amount. `dont_request_principal` is an action modelling the event where the borrower does not request for it on the day the loan agreement comes into effect. In this case, the contract immediately terminates in a fulfilled state. If the principal is requested for, ie the `request_principal` action is executed, then the lender is obligated to send it. The contract is breached in the event that the lender defaults on this payment, ie the `payment_default(Principal, Lender, Borrower, principal_amt)` action is executed. On the other hand, if the lender does send the principal amount, so that `pay(Principal, Lender, Borrower, principal_amt)` runs, then the actions `start_main_s` and `start_side_s` will be run. These communicate with the corresponding `start_main_r` and `start_side_r` actions which start the `Main` and `Side` processes.

Note here that we would have preferred to write:

```
pay(Principal, Lender, Borrower, principal_amt) .
(Main || Side)
```

However, the `mcrl2` tools do not allow for parallel composition nested under sequential composition. Parallel composition can only be used at the top-level, so that we had to lift it up via:

Listing 1: Top-level process version 1

```
init
  allow({start_main, start_side},
  comm({
    start_main_s|start_main_r -> start_main,
    start_side_s|start_side_r -> start_side,
  },
    Init || start_main_r . Main || start_side_r . Side;
  ))
```

2.2.3 Tracking global state

As `mcrl2` does not allow for global mutable variables as found in imperative programming language, we cannot simply create global variables to track the state of the contract, namely the remaining number of payments and outstanding amount.

Fortunately, [4, Section 3.2.2] describes a simple, common technique for this. We define a new `Outstanding` process which keeps track of these variables, which communicates with all the other processes, like `Main`, `Side` and `Accel`. Intuitively, payment events that occur in these other processes will cause these variables to be updated here and when accelerated repayment events occur in `Accel`, this process can be queried to obtain the current outstanding amount.

```
proc Outstanding(remaining_payments : Nat, outstanding : Nat) =
  remaining_payments_r(remaining_payments) .
  Outstanding(remaining_payments, outstanding)

  + next_payment_number_r(
      Int2Pos(max(1, total_num_payments - remaining_payments + 1))
    ) .
    Outstanding(remaining_payments, outstanding)

  + outstanding_r(outstanding) .
    Outstanding(remaining_payments, outstanding)

  + (outstanding > 0) -> (
      pay_r(Accelerated, Borrower, Lender, Nat2Pos(outstanding)) .
      Outstanding(0, 0)
    )

  + sum amt_paid : Pos. (
      pay_r(Normal, Borrower, Lender, amt_paid) .
      Outstanding(
        remaining_payments = Int2Nat(max(0, remaining_payments - 1)),
        outstanding = Int2Nat(max(0, outstanding - amt_paid))
      )
    );
```

Whenever a pay_s event corresponding to a borrower repayment occurs in the Main process, the corresponding pay_r event triggers, causing the Outstanding process to recurse with the updated variable values. The actions remaining_payments_r, next_payment_number_r and outstanding_r allow other processes to query this process for the remaining number of payments, next payment number and outstanding amount.

2.2.4 Main payment loop

TODO
It's worth mentioning how this explicates how we can model process interruption as a meta-level operator in process algebras that don't have them built in.

The Main payment process communicates with the Outstanding process, looping through all repayment stages. Recall that we want this Main process, along with the Side track process, to be interruptible because we want to terminate them before switching over to the Accel process for accelerated repayments. Consequently, we first define a wrapper process, Main_idling that represents the Main track process in an idle state:

```
proc
  Main_idling =
    stop_main_r . Main_idling + start_main_r . Main . Main_idling;
```

This idle process listens for start and stop signals in the form of start_main_s and stop_main_s actions. If a stop signal is received via a stop_main_r action, the Main_idling process continues to idle as indicated by the recursion. When start_main_s is run in the Init process, it communicates with this start_main_r action, kicking the Main_idling process out of its idle state so that the Main process begins to run. Thereafter, the process goes back to its idle state.

We define the Main process using tail recursion as in functional programming to express the looping structure:

```
proc
  Stop_main_r = stop_main_r . Main_idling;

  Main = sum n : Pos.
    next_payment_number_s(n) .
```

```
(n > total_num_payments) ->
  stop_side_s . fulfilled
<> (
  (pay_s(Normal, Borrower, Lender, payment_amt(n)) + Stop_main_r) .
  Main

  + (payment_default(Normal, Borrower, Lender, payment_amt(n))
      + Stop_main_r) . (
    (dont_notify_borrower + Stop_main_r) . Main

    + (notify_borrower_of_default + Stop_main_r) . (
      (cure_default + Stop_main_r) . Main

      + (dont_cure_default + Stop_main_r) .
        stop_side_s . Accel
      )
    )
  );
```

Notice that all actions corresponding to events like `pay_s` and `cure_default` are annotated with a choice `+ Stop_main_r`. This allows the `Main` process to listen out for a stop signal and transition back to the idling state upon receiving one. More precisely, `stop_main_r` communicates with `stop_main_s` actions, causing it to execute instead as per the idle `Main_idling` process.

TODO
Should probably mention about how we hide those actions and prioritize tau transitions in the LTS to give them priority.

Observe that we first perform `next_payment_number(n)`, which as mentioned earlier, communicates with the `Outstanding` process to determine what the next payment number is. `(n > total_num_payments) -> ...<> ...` is a conditional if-then-else that checks if all payments have been made. If so, the side track is stopped via the `stop_side_s` action and the `fulfilled` action is performed. Otherwise, the borrower is obligated to pay the lender. In the event that he does, `pay_s` is run and communicates with the corresponding `pay_r` action in the `Outstanding` process to update the state of the contract.

Suppose now the borrower defaults on the payment as indicated by the `payment_default` event. Then, if he is not notified by the borrower or eventually cures his default, he does not get into trouble, so we recurse to the start of the `Main` process and continue handling the remaining payments. If instead he is notified but fails to cure his default in time, `stop_side_s` is run, stopping the `Side` track process. Thereafter, we proceed with the accelerated repayment stage via the `Accel` process.

Some notes:
Talk about how the main repayment track is recursive, modelling a loop that iterates through the number of payments involved in the loan agreement.
While `mcrl2` does not have a notion of global variables like other formalisms (mention other formalisms here, maybe uppaal?), these can be easily simulated. Our specification defines "global variables" representing the current runtime state of the contract, namely the remaining number of payments as well as the outstanding amount. These variables are then updated whenever a normal repayment event occurs, and referenced during accelerated repayment event.
Finally, our formalization also allow for more events to be interleaved concurrently. In particular, we make no assumption that the borrower is notified immediately after a default event occurs. Payment events or even other default events may occur in the meantime, so that in the latter case, the borrower has multiple defaults to cure.

2.3 Visualization and simulation

2.4 Background on the modal mu calculus in mcrl2

2.5 Automated reasoning with the contract

3 Related work

4 Limitations and future work

1. No explicit notion of real time, ie no global clock process.

2. We don't keep track of blame assignment, ie who is in breach of the contract.

3. Lack of native notion of process interruption, and inability of linearizer to handle parallel composition under sequential comp and recursion, ie `A . (B || C)`, makes modelling a bit inconvenient and unnatural.

4. Squeezing out deontics completely means we have no way to really reason about obligations and permissions.

   Explore Guido's work on why deontics cannot be adequately represented in temporal logics, LTL in particular.

   Eg: if we take

   (a) Maintenance obligation ⇒ Always

   (b) Achievement obligation ⇒ Eventually

then how to handle permission?

Also, how to handle reparation/compensation?

5. Connecting this to l4.

   `mcrl2` is a capable tool but not exactly what we want to use, because it lacks some higher level conveniences that we would like to have. These include things like global state, global clock for time and interrupts.

   Still, this experiment gives us a good idea of what kind of features we want for l4's semantics, specifically if we want to use some process algebra for it. Moreover, it also demonstrates how we can model these in `mcrl2`. While global state, time and interrupts aren't too difficult to handle, the inability to express parallel composition like `A. (B || C)` suggests that we may need to do some non-trivial encoding into `mcrl2`.

   There seems to be various works out there about formalizing semantics of various things like UML state machines in various tools like `mcrl2` and beyond. Should look into this for clues on how we can model l4's semantics.

References

[1] Bunte, O., Groote, J. F., Keiren, J. J. A., Laveaux, M., Neele, T., de Vink, E. P., Wesselink, W., Wijs, A., and Willemse, T. A. C. The mcrl2 toolset for analysing concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems* (Cham, 2019), T. Vojnar and L. Zhang, Eds., Springer International Publishing, pp. 21–39.

[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] Groote, J. F., and Mousavi, M. R. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[4] Remenska, D., Willemse, T. A., Verstoep, K., Templon, J., and Bal, H. Using model checking to analyze the system behavior of the lhc production grid. *Future Generation Computer Systems 29*, 8 (2013), 2239–2251.

[5] van Weerdenburg, M. An account of implementing applicative term rewriting. *Electronic Notes in Theoretical Computer Science 174*, 10 (2007), 139–155. Proceedings of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2006).