Executing Finance in Metric Temporal Logic

A Formal Verification Schema

Quinn Dougherty Matthew Doty Mark Greenslade

August 17, 2024

Abstract

A financial contract is an agreement that governs monetary interactions between parties, with the interactions themselves being part of the contract's execution. Complex interactions among these parties may result in misunderstandings and errors, potentially leading to financial losses for all involved—an undesirable outcome. Previously, the Algorithmic Contract Types Unified Standard (ACTUS) attempted to tackle this twin problem by rendering interactions as atomic state machines that compose together. In this paper, we take a natural next step to Metric Temporal Logic (MTL) and take a contract to be an MTL *specification*, showing the proof of concept on the principal at maturity loan (PAM). Along the way, we implement MTL in the Lean prover. The code is available at https://github.com/cspr-rad/mtl-actus.

Casper Association[1]

1 Introduction

Financial contracts, at their core, are interactions between people involving money. However, the execution of these interactions is increasingly reliant on code, which introduces the risk of bugs - a problem that has led to significant financial losses. Moreover, as financial instruments become more exotic, they often become too complex to interpret easily, further complicating the situation.

The Algorithmic Contract Types Unified Standard (ACTUS) is an attempt to address this dual challenge. It aimed to simplify financial interactions by representing them as atomic state machines that could be composed together. However, this approach, while innovative, has its limitations especially around composite contracts.

We propose a step forward by leveraging ideas from the field of model checking. Specifically, we suggest using Metric Temporal Logic (MTL) as a foundation for defining financial contracts. In this framework, a contract becomes an

[1]Corresponding author: quinn@casper.network

MTL specification. This approach offers two key advantages: firstly, MTL specifications can serve as an interpretable grounding for even the most complex financial instruments. Secondly, contracts can be rigorously checked to ensure they comply with these MTL specifications. This methodology has the potential to significantly improve both the reliability and interpretability of financial contracts, addressing the core issues that have plagued the industry.

1.1 Contributions

The current paper shows in principle how to model finance as a reactive system for the purposes of formal verification, on an example Principal at Maturity (PAM) contract.

1. Reactive system approach: We demonstrate how financial contracts can be modeled as reactive systems, a paradigm commonly used in computer science for systems that maintain ongoing interactions with their environment. This novel perspective allows us to apply well-established formal verification techniques from the realm of reactive systems to financial contracts.
2. Formal verification framework: We introduce a framework for the formal verification of financial contracts. This approach enables rigorous mathematical proof of contract properties, potentially reducing errors and ambiguities in contract execution.
3. Principal at maturity (PAM) case study: To illustrate our methodology, we provide a detailed case study of a Principal at Maturity (PAM) contract. This common financial instrument serves as an accessible yet meaningful example to showcase the power and applicability of our approach.
4. Temporal logic specification: We demonstrate how to express financial contract terms and conditions using MTL. This formal language allows for precise and unambiguous specification of time-dependent behaviors, which are crucial in financial contracts.

2 Prior Work

2.1 Algorithmic Contract Types Unified Standard

In [14], the ACTUS (Algorithmic Contract Types Unified Standard) introduces a formalized approach to representing financial contracts using concepts from state machine theory. Developed by the ACTUS Financial Research Foundation, this framework models financial instruments as algorithmic contracts with well-defined states, events, and transition functions.

A key contribution of ACTUS is its event-based architecture for financial modeling. Each contract type is defined by a set of state variables and a series of event types. These events trigger state transitions and cash flows according to precisely specified functions. This approach allows for a representation of

financial contracts over their lifecycle, enabling accurate projection of future states and cash flows.

ACTUS formalizes the temporal aspects of financial contracts through standardized scheduling functions and date adjustment conventions. This allows for precise modeling of complex temporal patterns in financial agreements, such as irregular payment schedules or conditional events.

By applying state machine concepts to finance, ACTUS provides a rigorous mathematical foundation for representing and analyzing financial contracts. This standardization facilitates more precise financial modeling, risk assessment, and regulatory analysis across a wide range of financial instruments.

2.2 Algebraic and Inductively Defined Financial Interaction

[12] defined an algebra for contracts of two parties. In [9] a contract for more than two parties can be well-typed. This line of work is an alternative solution to the ACTUS-like solution, namely the typechecker, to a very similar problem set around correctness and interpretability of finance.

The seminal work on formalizing financial contracts using functional programming techniques was done by [12]. They introduced a combinator library for describing contracts declaratively, along with a compositional denotational semantics for valuing contracts. Their approach allowed complex contracts to be built up from simpler components in a modular way. Importantly, they separated the abstract specification of contracts from the concrete implementation details of valuation. This allowed different valuation models and numerical methods to be used without changing the contract descriptions.

Building on this foundation, [9] developed a certified symbolic management system for multi-party financial contracts. They extended the contract language to handle more than two parties and added a formal cash-flow semantics. Crucially, they implemented the contract language and management functionality in the Coq proof assistant, allowing them to formally verify properties like causality and provide certified contract analyses and transformations.

3 MTL ACTUS in Lean 4

Lean4 serves as our implementation language for the project. Lean4 [7] is an open-source theorem prover and programming language. Developed by Microsoft Research and the Lean community, it excels in mathematical formalization and formal verification. For our work on financial contracts, Lean4's strong type system is helpful for implementing MTL and reasoning about specifications and executions. Lean's syntax, designed to be close to mathematical notation, allows us to clearly express and verify our MTL formalization of financial contracts.

3.1 Money and time

Unlike ACTUS, we consider the hard part of time to be a presentational step taken later, and content ourselves with something like Unix Time [13] (a newtype of whole numbers). We do something similar with money.

```
inductive Timestamp : Type where
  | t : UInt64 -> Timestamp
  | infinity : Timestamp
  deriving BEq, Hashable, Repr, DecidableEq
...
def Window : Type := (Timestamp × Timestamp)
...
structure Money where
  amount : Int
  deriving BEq, Hashable, Repr, DecidableEq
```

Money is best represented as an `Int` [6], and you fix a precision (some degree of fractions of a cent) to place the decimal point later.

3.2 Metric temporal logic

Temporal logic [10] is widely used in model checking to verify reactive systems [2]. MTL is an extension of temporal logic that incorporates quantitative time constraints [8]. It allows for the specification of time-bound properties in real-time systems, making it particularly useful for describing and verifying time-sensitive behaviors. MTL formulas can express properties such as "event A must occur within 5 time units of event B" or "condition C must hold for at least 10 time units." This makes MTL well-suited for modeling and analyzing systems where precise timing is crucial, such as financial contracts with specific execution deadlines or duration-based clauses.

We give MTL in the usual way of an inductive data type. The modal constructors accept additionally a window in their input.

```
variable {T : Type} [AtomicProp T]

inductive Proposition (T : Type) : Type where
| tru : Proposition T
| atom : T -> Proposition T
| negate : Proposition T -> Proposition T
| conjunct : Proposition T -> Proposition T -> Proposition T
| until : Proposition T -> Window -> Proposition T -> Proposition T
| since : Proposition T -> Window -> Proposition T -> Proposition T
```

With `disjunct` derived from `negate` and `conjunct`, and the unary temporal operators given in the usual way as combinations of the binary temporal operators `since` (S) and `until` (U) with `tru` (the value true $\top$). In particular, `eventually` ($\Diamond_w \phi$) is given $\top U_w \phi$ and `always` ($\Box_w \phi$) is given $\neg \Diamond_w \neg \phi$. We additionally have

the temporal operator `next` ($\circ_w \phi$) from $(\neg\top)U_w\phi$, saying that ϕ is in the subsequent timestamp. In this MTL, indexing an operator with the window `(Timestamp.infinity, Timestamp.infinity)` means a degenerate case where the temporal operator simplifies to a Linear Temporal Logic (LTL) operator.

We use Lean's notations feature to make these operators easier to construct.

```
notation phi "and" psi => Proposition.conjunct phi psi
notation phi "U" psi "in" w => Proposition.until phi w psi
notation phi "S" psi "in" w => Proposition.since phi w psi
def or (phi psi : Proposition T) : Proposition T := ~ (~ phi and ~ psi)
def eventually (w : Window) (phi : Proposition T) : Proposition T := tru U phi in w
def always (w : Window) (phi : Proposition T) : Proposition T := ~ (eventually w (~ phi))
def next (w : Window) (phi : Proposition T) : Proposition T := mtf U phi in w
notation "{\Diamond" w "}" phi => eventually w phi
notation "{\Box" w "}" phi => always w phi
notation "{\circ" w "}" phi => next w phi
```

3.3 Automata

In the context of MTL, automata play a crucial role in both theoretical analysis and practical applications. Timed automata, in particular, are closely related to MTL as they provide a way to model and verify real-time systems with quantitative timing constraints [1].

The relationship between MTL and automata is significant for several reasons:

1. Decidability and complexity: Automata-based techniques are often used to establish decidability results and complexity bounds for various fragments of MTL.
2. Model checking: Timed automata can be used as an operational model against which MTL specifications can be verified, enabling efficient model checking algorithms.
3. Expressiveness: The expressive power of different MTL fragments can be characterized by corresponding classes of timed automata, providing insights into the logic's capabilities.
4. Synthesis: Automata-theoretic approaches can be employed to synthesize controllers or implementations that satisfy given MTL specifications.

We implement the usual timed automata, finite case (with accepting states), which involves natural number values called *clocks* ticking time. Every transition comes with a set of guards and a list of which clocks to reset to zero upon taking that transition.

```
structure State where
  idx : Nat
  deriving BEq, Hashable, Repr

variable (Alphabet : Type) [AtomicProp Alphabet]
```

```
structure Transition where
  source : State
  target : State
  symbol : Alphabet
  guards : GuardConditions
  reset : List ClockVar
  deriving BEq, Hashable

structure TFA where
  states : Lean.HashSet State
  alphabet : Lean.HashSet Alphabet
  initialState : State
  transitions : List (Transition Alphabet)
  acceptingStates : Lean.HashSet State
```

The timed finite automata (TFA) type is defined in the usual way, where a guarded and labeled transition is better represented as a struct than a function.

3.4 Contracts

A contract consists of a type for the terms that the counterparties agree to and a type of events. With that, you give a *contract* as an MTL formula (or spec) along with it's corresponding automaton.

```
structure ActusContract where
  terms : Type
  event : Type
  event_atomicprop : AtomicProp event
  contract : terms -> Proposition event
  automaton : terms -> TFA event
```

3.4.1 PAM

The schema is demonstrated in a proof of concept for Principal at Maturity (PAM). A PAM is a loan with periodic interest payments, but the principal is not paid at all until the maturity date, which closes the contract.

```
-- payment interval assumed to be 1
structure Terms where
  principal : Money
  interest_rate : Timestamp -> Scalar -- fixed rate as constant function
  start_date : Timestamp
  maturity : TimeDelta

inductive Event :=
| Maturity : Event
| PrincipalRepayment : Event
```

```
| InterestPayment : Event
  deriving BEq, Hashable, Repr, DecidableEq

def Contract := Proposition Event deriving BEq, Hashable, Repr
```

A modest safety property is that it is not maturity date until at least one interest payment.

$$S := (\neg \texttt{Maturity}) U_{cl} \texttt{InterestPayment}$$

Where cl represents the time window of the whole contract length, `(start_date, start_date + maturity)`.

Another important factor if we're going to capture PAM behavior in the spec is that interest payments are continuous until maturity

$$C := \Box_{cl}(\texttt{InterestPayment} \rightarrow (\circ_{cl}(\texttt{InterestPayment} \vee \texttt{PrincipalRepayment})))U_{cl}\texttt{Maturity}$$

So our *contract specification* for PAM can be written

$$S \wedge C$$

Which looks and lean4 precisely as

```
def safety (terms : Terms) : Contract :=
  let cl := contract_length terms;
  (~ [[Event.Maturity]]) U [[Event.InterestPayment]] in cl

def ip_continuous_till_mat (terms : Terms) : Contract :=
  let cl := contract_length terms;
  {\Box cl} ([[Event.InterestPayment]] implies
      ({○ cl.incr_start {dt := 1}}
        ([[Event.InterestPayment]]
        or [[Event.PrincipalRepayment]])
      )
    )
    U [[Event.Maturity]] in cl

def contract (terms : Terms) : Contract :=
    ip_continuous_till_mat terms and safety terms
```

Omitted here for brevity is the PAM TFA template.

3.4.2 Accepting words

A word is an element of a language induced by an automaton. In this work, we would hope to draw out words from execution traces of running contracts. Then, checking if a contract automaton *accepts* that word amounts to verifying that the contract ran correctly. This approach allows us to model smart contract behavior as a formal language and leverage automata theory for verification. By

transforming contract executions into words and defining acceptance criteria, we can systematically analyze contract correctness and detect potential violations or unexpected behaviors.

4 Future work

4.1 Implementing the remaining ACTUS taxonomy

There is a folk wisdom among ACTUS circles that PAM is 80% of the work. However, it still remains for us to implement the other seventeen contracts in the ACTUS taxonomy. Since the specification language is monadic in event types, we may get a clean and elegant way to compose the atomic contracts into more intricate instruments. This modular approach not only simplifies implementation but also enhances flexibility, allowing for the creation of new, custom financial products by combining existing contract types. As we expand our implementation to cover the full ACTUS taxonomy, we'll gain a more comprehensive toolkit for financial modeling and risk assessment across diverse asset classes and contract structures.

4.2 Assume-guarantee contracts

Another obvious approach to finance would be assume-guarantee contracts [4] [11]. A rough sketch of what this would look like would be showing invariants not as temporal logic specs but as contracts with preconditions and postconditions. This would allow for clearer specification of obligations and expectations in financial transactions. Such a framework could potentially improve formal verification of financial systems and contracts, enabling better risk assessment and compliance checking.

4.3 Lustre and Kind2

A promising direction for future work is the implementation of ACTUS using the Lustre programming language [5], with verification via the Kind2 tool [3]. This approach could significantly advance formal methods in financial contract modeling and verification.

Lustre is a synchronous dataflow programming language designed for reactive systems that require real-time execution and high reliability. Originally developed for critical systems in domains like avionics and automotive control, Lustre's paradigm aligns well with the state-machine nature of ACTUS contracts. Of particular interest is Iowa Lustre, an extended version of the language that offers enhanced expressiveness through features like array support and an improved type system. Kind2 is a powerful, open-source formal verification tool specifically designed for Lustre programs. It combines various model checking techniques to verify safety properties of synchronous systems. Using Kind2 and Lustre

together to develop and verify critical reactive systems is an active area of research engineering.

In the context of financial contracts, we envision the following steps:

1. ACTUS in Lustre: Implement the core ACTUS contract types as Lustre modules. Each contract type would be represented as a synchronous dataflow program, with its state transitions and temporal behaviors explicitly modeled.
2. Formal specification: express contract properties, regulatory requirements, and desired behaviors as formal specifications in Lustre's assertion language.
3. Verification with Kind2: utilize Kind2 to formally verify the Lustre implementation against these specifications. This could include proving the absence of certain types of errors, ensuring compliance with regulatory requirements, and verifying key properties of contract behavior.
4. Composition and scalability: Leverage Lustre's modularity to compose complex financial instruments from simpler components, and investigate the scalability of this approach to large-scale financial systems.
5. Real-time analysis: exploit Lustre's real-time capabilities to model and analyze time-critical aspects of financial contracts, such as payment deadlines or market-responsive behaviors.

Implementing ACTUS using Lustre and verifying it with Kind2 represents a promising frontier in formal methods for financial contract modeling. This approach could significantly enhance the reliability and trustworthiness of financial software systems by leveraging the strengths of synchronous programming and advanced verification techniques. By rigorously formalizing contract behaviors, ensuring compliance with specifications, and enabling scalable composition of complex financial instruments, this direction of research has the potential to revolutionize how we design, verify, and reason about financial contracts in an increasingly complex and interconnected global economy.

5 Conclusion

In this paper, we have taken a preliminary step towards addressing two persistent challenges in financial technology: the risk of code bugs in contract execution and the increasing complexity of financial instruments. While the Algorithmic Contract Types Unified Standard (ACTUS) previously attempted to address these issues by modeling financial interactions as composable state machines, we recognized the potential for further improvement.

Our contribution lies in exploring the application of MTL to financial contract specification. Using a Principal at Maturity (PAM) loan as a simple yet illustrative example, we have demonstrated how MTL can be used to express contract terms and behaviors. We implemented this approach using the Lean prover, providing a proof of concept for representing financial contracts as MTL specifications.

This work represents an initial exploration rather than a comprehensive solution. By treating a financial contract as an MTL specification, we aim to provide a foundation for more rigorous verification and potentially improved interpretability. However, we acknowledge that significant work remains to be done to fully realize these benefits and to extend this approach to more complex financial instruments.

Our implementation of MTL in Lean, while functional for our example, is limited in scope and would require substantial expansion to handle a wider range of financial contracts. Moreover, the practical implications of this approach for the financial industry remain to be fully explored and validated. Looking ahead, this work opens up several avenues for future research. These include extending the MTL framework to cover a broader range of financial instruments, investigating the scalability of this approach to more complex financial ecosystems, and exploring how this method might integrate with existing financial systems and regulatory frameworks.

In conclusion, while our work on applying MTL to financial contracts shows promise, it is best viewed as a starting point for further research rather than a fully developed solution. We hope that this initial exploration will stimulate further discussion and investigation at the intersection of formal methods and financial technology, potentially contributing to the development of more reliable and interpretable financial systems in the future.

6 Acknowledgements

Thank you to Yves Hauser and Morgan Thomas for comments on a draft.

7 Bibliography

[1] Alur, R. and Dill, D.L. 1994. A theory of timed automata. *Theoretical Computer Science*. 126, 2 (1994), 183–235. DOI:https://doi.org/https://doi.org/10.1016/0304-3975(94)90010-8.

[2] Baier, C. and Katoen, J.-P. 2008. Principles of model checking. (2008).

[3] Champion, A. et al. 2016. The kind 2 model checker. *Computer aided verification* (Cham, 2016), 510–517.

[4] Girard, A. et al. 2022. Invariant sets for assume-guarantee contracts. *2022 IEEE 61st conference on decision and control (CDC)* (2022), 2190–2195.

[5] Halbwachs, N. et al. 1992. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*. 18, 9 (1992), 785–793. DOI:https://doi.org/10.1109/32.159839.

[6] Kerckhove, T.S. 2022. How to deal with money in software.

[7] Moura, L.M. de and Ullrich, S. 2021. The lean 4 theorem prover and programming language. *CADE* (2021).

[8] Ouaknine, J. and Worrell, J. 2008. Some recent results in metric temporal logic. *International conference on formal modeling and analysis of timed systems* (2008).

[9] Patrick Bahr, M.E., Jost Berthold 2015. Certified symbolic management of financial multi-party contracts. *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. (2015).

[10] Pnueli, A. 1977. The temporal logic of programs. *18th annual symposium on foundations of computer science (sfcs 1977)* (1977), 46–57.

[11] Saoud, A. et al. 2021. Assume-guarantee contracts for continuous-time systems. *Automatica*. 134, (2021), 109910. DOI:https://doi.org/https://doi.org/10.1016/j.automatica.2021.109910.

[12] Simon L. Peyton Jones, J.S., Jean-Marc Eber 2000. Composing contracts: An adventure in financial engineering (functional pearl). *ACM SIGPLAN international conference on functional programming* (2000).

[13] Wikipedia contributors 2024. Unix time — Wikipedia, the free encyclopedia.

[14] Willi Brammertz, A.I.M. 2019. Smart contracts, distributed ledgers, and the need for an algorithmic financial contract standard. *Banking & Insurance eJournal*. (2019).