University College London

Representing the Temporal Semantics of Financial Derivatives Contracts

Varun Mathur

MEng Computer Science
Submitted: April 29, 2019

Supervisor
Dr. Christopher D. Clack

This report is submitted as part requirement for the MEng Degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

Contract formalisation will enable the creation of smart contracts which in turn could be used to automate a huge volume of commercial transactions, reduce errors and inefficiencies, and enhance productivity. Unfortunately, the problem of developing such a formalism is highly complex, has a very large scope, and comprises numerous sub-problems with non-obvious solutions. This project addresses a subset of those problems: namely it examines the problem of formalising the temporal aspects of contracts. The scope is further reduced by limiting the contract domain to financial derivatives contracts. A thorough analysis of prior work is done. The temporal aspects of existing smart contract formalisms are examined. In addition, point-based and interval-based temporal logics are also examined. Then a set of requirements for the temporal aspects of smart contract formalisms is drawn up. It is based on the requirements set out by Clack and Vanca [1], with some modifications. The prior work is then evaluated against these updated requirements. The problems that are identified at this stage are then used to motivate some of the features of the new formalism. The new formalism is an interval-based temporal logic where intervals are bounded by points in absolute-time. The location of these points in time is specified with respect to a globally recognised timing system with familiar time quantities. Various additional features are introduced, motivated, thoroughly explained. The model is then evaluated with respect to the updated requirements, and it is shown that the model is able to meet most of the requirements. However, some compromises were made in designing the model and ambiguities remain. Work must be done to further evaluate the model and test if alternative solutions perform better.

I would like to thank Dr Christopher Clack for his consistent support and guidance throughout this project.

Contents

Chapter 1

Introduction

Contracts are legally binding documents that govern agreements between two or more parties. Generally, contracts consist of a set of obligations that each party must satisfy. These obligations typically involve a transfer of resources or performance of services. Contracts are a crucial part of doing business. Consequently, businesses are heavily invested in the task of *Contract Life-cycle Management* (CLM). CLM encapsulates all the various processes and operations that go into supporting a contract throughout its life including: contract creation, negotiation, approval, execution, and analysis. A study by the Aberdeen Group [2] found that 80% of businesses were manually performing some or all of these tasks. They found that standardising, formalising, and ultimately automating these processes would result in cost savings from greater efficiency. In their analysis of the global financial derivatives industry, The International Swaps and Derivatives Association (ISDA) [3] had similar findings. Therefore, there is a clear need for a solution that would allow us to automate some or all of these CLM processes.

Smart contracts are a solution to this problem. Clack et al [4] offer this definition of a smart contract:

> "A smart contract is an automatable and enforceable agreement. Automatable by computer, although some parts may require human input and control. Enforceable either by legal enforcement of rights and obligations or via tamper-proof execution of computer code"

Smart contracts would allow us to automate and monitor the performance of legal agreements electronically. For example, smart contracts could standardise and automate the performance of actions that occur over the lifetime of a financial derivative contract, reducing infrastructure cost. To start with, only a small part of the legal agreement would be automated, but as technology improves these agreements would be increasingly automated,

including automatic detection of non-performance. Additionally, the use of smart contract templates [4, 5, 6] would simplify the task of generating complex legal agreements between parties.

The first step in developing smart contracts is formalisation. This is the conversion of a contract into a formal representation that is unambiguous and machine interpretable. This conversion must happen in a manner such that we can be assured that the semantics of the contract and that of the formal representation are identical. Once a contract has been converted to this form, we could write computer code to automate its performance. This code could easily make reference to the features of the contract as specified in its formal representation. In this paper, the contract domain is limited to financial derivatives contracts that have been generated according to the ISDA Master Agreement template. The contents of a contract can be broadly categorised into three groups:

- The **deontic** aspects of the agreement; these are expressions related to obligations, rights, and prohibitions.
- The **operational** aspects of the agreement; these are expressions related to the actions that are performed by the parties to the contract.
- The **temporal** aspects of the agreement; these are expressions related to time and are typically found as annotations to the deontic or operational aspects of the contract.

This paper is focused on formalising the temporal aspects of contracts.

Objectives

The primary objective of this work is to motivate, present, and evaluate a new temporal formalism that can be used to model the temporal aspects of financial derivative contracts. To achieve this, the following sub-objectives must be met:

- Develop a suitable set of requirements against which to evaluate temporal formalisms. The requirements drawn up by Clack and Vanca [1] provide a good base, but they must be evaluated and updated and then used to evaluate prior work.
- Evaluate existing temporal formalisms and highlight the deficiencies that could be addressed by the new formalism.
- Evaluate the suitability of interval-based temporal logics for the task, since such formalisms are quite different to the temporal formulations found in existing smart contract formalisms.
- Develop a new temporal formalism that addresses the problems found in prior work. Each novel aspect of this formalism must be well motivated and thoroughly explained.

- Evaluate the formalism with respect to the requirements that are developed.
- Assess how the new temporal formalism could be integrated into a wider smart contract formalism.

Chapter 2

Prior Work

2.1 Smart Contract Formalisms

Hvitved [7] reviews state of the art smart contract formalisms and evaluates them with respect to their ability to model contracts. The temporal aspects of these formalisms are presented in this section and evaluated in chapter 3.

2.1.1 Event Condition Action

Goodchild et al. [8] propose a structured language based on XML for specifying business to business contracts. In this system a *policy* is an obligation or permission for a *role* (party) to perform some action under some conditions.

Temporal constraints are represented by including them in the policy conditions. In this model, the primary temporal objects are dates: fixed precision, absolute time references that are specified to *day level* precision. It is possible to name references, perform addition and subtraction on them in terms of days, and compare precedence. The model does not include a formal definition of how these dates can be constructed nor does it go into detail about the range of temporal semantics that can be expressed.

2.1.2 Normative Statements

Boulmakoul and Sallé [9] propose normative statements as a formalism for modelling contracts. Each normative statement defines some obligation between parties. A normative statement has the following form:

$$l : f \mapsto D_{i_1,i_2}(a < T)$$

Where l is a label, f is some predicate, D is some deontic relationship (either obligation, permission or prohibition), i_1 and i_2 are identities of parties, a is an action, and T is a temporal constraint. The statement reads as "when f holds, i_1 is obliged/permitted/prohibited (depending on D) by i_2 to achieve/perform a before T"

Here T is a date. In this model dates can be named, addition and subtraction can be performed on dates in terms of days, and an ordering is defined on dates.

2.1.3 Functional Programming

Peyton Jones and Eber [10] propose a formalism for modelling contracts that is implemented using the Haskell functional programming language. The core concept in this model is that, at the lowest level, contracts are either rights (entitlements) or obligations. They propose a compositional model where various combinators are used to combine basic contracts into more complex ones in order to express richer semantics.

A key aspect of this formalism is that rights or obligations may only be activated if some observable condition is satisfied, with temporal constraints forming a subset of these observable conditions. Absolute points in time themselves are represented using the simple *date* type. They can be created from simple English strings up to minute level precision. The range of temporal semantics that can be expressed using this model is quite broad: we can express choice, intervals and more using combinators on observables.

Temporal constraints could make use of predicates and functions defined on *dates*, enabling us to express even more complex temporal semantics. Additionally, although The authors do not explicitly mention this in the paper, it is presumably possible to creates of *dates* given that the model is presented in Haskell. The only hard constraint imposed by the model is that *dates* correspond to precise minutes in time. Another issue with this formalism is that arithmetic on dates in not defined.

2.1.4 Finite State Machines

Molina-Jimenez et al. [11] propose *executable contracts* where contracts are represented as finite state machines. Each state of the machine corresponds to some run time state of the contract where at each state, parties have a distinct set of obligations, permissions, and prohibitions. Contract related events serve as the inputs to the machine, while administrative contract actions serve as the outputs.

Absolute and relative temporal constraints are implemented using timers. Timers are activated on transitions between states (timer activations are one of the possible outputs of the machine). Timer timeout is a contract related event and as such is represented as a

machine input. This model uses dates for its basic absolute time references. These dates can be named, and addition and comparison can be performed implicitly using timers.

2.1.5 Business Contract Language

Milosevic et al. [12][13] propose the Business Contract Language (BCL). The language consists of roles and policies. Policies encapsulate deontic semantics, as well as associated relative temporal and state based ordering, and absolute temporal constraints.

In their model they acknowledge the temporal aspects of contracts, identifying the fact that they could be single, absolute points in time or durations, or even more complex expressions like sliding windows. They use a combination of basic time values as well as events to reason about time. In this model events may define an interval during which the event was "active". These intervals are continuous, and operations like intersections can be performed on them.

2.1.6 Process Algebra

Andersen et al. [14] propose an approach that is similar to that of Peyton Jones and Eber [10] in the sense that contracts are defined recursively. The base, atomic definition of a contract is either *success*, signifying that all obligations have been met, or *failure*, signifying that a party has defaulted.

A contract can also be a $transmit(a_1, a_2, r, t \mid p)$ which defines an obligation for party a_1 to transfer resource r to party a_2 under the constraints defined by the predicate p. t is a variable that represents the time at which the event of the obligation being discharged occurs. This variable is available to be used in the constraints p. t is a date. Dates can be named, addition and subtraction can be performed on dates, and dates are ordered.

2.1.7 Dynamic Logic

Prisacariu and Schneider [15] propose a contract language $\mathcal{CL}$. $\mathcal{CL}$ is made up of elements of deontic, dynamic and temporal logics. An extension to $\mathcal{CL}$ was proposed by Fenech et al. [16] Wtih this extension, they are able to specify obligations, permissions and prohibitions with state based constraints, and relative temporal constraints. The system also has support for reparations.

This formalism has support for expressing a relative ordering of obligations, however it seems there is no support for absolute temporal expressions.

2.1.8 Defeasible Logic

Defeasible deontic logic of violation as proposed by Governatori [17] and later refined by Governatori and Pham [18] extends defeasible logic [19] and adds deontic modalities. In its basic form, defeasible logic provides a method of reasoning about rules and precedence among those rules. Rules are similar to implications in that if we have

$$r_1 : \alpha \mapsto \beta$$

Then if α is known to be true, β is also known to be true. However, contradictory facts can defeat previously known facts. For example if we have

$$r_2 : \alpha \mapsto \neg\beta$$

And $r_2 > r_1$, then r_2 has precedence over r_1 and therefore we know that $\neg\beta$ must be true. The extension proposed by Governatori and Pham takes defeasible logic and introduces deontic predicates.

There is no explicit mention of absolute temporal semantics. However, in some examples, it seems that the model is able to implicitly name, perform addition and subtraction on, and compare time values.

2.1.9 Lee's Logic Programming Formulation and Recent Extensions

Lee [20] proposes a formalism for capturing contract semantics that brings together many different tools and frameworks into a single cohesive framework. This includes using operators from deontic logic to model obligations, permissions, and prohibitions.

In Lee's formalism, the *day* is the basic unit of time, and absolute time values are specified to day-level precision. Addition and subtraction can be performed on these values, again to a day-level precision. Lee uses an adaptation of the Rescher-Urquhart (RU) calculus to denote the occurrence of events or, more generally, the realisation of some condition with respect to time. The $R_t\phi$ operator tells us that the first order logic formula ϕ is satisfied at time t. Lee extends the model to incorporate *spans* of time values. A span is bounded by two absolute time values which may be special elements representing undefined days that are infinitely in the past or future. We can use $RD_d\phi$ and $RT_d\phi$ to test if the formula ϕ is realised during or throughout (respectively) the time span d.

Vanca [21] makes significant improvements to the temporal aspects of Lee's model. He

questions the limited precision of discrete time values in Lee's formalism and raises the valid point that contract authors may wish to use time values with finer granularity. In this model, absolute time references can be specified with (optionally) more precision. Time references are specified in the following format where each parameter except the year is optional:

$$Year - Month - Day \text{ at } Hour : Minute : Second.Millisecond$$

This model also addresses the fact that Lee's model has no support for expressing recurrences in time. It includes the SCH operator which allows us to specify recurrences using Cron-like syntax [22, 23]. This represents an incremental improvement over Lee's *spans* as the SCH operator allows us to express sets of non-contiguous time values with more complex membership rules. However, in Vanca's construction, the recurrences that are produced by SCH are not treated like sets. Instead, when they are named they are indistinguishable from single discrete time values and are used solely for expressing recurring obligations. Consequently we cannot perform unions or intersects on these recurrences. Finally, Vanca's SCH operator cannot express recurrences with sufficiently complex membership rules including rules that reference user defined properties. Values such as *the next succeeding scheduled settlement date* or *every first Monday of every month* cannot be expressed using SCH.

2.2 Temporal Logics

A temporal logic system is a tool that can be used to represent and reason about knowledge, state, or the occurrences of events with respect to time.

Temporal logic formulations can be categorised according to various characteristics. One fundamental separating factor in this space is what object the formulation uses as its fundamental unit of time. Some formulations use *points* while some use *intervals*. Generally speaking a point represents a single, indivisible point in time, while an interval represents some time period that could possibly be decomposed into smaller periods. Some interval-based formulations consider intervals to be sets of points, while others consider intervals to be primitive objects that cannot be deconstructed (although they can be split into smaller sub-intervals). However, interval-based formulations cannot avoid using points entirely since most of these formulations use points as the lower and upper bound on intervals, thus most interval-based formulations must include some formal definition of points.

Another key separating factor is whether or not the formulation allows the user to reason about the occurrence of events with respect to some global clock (absolute time) or just relative to the occurrence of other events (relative time). The former is clearly much more

useful for the use case discussed in this paper.

2.2.1 Point Based Formulations

A point is an atomic, zero-duration temporal quantity that represents an exact point in time. Points are typically used to describe instantaneous changes in state.

To formalise this, a point based temporal logic has a temporal structure consisting of a the pair $\langle \mathcal{T}, < \rangle$, where $\mathcal{T}$ is the set of points and $<$ is a strict ordering relation. The following rules hold if $<$ is a strict partial order:

$$\forall x \neg (x < x)$$
$$\forall x \forall y \forall z (x < y \wedge y < z \rightarrow x < z)$$

If $<$ is a strict *total* order, then the following also holds:

$$\forall x \forall y (x = y \oplus x < y \oplus y < x)$$

Where $\oplus$ represents *exclusive-or*.

A further distinction that must be made is whether the formulation considers points to be *discrete*, *dense* or *continuous*. These properties are stated below:

Discrete $\forall x \exists y (x < y \wedge \forall z (x < z \rightarrow y < z))$
Dense $\forall x \forall y (x < y \rightarrow \exists z (x < z < y))$

A *continuous* set is one that is *dense* and that represents an *unbroken continuum of elements*. This is best explained through an example: The set of real numbers is continuous because the elements in this set form the continuous, unbroken number line. Meanwhile, the set of rational numbers (a proper subset of the reals) is dense because, although there are an infinite amount of points between any two points in this set, it does not constitute an unbroken continuum since there are definitely gaps (the rational numbers do not include irrational real numbers such as π or $\sqrt{2}$.).

A final distinction must be made between *linear* and *branching* formulations. To summarise the difference, in a linear temporal structure we assume that time flows in a single uninterrupted line. In a branching structure we assume that time can branch into multiple paths to represent different possible futures (CTL [24] and CTL* [25] are examples of formulations that assume a branching structure).

Pneuli [26] proposes Linear Temporal Logic (LTL). This is a point-based, discrete, relative-time temporal logic formulation. It is an extension of propositional logic which includes some

additional, temporal operators to enable reasoning about time. Those operators are $\mathcal{U}$ (until) and $\circ$ (next). For some proposition p, $\circ p$ means that in the next time-step, p must be true. If we have another proposition q, $p\mathcal{U}q$ means that p must remain true until q becomes true. Additional operators including $\Diamond$ *eventually* and $\Box$ *always* can be constructed using *until* and *next*.

Metric Temporal Logic (MTL) [27] is an absolute time version of LTL. The temporal operators from LTL can be annotated with absolute time intervals. This example, provided by Konur [28], means that if p occurs, then q must eventually occur *within 0 to 10 time units of p occurring.* Clearly this is useful for expressing deadlines.

$$p \rightarrow \Diamond_{(0,10)} q$$

MTL works on discrete, dense, or continuous temporal structures.

Another absolute time formulation is Real Time Logic (RTL) [29]. This formulation is an extension of first order logic as opposed to propositional logic. A feature of this formulation is that it has an explicit global clock. The occurrence of some action A is divided into a pair of "events", $(\uparrow A, \downarrow A)$. $\uparrow A$ is the event describing when the action begins and $\downarrow A$ is the event describing when the action ends. The time at which the i^{th} occurrence of any event a happens is accessed using the $@(a, i)$ operator. This is called the occurrence function. We can use it to express properties about a system. For example, if we had two actions A and B and we wanted to express the rule that A must be fully completed before B we could have:

$$\forall i, j(@(\downarrow A, i) < @(\uparrow B, j))$$

This formulation also has support for performing integer addition on time values. For example, if we wanted to enforce a delay between actions A and B we could have:

$$\forall i, j(@(\uparrow B, j) > @(\downarrow A, i) + 5)$$

To enforce a 5 second delay between A and B. Clearly, the addition of an explicit global clock brings us closer to the functionality we desire.

Real Time Temporal Logic (RTTL) [30, 31] is similar to RTL in that it also extends first order logic and has an explicit global clock. It differs in the way that the global clock is queried: a variable t refers to the *current time* and can be used in formulas. Moreover, points can be stored in variables and compared to t. Konur [28] provides an example of a

formula in this logic.

$$\Box T[(red \wedge t = T) \rightarrow \Diamond(green \wedge T + 3 \leq t \leq T + 5)]$$

This can be interpreted as "if the traffic light is red at time T, then it will eventually be green with 3 to 5 ticks of the discrete global clock".

2.2.2 Interval-based Formulations

Interval-based formulations allow us to reason about spans of time as opposed to precise points. This is useful for certain types of expressions: Halpern and Shoham [32] provide this example: "I solved the problem while jogging to the ocean and back". Here, the event of "solving the problem" seemingly occurs over the interval of time during which the speaker was "jogging", and not at any a *single* point. From this example, we can extract some properties of intervals. First, we can see that intervals have some non-zero duration: the subject must have been jogging for some amount of time. Second, we can see that intervals are bounded by points since there must have been some distinct point at which the subject changed state from "not jogging" to "jogging" and eventually from "jogging" to "not jogging".

These intuitions shape the formal temporal structure that underlies interval-based formulations. An interval-based formulation is dependent on a point based structure $\mathbb{T} = \langle T, < \rangle$. This means that the same decisions that are made when defining a point based temporal logic must also be made when designing the point structure for an interval-based logic. This includes properties like the density of points and whether the structure is linear. We must also decide if the formalism is absolute time or relative. Points from the set T are used to define the bounds of intervals. The interval structure is defined as the pair $\langle \mathbb{T}, I(\mathbb{T}) \rangle$ where $I(\mathbb{T})$ is the set of all intervals in the structure. An interval is denoted as $[t_1, t_2]$ where $t_1, t_2 \in T$. A distinction must be made between the kinds of intervals that are acceptable in different formulations. In some formulations, $I(\mathbb{T}) = I(\mathbb{T})^-$ which is the set of *strict* intervals. These are intervals of the form $[t_1, t_2]$ where $t_1 < t_2$. The set of intervals where $t_1 = t_2$ are called *point* intervals and are included (along with strict intervals) in the set $I(\mathbb{T})^+$.

There are two views on what intervals are and how they should be represented [28]. One is that intervals are *primitive objects* of time - that is intervals are not decomposable into any other kind of object. The second view is that intervals are sets of points, where points are defined as they are in point based formulations.

Another issue, discussed by Allen [33], is whether the bounds of an interval are open or closed. He argues, if points are continuous, the lower bound must be closed while the upper bound should be open. He presents his argument using an example: imagine trying to create

two intervals, one interval describing the period of time during which a light is switched on ($[n_1, n_2]$) and the next interval describing the period of time during which the light is switched off ($[f_1, f_2]$). First, since the points are continuous, the only justifiable choice of n_2 and f_1 is $n_2 = f_1$. This is because continuous points have no direct successor, so choosing $n_2 < f_1$ would imply that there exists a non-empty interval $[n_2, f_1]$ during which the light is neither on nor off. Choosing $f_1 < n_2$ would be nonsensical as the light would be both on and off during the interval $[f_1, n_2]$. Therefore n_2 must equal f_1. If the first interval were inclusive of n_2 and the second interval inclusive of f_1, then the points f_1 and n_2 would be in both intervals, meaning that the light is both on and off at this point. Conversely if the first interval were exclusive of n_2 and the second interval exclusive of f_1, then the points f_1 and n_2 would be in neither interval, meaning the light is neither on nor off at this point. Since points have no duration, there is nothing explicitly wrong with either of these situations. However, intuitively, making the first interval exclusive of n_2 and the second interval inclusive of f_1 *seems more correct* since it mitigates both of those scenarios.

Allen has published several influential works on the subject of interval-based temporal logic. In [34] Allen proposes an interval-based temporal formulation and introduces some nine binary relations to express the various ways two intervals may be arranged relative to each other. He expands on this model in [33], formalising some of the notions from the previous work and adding more relations, bringing the total to thirteen. In [35] Allen goes into more detail about the relationship between intervals and points. He also identifies an important temporal object which he calls *moments*. Moments are used to describe the occurrence of *near instantaneous events*. Allen offers this example of such an event: "A light bulb burns out with a flash, plunging the room into darkness, and in that instant one sees a face at the window, caught in the intensity of the flash." Moments are like points in that they are atomic, however they are like intervals in that they have some non-zero duration. He describes a moment as "an interval with no internal structure. It therefore does not overlap any other interval, and contains no other intervals ... One can sum it up by saying that there are no points inside it".

Another influential interval-based temporal formulation is the logic HS proposed by Halpern and Shoham [32]. This model extends propositional logic and provides unary operators to test whether a proposition is true during special intervals that are related to the contextual "current interval" in various ways (i.e. after, before, during (starts-with and ends-with) etc.).

Real Time Interval Logic (RTIL) [36] is a absolute time interval-based temporal logic proposed by Razouk and Gorlick. In this model, intervals are considered to be sequences of states, where each change in state is called a transition point. Intervals are defined

by their initial state, the final state, and all of the interior states that are contained in the interval. The authors note that other absolute time formulations implement absolute time functionality by annotating events (or transition points) with time stamps. This is problematic because there is a mapping between time stamps and transition points, and it is not possible to refer to points in time that do not coincide with transition points. This motivates the solution presented in this paper which is to allow for "absolute time values" which are defined as point intervals whose beginning and end do not correspond to transition points. Instead, these intervals begin and end at that same pseudo-transition point that does not correspond to any actual change in state. Real time points contain no interior states, and their beginning and end state are the same. Intervals can be bounded by absolute time points in the same way they can be bounded by transition points.

Chapter 3

Evaluation of Prior Work

3.1 Temporal Requirements for Formalisms

Clack and Vanca [1] conduct an analysis of the 2002 ISDA Document set and identify a range of different expressions related to time that are found in these documents. They extract the set of abstract temporal objects that are implicitly constructed or otherwise manipulated in those expressions. These objects are named and defined. Additionally, for each object they identify and describe the set of behaviours that the object can exhibit, and the set of operations that can be performed on the object. In order satisfy the requirements, a formalism must be able to correctly model each of these abstract temporal objects, emulate its behaviours, and implement the required operations. The objects identified are:

- Discrete time values
- Sets of discrete time values
- Bags of discrete time values
- Continuous time intervals

A key problem that is identified in this work is that the deontic, temporal, and operational semantics in contracts are closely linked and difficult to separate. Clack and Vanca call this the *separability* problem. Consequently, they also describe how each of the objects listed above interact with the deontic and operational aspects of contracts and the additional requirements that arise out of these interactions.

3.1.1 Requirements

What follows is brief description of the objects and their associated requirements.

Discrete Time Values

A discrete time value is a reference to a date representing a single day. Discrete times can be found as annotations to obligations. Obligations have three associated temporal quantities: the start date (when the obligation is incurred), a due date, and a discharged date. The start date and the discharged date are represented using discrete time values, while the due date may be represented using a discrete time value or a set or bag of discrete time values to express choice (this will be discussed in more detail later on). Discrete time values may also be used to describe the time at which an event begins and the time at which it ends.

The requirements for discrete time values are that we must be able to:

- create a new single discrete time value;
- provide the special single discrete time value that is the current time;
- bind a name to a historical list of single discrete time values (and with each such value record when it was bound, who by, why, whether it was bound in the text or during performance, and perhaps some further properties);
- perhaps provide the extreme values $T_{-\infty}$ and T_{∞} and to test whether a single discrete time value is one of these two extreme values;
- increment a discrete time value by one day;
- decrement a discrete time value by one day;
- get the difference in days between two discrete time values (though this deserves more attention, since we may need to calculate the number of days with a specified property - e.g. Business Days);
- associate a set of properties with a discrete time value and test a discrete time value to see if it has a stated property
- apply a predicate to a discrete time value to see if it passes or fails a test;
- provide equality and relational operators to use on two dates;
- create and use date expressions combining any of the above operations.

Sets and Bags of Discrete Time Values

Clack and Vanca describe a a set of discrete time values as "a time-ordered set of discrete dates with a start date and an end date, without duplicates." A bag is an unordered collection of dates that can have duplicates. Sets and bags are needed to model deadlines that are expressed as a collection of acceptable dates. They may also be used to represent the temporal constraints associated with repeated obligations.

The requirements for sets and bags of discrete time values are that we must be able to:

- create a new set or bag of discrete time values;

- get the start date or end date of a set or bag;
- test whether a given date is a member of the set or bag;
- get the intersection of two sets or bags, returning a set or bag respectively (which might be empty);
- get the union of two sets or bags, returning a set or bag respectively;
- test whether two sets or two bags are equal;
- bind a name to a set or bag;
- apply a filter to a set or bag, to produce a set or bag (respectively) that may be smaller or equal in size;

Continuous Time Intervals

A continuous time interval is a range of time that is bounded by two discrete time values. These intervals are typically used to express the duration over which some permission or prohibition holds.

The requirements for such objects are as follows:

- create a new continuous time interval with discrete time values for the start and end points (if the end-points are defined to be outside the interval, this would give a straightforward representation of an "empty" interval as being one where the start and end points are the same, and it would not be an error to request the start or end point of an empty interval);
- bind a name to an interval;
- create an aggregate collection of intervals (this is one way to implement a union of non-overlapping intervals, since we cannot have an interval result containing gaps);
- get the start or end point of an interval;
- get the intersection of two intervals (perhaps returning an "empty" interval);
- test whether a discrete time value is before the start of such an interval; and
- test whether a discrete time value is after the end of such and interval.

3.1.2 Evaluating and Updating the Requirements

Sets and Bags

Clack and Vanca express the need for two distinct types of collections of discrete time values: sets and bags, however the motivation behind some requirements for these objects is not clear. First, there is no immediately obvious reason why sets must be ordered. Second, it is not clear whether the additional functionality offered by bags is necessary. No contract scenarios

are presented which demonstrate a situation where the semantics cannot be represented using a set and must be represented using a bag. Performing a thorough analysis and refinement of these requirements is beyond the scope of this project. In the model presented in chapter 4, an effort is made to remain faithful to the requirements for sets and bags as they are.

Arbitrary Precision Absolute Time References

Discrete time values are used to model references to points in absolute time, however they are limited to just modelling references that refer to days or dates. Therefore they are unable to model the following expressions that refer to other quantities of time of different precision:

> "The report must be submitted in March 2018"
> "The trade was made at 13:45:56 on 14 April 2020"

These phrases make reference to absolute points in time, but they are specified to either more or less precision than *days*. As a result, we cannot call such objects discrete time values. These objects are called **arbitrary precision absolute time references (APATRs)**. As the name suggests, they represent references to absolute points in time specified to any level of precision.

Although the absolute time references in the 2002 ISDA Master agreement are made to a precision level of days, one could imagine that as technology progresses and with the rise of automatic high frequency trading, contracts may need to include time references with greater precision than days. For this reason, the requirements are changed to replace discrete time values with APATRs.

APATRs are used in contracts to describe the measured time at which events occur and to express deadlines. What determines the precision with which a reference is made depends on how it is being used. When describing the measured time at which an event occurred, the precision of the reference is determined by the precision of the tool that is used to measure time. When a deadline is expressed with limited precision, this typically signifies indifference on the part of the beneficiary of the obligation with respect to the precise time at which an obligation can be discharged. Alternatively, deadlines may be expressed at some limited precision because it is known that it will not be possible to measure the time at which the discharge event occurs to greater precision.

The functional requirements for discrete time values must be adapted slightly to suit APATRs. It must be possible to:

- create a new single APATR;

- provide the special single APATR that is the current time - the precision of this reference is dependent on context;
- bind a name to a historical list of single APATRs (and with each such value record when it was bound, who by, why, whether it was bound in the text or during performance, and perhaps some further properties);
- perhaps provide the extreme values $T_{-\infty}$ and T_{∞} and to test whether a single APATR is one of these two extreme values;
- increment a APATR by any time quantity up to the precision that it has been specified to (years, months, days etc.);
- decrement a APATR by any time quantity up to the precision that it has been specified to (years, months, days etc.);
- get the difference between two APATRs. This will be in terms of standard time quantities. (though this deserves more attention, since we may need to calculate the number of days with a specified property - e.g. Business Days);
- associate a set of properties with a APATR and test a discrete time value to see if it has a stated property
- apply a predicate to a APATR to see if it passes or fails a test;
- provide equality and relational operators to use on two dates;
- create and use date expressions combining any of the above operations.

In addition, the requirements for sets and bags of discrete time values hold instead for sets and bags of APATRs.

3.2 Evaluation Criteria for Temporal Models

Faithfulness to Semantics

When evaluating a model, an obvious question that arises is: "Is this model capable of *faithfully* and *totally* representing *all* of the real objects that we want it to model?" An object's representation in a model is *faithful* to the object if the representation accurately encodes the object's meaning and the representation is capable of emulating all of the object's behaviours and functionality. *Totally* here means that the model should be capable of faithfully modelling all of the possible instances of a real object.

Functional Requirements

The functional requirements for each object were outlined earlier. The only changes are that the requirements for discrete time values have been inherited by APATRs and modified to

suit this new object.

3.3 Evaluation

3.3.1 Smart Contract Formalisms

APATRs

Clearly most of the formalisms that were presented earlier are unable to faithfully represent APATRs. This is because in most of the formalisms that have support for absolute time references, it is only possible to make fixed precision references (typically to a precision of days.) Only Vanca [21] identifies the need for variable precision. However this formalism technically still falls short since references are ultimately limited to millisecond precision. It could be argued that this level of precision is sufficient for current needs, but in the future more precision could be required.

In terms of the functional requirements, some of the formalisms do allow for naming absolute time references, testing equality and ordering, and performing addition and subtraction in terms of days. With Vanca's variable precision values, addition can be performed using years, months, days, hours, minutes, seconds, and milliseconds. This is achieved through the use of so called "differential time periods".

Sets and Bags of APATRs

Most formalisms do not allow for the construction of sets or bags of their respective absolute time reference objects.

Lee's [20] formalism allows us to construct spans of contiguous dates. It could be argued that this could be considered either a set of time references or a continuous time interval. If we consider it to be a set of time references, then it falls short of fulfilling the requirements for sets since we can only construct a subset of all of the possible sets of time values between the bounds (since we can only construct the sets consisting of contiguous values).

Vanca [21] attempts to improve on this with the SCH operator. With this operator, we can construct sets using Cron-like recurrence syntax. While this allows us to express a larger range of sets, we still are not able to construct *every* possible set of time values. For example, we could not construct the set consisting of *all the business days in March 2019* because we have no way of accessing arbitrary properties (like whether or not a day is a business day) using this syntax.

The formalism proposed by Peyton Jones and Eber [10] is interesting. While it does not

include support for sets or bags of discrete time values in the original formulation, since the formulation is presented in Haskell it could easily be extended to include such objects. Bags could potentially be emulated using `lists` of dates while sets could also be `lists` of dates with the additional constraint that they are ordered and have no duplicates.

Continuous Time Intervals

Only a handful of formalisms have support for objects that resemble continuous time intervals.

As mentioned, Lee's formalism allows us to construct spans. If we interpret these as continuous intervals then, for the most part, they come close to faithfully represent our desired semantics. However, they are clearly not *continuous* since they are always countable/discrete sets of dates. Additionally they are limited by the fact that their bounds are specified using dates, which are not precise enough for our needs. Vanca [21] contends that Lee's spans can represent both discrete and continuous time intervals. He asserts that SPANs are treated as discrete intervals when the RD_Φ is used, and as continuous intervals when RT_Φ is used. His rationale behind this is that when checking if some formula ϕ is realised throughout a span, we must check that $\neg\phi$ does not hold at every single infinitesimally small point in time in the span. However it could be argued that, based on Lee's presentation of his temporal model, *there are no infinitesimally small points in a span* since dates are the indivisible unit of time in this model. To check if ϕ is realised throughout a span, we simply check if ϕ is realised at each of the dates in the span.

Lee's spans do meet some of the functional requirements of continuous time intervals. We can get the dates that represent the beginning and end of a span, and since dates are ordered we can test whether a date is before the beginning of the span or after the end. In Lee's implementation (the logic programming formulation), spans can be named. However, we are unable to take the union or intersection of two spans, nor can we create aggregate collections of intervals.

Milosevic et al. [12] propose some complex temporal objects in their model, including intervals. However, they do not provide a formal description of how such objects can be constructed or used in their model, nor do they describe the functionality of such objects.

3.3.2 Temporal Logics

APATRs

It is possible to make broad judgements about the ability of temporal logics to model APATRs based on their defining characteristics (i.e. whether they are point based or interval-

based, absolute time or relative, discrete or continuous etc.) Clearly relative time formulations in any form are not useful, because for this use case it is critical to be able to reason about the occurrence of events with respect to a global (absolute) clock.

Point based formalisms are not useful for representing APATRs. This can be argued as follows: Consider a *discrete*, linear, absolute time, point based formulation. Any point in that structure represents some fixed precision absolute time reference, but it is not possible to vary the precision. Similarly, in a *continuous*, linear, absolute time, point based formulation each point represents an absolute time reference specified to exact (or infinite) precision. Points in such a formulation must be infinitely precise, since if they had fixed precision then each point would have a unique successor and the points would be discrete. Thus in continuous point based formalisms it is not possible to model APATRs.

Interval-based formulations could be used to model APATRs. Intervals of points and APATRs are fundamentally similar since they both represent time quantities that have non-zero duration and are *decomposable*. Conceptually, APATRs can be *decomposed* into smaller constituent objects. For example: an hour can be decomposed into a set of minutes which can each be decomposed into a set of seconds and this can continue all the way down to some infinitely precise time quantity (which we call points in formal models). Similarly, an interval of points (in a continuous model) can be decomposed into smaller sub-intervals (each of which can be decomposed in the same way). Thus there are significant similarities between APATRs and continuous intervals of points. This relationship reconciles with the two ways in which APATRs arise: uncertainty in measurement and indifference to precision. In the case of approximate measurements, an APATR refers to the set of points at which some event *could have happened*. APATRs which arise out of precision indifference also refer to sets of points. For example, if the deadline for an obligation is specified in a contract as *November 12, 2021*, this means that the obligation can be satisfied at any time within the continuous range of points that comprise that day.

The absolute time interval-based formulations that have been presented come close to satisfying the requirements, but they are clearly lacking in some areas. It must be possible to label ATAPRs with human readable date times and allow them to be constructed using such syntax. Additionally, it must be possible to perform arbitrary precision addition and subtraction on such values. The formulations presented in the previous chapter do not explicitly have such functionality.

Sets and Bags of APATRs

The interval-based formulations presented earlier do not have explicit support for creating sets or bags of intervals. It could be inferred that it is possible to create sets of intervals

(and therefore sets of APATRs) since sets are relatively simple and common. However, bags are slightly more complex and would require special attention.

Continuous Time Intervals

Clearly it is only possible to express continuous time intervals in continuous interval-based formulations, so only these formulations are capable of faithfully modelling the desired semantic for intervals. However, a conflict arises between the requirements and these models. In the requirements, Clack and Vanca state that intervals should be bounded by discrete time values. In Chapter 3 these objects were replaced with APATRs, so it seems as though the requirements should now state that intervals must be bounded by APATRs. This presents a problem because in interval-based formulations, *APATRs are intervals*, creating a circular dependency. Thus, we must flex the requirements slightly to allow intervals to be *bounded by points*.

Chapter 4

New Temporal Model

The temporal models that are used in state of the art smart contract formalisms do not fully meet the requirements. They resemble discrete point based temporal logics, since they only allow the specification of fixed precision time quantities. This is problematic, since this prevents them from being able to represent arbitrary precision absolute time references. Moreover, these systems have limited support for representing intervals of continuous time. In addition, we are unable to associate properties with single time values and perform arithmetic on those properties. Finally, we are unable to construct sets or bags of time values.

Real time interval-based temporal formulations with continuous point structures seemed to be the most promising in terms of satisfying Clack and Vanca's requirements. The model proposed in this section extends such models and includes some new features. This includes the ability to construct points using human-friendly date time syntax, syntax for representing APATRs, enabling the use of predicates to query properties on APATRs, arbitrary precision addition and subtraction on APATRs, and sets and bags (implemented using multisets) of APATRs.

4.1 Point Structure

This model is an absolute time interval-based temporal formulation. Thus it is dependent on an underlying point structure $\langle \mathcal{T}, < \rangle$. Points in this structure are linear and continuous, and points correspond to points in absolute time. The decision to make points continuous was briefly mentioned earlier. One justification for this choice is that, while humans might instinctively think of time as a discrete procession of *ticks*, it is easy to see that each of those ticks can always be further decomposed into a series of smaller ticks for any choice of tick. For example, seconds can be decomposed into milliseconds which can in turn be decomposed into microseconds. This process can continue *ad infinitum*, thus we inevitably

reach the conclusion that time must consist of a *dense* set of points. This model assumes time to be *continuous* because, given that it is dense, there is no reason to believe that time is not a continuum since there is nothing to indicate that there is any *empty space* between two points in time. A consequence of this is that a point describes an *exact, infinitely precise* point in time (this is reinforced through the connection between points and the real numbers described in 4.1.2).

There is a strict total order on points. Put simply if we have two points x and y, then $x < y$ if and only if x precedes y in time. The ordering is irreflexive because it is strict:

$$\forall x \neg (x < x)$$

In addition, the ordering is transitive:

$$\forall x \forall y \forall z (x < y \wedge y < z \rightarrow x < z)$$

Finally, since this is a strict, total ordering, the following holds:

$$\forall x \forall y (x = y \oplus x < y \oplus y < x)$$

Points are bounded in the past but unbounded in the future. The fixed lower bound corresponds to the point of the beginning of the universe. Since this value is not known it is denoted using $\perp$ with the property that $\forall x \in \mathcal{T}(x \neq \perp \rightarrow \perp < x)$. There is no fixed upper bound on this set, since time is (presumably) infinite. Therefore we include positive infinity in this set with the property that $\forall x \in \mathcal{T}(x \neq \infty \rightarrow x < \infty)$.

4.1.1 Additional Features

Construction and Annotation with Human Friendly Syntax

In some absolute time formulations, absolute time references are created by assigning integer time stamps to events such that an event that precedes another in time will have a smaller time stamp. This is not sufficient for smart contracts. It is necessary to be able to reason about the occurrence of events with respect to a globally recognised timing system using familiar syntax that has meaning across domains. It should be possible to describe points using a standardised calendar and time format.

To address this, points are always identified using standard, human time quantities: year, month, day of month, hour, minute, second, and fractional seconds. For ease of use, this model uses an adapted form of ISO-8601 [37, 38] notation to refer to points because

it is an internationally recognised standard for referring to absolute points in time. This notation is referred to as "date-time notation" in this model. In date-time notation, points are specified first with the date using the Gregorian calendar system, and then their time using the standard 24 hour clock notation:

$$YYYY : MM : DD : hh : mm : ss.f \cdots$$

All values are integers except for f which is a real number $0 \leq f < 1$. f can have any number of digits and for convenience a bar is used to denote recurrence in this number. Below are some examples of some phrases that refer to points, and their equivalent counterparts expressed using date-time notation. In these phrases, the key word **exactly** tells us that they refer to infinitely precise points as opposed to fixed precision quantities.

"Exactly 11:30 on 21 January 1973"
$1973 : 01 : 21 : 11 : 30 : 00.0$

"Exactly 3 millionths of second after 15:42:37 on 27 April 2014"
$2014 : 04 : 27 : 15 : 42 : 37.000003$

"Exactly one third of a second after 22:30 on 30 March 2009"
$2009 : 03 : 30 : 22 : 30 : 00.\overline{3}$

Points have been defined as being *infinitely precise*, thus it may seem contradictory that their representation uses a variable number of digits. This is done purely for convenience. One can imagine that that any date-time representation of a point (except those with recurrences) is actually followed by an infinite amount of zeros. For example:

$$1973 : 01 : 21 : 11 : 30 : 00.0 = 1973 : 01 : 21 : 11 : 30 : 00.000000000\ldots$$

For notational convenience, the trailing zeros are stripped and the point is represented using the minimal amount of digits it needs in order to ensure accuracy. An important point to note is that, although points relate to points in absolute time, they *are not* APATRs. Points have no duration and are specified to infinite precision, therefore they cannot be APATRs.

Date time syntax must be constrained to ensure validity. Using unconstrained date-time notation, it is possible to express invalid points. For example $2019 : 13 : 12 : 11 : 30 : 00.0$ or $2019 : 03 : 34 : 14 : 20 : 53.001$ would be invalid points since they refer to dates that do not exist. Even $2019 : 02 : 29 : 09 : 24 : 00.0$ would be invalid because 2019 is not a leap year. Clearly, values must be constrained to ranges to ensure that points are valid. Some of these

constraints are simple: months must be between 1 and 12, hours between 0 and 23, minutes and seconds between 0 and 59. The maximum value of the element representing the day is dependent on both the month and the year. The value of the element representing the year is unconstrained. Positive values represent years in the *Common Era (Anno Domini)*", while years 0 and below represent years *Before the Common Era (Before Christ)*". (Year $0 = 1$ BCE, Year $-1 = 2$ BCE...). This is consistent with the *ISO-8601* system for representing dates and times.

Arbitrary Precision Addition and Subtraction

A common operation that could be performed on APATRs is the addition of arbitrary time quantities to the reference. For example we might be interested in knowing which point is *2 days after* $2019:07:27:12:32:00.0$ (the answer would be $2019:07:29:12:32:00.0$). Computing the answer entails *incrementing* the *days* field of the point by 2. We represent such operations using addition between points and **time deltas**. Time deltas represent some incremental quantity that we can add on to a point. Time deltas are represented using the same syntax as points, but with a δ subscript to distinguish them from points. The addition given above can be written as

$$2019:07:27:12:32:00.0 + 00:00:02:00:00:00.0_\delta = 2019:07:29:12:32:00.0$$

When addition is performed between a point and a time delta, what actually happens is that we increment each value in the point by the corresponding value in the time delta. We also perform carrying to ensure that each value in the point is valid for its unit, and to ensure that the point as a whole is valid.

Performing addition using time deltas is useful in order to dynamically specify intervals of a fixed length. Intervals in this model have not been discussed yet, however they are bounded by points in the form $[p_1, p_2]$. Since points can be named we could define an interval as $[x, x + 00:00:02:00:00:00.0_\delta]$ to create an interval whose lower bound is x and whose upper bound is the point two days following x. This enables the bounds of the interval to be dynamically evaluated.

The set of points is a continuum, with each point representing some exact point along an infinite "time line". Given this information, a fair question that one might ask is: *is addition between two points defined?* The answer is that this operation is defined, and an explanation for this is given in 4.1.2. A question that now arises is: *what does this operation mean?* It

seems that the answer is that this operation does not have any meaning. It is impossible to think of a situation where one might want to add together two points in time.

4.1.2 The Connection Between Points and the Real Numbers

In prior work, an implicit mapping exists between fixed precision discrete time values and the *integers*. For example, in Lee's model where time is represented using discrete days, we can choose one day to be the "zeroth" value and represent all other values as the difference in days between that value and the zeroth value. Since all values are represented as days, this difference is an integer.

In this model, every point implicitly has a meaningful counterpart in the *real numbers*. Intuitively, a point's real number counterpart corresponds to its displacement from *some* origin, expressed in *some* time unit.

Each point actually has infinitely many counterparts in the real numbers since there are infinitely many choices of origins and units. Therefore, there exists an injective function $C_{O,U} : \mathcal{T} \to \mathbb{R}$ and a surjective function $C^{-1}_{O,U} : \mathbb{R} \to \mathcal{T}$ for each valid choice of origin O and unit U.

As an example, we could choose $O^* = 1970 : 01 : 01 : 00 : 00 : 00.0$ as our origin and use $U^* = \textit{milliseconds}$ for our displacement unit. As a result:

$$C_{O^*,U^*}(1970 : 01 : 01 : 00 : 00 : 00.0) = 0$$
$$C_{O^*,U^*}(1970 : 01 : 01 : 12 : 00 : 00.0) = 43,200,000$$
$$C_{O^*,U^*}(1969 : 12 : 31 : 12 : 00 : 00.0) = -43,200,000$$

(There are $43,200,000$ milliseconds in twelve hours.)

Not every real number has a counterpart in the points: real numbers that correspond to points in time before the beginning of the Universe have no corresponding point. As a result, these functions are not bijections.

A useful result from this relationship is that it serves to reify some of the operations that are defined on points. We can express equality and ordering relations in terms of the points' real number counterpart given some choice of O and U:

$$x = y \Leftrightarrow C_{O,U}(x) = C_{O,U}(y) \quad x, y \in \mathcal{T}$$
$$x \leq y \Leftrightarrow C_{O,U}(x) \leq C_{O,U}(y) \quad x, y \in \mathcal{T}$$
$$x < y \Leftrightarrow C_{O,U}(x) < C_{O,U}(y) \quad x, y \in \mathcal{T}$$

An important point to note is that there is no dependency between points and their real number counterparts. Points and the operations that can be applied to points exist and are defined independently of the mapping between points and real numbers. The motivation for describing the mapping was to make it clear what points are and to give an alternative explanation for the operations that are defined on points. That being said, in many practical implementations it is not uncommon for time values to be *represented* in the manner described above (or similar). For example, in the UNIX time system, time values are represented using (an integer approximation of) their displacement in seconds from midnight (UTC) on the first of January 1970. Many date and time libraries for popular programming languages use this representation to store time values since it is independent of time zones and it makes it easy to perform addition and subtraction.

Although addition between points is not a very useful operation, we can still give meaning to it:

$$x + y = C_{O,U}^{-1}(C_{O,U}(x) + C_{O,U}(y)) \quad x, y \in \mathcal{T}$$

Time deltas do not have a real number counterpart, since two equivalent time deltas may actually express different quantities of time depending on the point they are added to. This can be expressed mathematically:

$$\exists x, y, d(C_{O,U}(x + d) - C_{O,U}(x) \neq C_{O,U}(y + d) - C_{O,U}(y))$$

Where $x, y \in \mathcal{T}$ and d is a valid time delta. This is made more clear with a specific example:

$$\begin{aligned} &2019:07:04:00:00:00.0 + 00:01:00:00:00:00.0_\delta \\ &\qquad = 2019:08:04:00:00:00.0 \\ &2019:02:04:00:00:00.0 + 00:01:00:00:00:00.0_\delta \\ &\qquad = 2019:03:04:00:00:00.0 \end{aligned}$$

However, clearly:

$$\begin{aligned} &C_{O,U}(2019:03:04:00:00:00.0) - C_{O,U}(2019:02:04:00:00:00.0) < \\ &\qquad C_{O,U}(2019:08:04:00:00:00.0) - C_{O,U}(2019:07:04:00:00:00.0) \end{aligned}$$

Because there are fewer intervening days between $2019:02:04:00:00:00.0$ and $2019:03:04:00:00:00.0$ than there are between $2019:07:04:00:00:00.0$ and $2019:08:04:00:00:00.0$. Clearly we cannot define a function that maps time deltas to a single real number counterpart.

4.2 Intervals

This model has an interval structure that consists of the pair $\langle \mathbb{T}, I(\mathbb{T}) \rangle$, where $\mathbb{T} = \langle \mathcal{T}, < \rangle$ is the point structure defined in the previous section. $I(\mathbb{T})$ is the set of all intervals in this model. Intervals are *strict* meaning that they cannot be point intervals. Intervals are considered to be sets of points as this provides greater meaning to some of the operations and objects that are defined later. As a result, intervals of points can be thought of as uncountably infinite sets because points are continuous. Intervals are constructed using the following syntax.

$$\langle p_1; p2 \rangle$$

For example,

$$\langle 2018:02:18:00:00:00.0; 2019:02:25:00:00:00.0 \rangle$$

Would yield the interval of points that fall within the week beginning Monday 18 February 2019. The greatest point in this set is the point $2019:02:24:23:59:59.\overline{9}$.

An important point to note is that intervals are always *inclusive* of their lower bound and *exclusive* of their upper bound. This choice can be justified by using the same logic that Allen uses in [33]. Admittedly, since points are continuous, this choice constrains the range of intervals that can be expressed. Continuous points have no single direct successor (or predecessor), so it is impossible to claim that intervals with inclusive lower bounds and exclusive upper bounds are capable of representing all other intervals with any other combination of exclusivity on their bounds. However, in the context of the use case discussed in this dissertation, it can be argued that this does not present a problem. The users of this system are not interested in points, as points have no analogue in their domain. Instead they are always interested in limited precision time quantities that do have meaning in their domain (such as hours, minutes, seconds etc.). Thus, if a user wanted to exclude some time values from the lower end of an interval or to include some time values at the upper end of an interval they almost certainly would not be interested in excluding or including any exact points. Instead they would be interested in excluding or including fixed precision quantities. This can be easily done by adding the requisite time deltas to the bounds of an interval.

The thirteen relations that were identified and developed by Allen [33] are incorporated into this model. However, for the needs expressed in the requirements, the ones that prove most useful are:

Equality ($=$)	$(\langle a_1; a_2\rangle = \langle b_1; b_2\rangle) \Leftrightarrow (b_1 = a_1 \wedge a_2 = b_2)$
During ($\subseteq$)	$(\langle a_1; a_2\rangle \subseteq \langle b_1; b_2\rangle) \Leftrightarrow (b_1 \leq a_1 \wedge a_2 \leq b_2)$
Contains ($\supseteq$)	$(\langle a_1; a_2\rangle \supseteq \langle b_1; b_2\rangle) \Leftrightarrow (a_1 \leq b_1 \wedge b_2 \leq a_2)$
Overlaps (o)	$(\langle a_1; a_2\rangle o \langle b_1; b_2\rangle) \Leftrightarrow (a_1 < b_1 \wedge b_1 < a_2 \wedge a_2 < b_2)$
Overlapped by (ob)	$(\langle a_1; a_2\rangle ob \langle b_1; b_2\rangle) \Leftrightarrow (b_1 < a_1 \wedge b_2 < a_2)$
Before ($<$)	$(\langle a_1; a_2\rangle < \langle b_1; b_2\rangle) \Leftrightarrow (a_2 < b_1)$
After ($>$)	$(\langle a_1; a_2\rangle > \langle b_1; b_2\rangle) \Leftrightarrow (b_2 < a_1)$

Two relations are introduced which correspond to special kinds of overlaps:

Maybe Before ($<?$)	$(\langle a_1; a_2\rangle <? \langle b_1; b_2\rangle) \Leftrightarrow (a_1 < b_1 \wedge a_2 \leq b_2)$
Maybe After ($>?$)	$(\langle a_1; a_2\rangle >? \langle b_1; b_2\rangle) \Leftrightarrow (b_1 < a_1 \wedge b_2 \leq a_2)$

This model allows for intervals to be created by taking the *intersection* ($\cap$) of two intervals. Note that this may produce an empty interval. Thus $\emptyset$ is included in the set of possible intervals.

4.3 Periods

Periods are this model's implementation of arbitrary precision time references. A period is an interval of points with the unique property that their inclusive lower bound is the very beginning of some human time quantity (e.g. an hour, a day, a month etc.) with respect to a global calendar and time system, and their exclusive upper bound is beginning of the very next time quantity of the same unit. For an example of this, Lee [20] uses discrete time references that are specified to a day level precision (for instance, we might have $27 - jun - 1987$). The corresponding period would be the interval:

$$\langle 1987:06:27:00:00:00.0; 1987:06:28:00:00:00.0\rangle$$

Using interval syntax to specify periods can be quite verbose. This model uses finite lists of integers to specify periods, where each successive integer represents a time unit of increasing precision. These lists can be of any lengths. The first seven integers in a list are interpreted as the year, month, day, hour, minute, and second of the period. The eighth integer represents the milliseconds, and each successive integer following that represents one thousandth of the previous unit. So the interval can be rewritten as $[1987, 6, 27]$, with the understanding that:

$$[1987, 6, 27] = \langle 1987:06:27:00:00:00.0; 1987:06:28:00:00:00.0\rangle$$

We denote the set of all periods as $\mathbb{P} \subset I(\mathbb{T})$. An interesting result is that the set of all periods that have been specified to the same fixed precision (i.e. days) is a *discrete set*. That is, each period in this set has a well defined successor which is the period that describes the next day (or, more generally, the next period of the same precision). To formalise this, we can define the sets of all periods that have been specified to the same level of precision as follows: Let $\mathbb{P}_i \subset \mathbb{P}$ be the set of periods that are specified using i integers. For example the set $\mathbb{P}_3$ is the set of all periods that represent days.

$$[1987, 6, 27] \in \mathbb{P}_3$$
$$[2019, 3, 15] \in \mathbb{P}_3$$
$$[2020, 6] \notin \mathbb{P}_3$$
$$[1876, 6, 12, 14, 45] \notin \mathbb{P}_3$$

These sets will be useful later when constructing multisets of periods.

The semantics of periods depend on the context of their use. For example, periods may arise from **uncertainty in measurement**. Human time-measuring instruments have fixed precision and consequently when we say that an event has occurred at *11:56:32:67 on October 18, 2019* (according to our instruments) what we actually mean is that the event has occurred *at exactly one of the points within the period from* $2019:10:18:11:56:32.67$ *up to but not including* $2019:10:18:11:56:32.671$. This is the period $[2019, 10, 18, 11, 56, 32, 670]$ However, due to the limited precision of our instruments, we cannot be sure which point it actually occurred at. Another source of periods is **precision indifference**. This arises often in contracts. For example, a contract may state that *Party B must pay a sum of £400 to Party A on July 8, 2020*. In this case *Party B* will successfully fulfil this obligation as long as their payment is completed at *exactly one point in the period from* $[2020, 7, 8]$ *up to but not including* $[2020, 7, 9]$. This is because *Party A* is actually indifferent to the exact point at which the payment is made, as long as it is made in the specified period.

4.3.1 The Semantics of Interval Relations

Since periods are just special instances of intervals, the same relations that hold between pairs of intervals also hold between pairs of periods and pairs of periods and intervals. While the actual mechanics of these relations do not change, their meaning changes depending on what combination of objects they are applied to.

Equality

Two periods are equal if and only if they refer to the exact same abitrary precision absolute time reference. More generally, two intervals are equal if they refer to the same continuous interval of time.

Before and After

When applied on two periods, the *before* ($<$) and *after* ($>$) relations can be used to test if a period precedes another in time. When applied between a period and any general interval, these relations tell us whether an period occurs before or after a continuous interval. For smart contracts this relation is useful for checking if party has committed an action during the continuous time interval associated with a prohibition on that action.

During and Contains

If one period is *during* another this means that all of the points in the first period are also in the second period. At a high level, we interpret this to mean the first period *occurs during* the second. This relation is crucial for contracting as it would be used to test whether an obligation has been satisfied within an acceptable period. For example, If we say the time at which the *the obligation is discharged* is represented using the period $[2018, 4, 13, 11, 30]$, and the obligation has a deadline given by $[2018, 4, 13]$ then the obligation has been successfully satisfied because $[2018, 4, 13, 11, 30] \subseteq [2018, 4, 13]$.

This relation is also used to test if a period falls within an interval. This is necessary to test if a party has committed a prohibited or permitted action. The time at which the action is committed is described using an period, while the duration over which a permission or prohibition holds is described using an interval. Thus if the period is wholly contained in the interval then it can be said for certain that the action occurred at a time at which the prohibition or permission held.

Maybe Before, Maybe After

The *maybe before* $<?$ and *maybe after* $>?$ relations have a relatively specific use case. They are not very useful when used on two periods (because periods cannot overlap one another), but when used between a period and an interval they allow the user to test if a period *might* fall within an interval. This could be useful to check if a party *might have* committed an action during an interval in which the action is prohibited. Such scenarios can only arise if there is a mismatch between the granularity of the bounds of the interval and the precision

of the period. For example, we might have a situation where an action is prohibited during the interval.

$$\langle 2019:07:03:12:00:00.0; 2019:07:10:12:00:00.0\rangle$$

If an action occurs at the measured time of $[2019, 7, 3]$ then this period is *maybe before* the interval. That is the action might have occurred before the interval, or it may have occurred during the interval. The user must decide how they want to interpret such a scenario.

4.3.2 Functionality

Since periods are just intervals, they retain all of the functionality of intervals. However, this also provides some interesting new semantics to some of the existing operations as will be shown.

Addition and Substraction using Period Deltas

Addition and subtraction between periods work in a similar manner to time deltas with points. Period deltas are specified using the same syntax as periods, but are distinguished from periods using the subscript δ. A period delta can be at most as precise as the period it is being added to. Adding a period delta to a period has the effect of incrementing each value of the period by the corresponding value in the delta. Here *1 month* is added to the period $[2019, 4, 23, 11, 30]$

$$[2019, 4, 23, 11, 30] + [0, 1]_\delta = [2019, 5, 23, 11, 30]$$

This construct allows for arbitrary precision addition with APATRs.

Difference Between Two Periods

The difference between two periods is the period delta that describes the amount of time in between the two of them. This operation is expressed as:

$$[2019, 4, 23, 11, 30] - [2019, 4, 23, 11, 25] = [0, 0, 0, 0, 5]_\delta$$

The operation is defined in such a way that:

$$p_2 + (p_1 - p_2) = p_1$$

Where $p_1, p_2 \in \mathbb{P}$.

This operation is only supported between two periods of the same precision. The reason for this is that it would become unclear how to define the difference in time between two periods of unequal precision. For example, the difference in time between [2010] and $[2011, 3, 14]$ is ambiguous. It is not clear if the difference should be taken from the beginning of [2019] or the end. Further uncertainties arise when considering the difference between a period and another that contains it. For example, how do we define the difference between $[2019, 8]$ and $[2019, 8, 13, 11, 30]$?

Note that this interpretation of period difference is not necessarily the only interpretation. Another idea might be to take the *time delta* between the upper bound of the earlier period and the lower bound of the later period. Note that this would produce different results in many cases. The interpretation given in this model was selected in order to maintain consistency with period addition and subtraction.

Predicates and Functions

Predicates allow us to query properties on periods, while functions allow us to extract information from periods in order to construct more useful predicates. A useful pre-defined function is:

$$getElem :: \mathbb{P} \times \mathbb{Z} \to \mathbb{Z}$$

This function gets the n^{th} integer element from a period if the period has been specified to at least n integers of precsion and is undefined otherwise. For example:

$$getElem([2019, 4, 23, 11, 30], 3) = 23$$
$$getElem([2019, 4], 3) = \text{undefined}$$

Here is an example of a predicate that can be defined using this function:

$$inFeb :: \mathbb{P} \to \{True, False\}$$
$$inFeb(p) := getElem(p, 2) = 2$$

4.4 Multisets of Intervals

The requirements state the need for sets and bags of arbitrary precision absolute time references. In this model, sets and bags are implemented using multisets [39]. A multiset is a collection of elements where each element can appear more than once. The number of times an element appears in a multiset is called its *multiplicity*. Multisets can clearly be used to

represent bags. A key feature of multisets is that they can also be used to represent sets, since a set is just a multiset where all elements have a multiplicity of 1.

In this model, multisets are ordered according the the ordering relation on intervals. This is done for convenience, as it allows sets and bags to be represented by the same object. Clearly this means that, when representing bags, multisets violate the requirement that bags should be unordered. However, for the sake of simplicity this is allowed.

This model allows for the creation of *multisets of intervals*. Since periods are a type of interval, it is possible to construct sets and bags that contain just periods. Therefore it is possible to model sets and bags of ATAPRs and it also possible to create "aggregate collections of intervals" as stated in the requirements.

4.4.1 Interpreting and Constructing Sets and Bags

Multisets can be constructed using an adaptation of set builder syntax. In general we can create a multiset as follows:

$$(\{x \in U | P(x)\}, M(x))$$

Where U is the universe from which elements are drawn from, $P : U \to \{True, False\}$ is a predicate the determines multiset membership and $M : U \to \mathbb{Z}^+$ is the function that determines the multiplicity of each member of the multiset.

In this model, U can be $I(\mathbb{T})$ (the set of all intervals) or any of its subsets including $\mathbb{P}$ (the set of all periods), or one of the sets $\mathbb{P}_i \subset \mathbb{P}$ that were discussed earlier. For convenience, the set $\mathbb{P}_{i+}$ refers to the set of all periods with precision of *at least* i.

For convenience, the **Iverson bracket** [40] is often used to create sets. For a predicate $P : U \to \{True, False\}$:

$$[P(x)] = \begin{cases} 1 & P(x) \\ 0 & \neg P(x) \end{cases}$$

This enables this creation of multisets where every element that satisfies the membership rule has a multiplicity of 1.

Here is the definition of a set which contains of *the eighth day of each month in 2019*:

$$(\{x \in \mathbb{P}_3 \mid P(x) = (getElem(x, 3) = 8) \wedge (getElem(x, 1) = 2019)\},$$
$$M(x) = [(getElem(x, 3) = 8) \wedge (getElem(x, 1) = 2019)])$$

As mentioned in Chapter 3, it is unclear why multiplicity is necessary or what it might represent. One possible interpretation is that if we have a collection with multiple member-

ship rules (i.e. a disjunction of rules), then elements that satisfy one or more of the rules should have their multiplicity reflect the number of rules they satisfy. For example, the following is a bag whose membership rule contains a disjunction: *All days in 2019 that are designated payment days or are in January, February or March.* The corresponding multiset definition would be as follows:

$$(\{x \in P_3 | (getElem(x,1) = 2019 \wedge (1 <= getElem(x,2) <= 3 \vee isPaymentDay(x)))\},$$
$$M(x) = \begin{cases} 2 & (1 <= getElem(x,2) <= 3 \wedge isPaymentDay(x)) \\ 1 & otherwise \end{cases})$$

It is clear that by using this syntax, sets and bags can be created with arbitrarily complex membership rules. One might observe that this syntax is similar to list comprehension syntax in functional programming languages like Haskell [41] or Miranda [42].

4.4.2 Operations

Membership testing ($\in$)

$$x \in S \iff M_S(x) > 0$$

Subset ($\subseteq$) **and Strict Subset** ($\subset$)

$$A \subseteq B \iff \forall x \in A, M_A(x) \leq M_B(x)$$
$$A \subset B \iff \forall x \in A, M_A(x) < M_B(x)$$

Union ($\cup$) **and Intersect** ($\cap$)

$$A \cup B = (\{x \in I(\mathbb{T}) \mid P(x) = (x \in A \vee x \in B)\}, M(x) = Max(M_A(x), M_B(x)))$$
$$A \cap B = (\{x \in I(\mathbb{T}) \mid P(x) = (x \in A \wedge x \in B)\}, M(x) = Min(M_A(x), M_B(x)))$$

Observe that these definitions of union and intersect preserve the semantics of the equivalent set operations when they are used on a pair of multisets that represent sets.

Concatenation $(++)$

Concatenation is an operation that is unique to multisets and is defined as follows:

$$A++B = (\{x \in I(\mathbb{T}) \mid P(x) = (x \in A \vee x \in B)\}, M(x) = M_A(x) + M_B(x))$$

4.4.3 Iteration Using Quantifiers

It is possible to iterate over the members of a multiset using the $\exists$ and $\forall$ quantifiers. This is necessary to check if any or all of the members of a multiset satisfy some property, or if they satisfy a binary relation with another interval. For example, we could test if the period p is *during* any of the intervals in the multiset S. This can be expressed as:

$$\exists x \in S, p \subseteq x$$

Alternatively, we may be interested in testing if all of the periods in a multiset of periods are business days. If we assume there exists a predicate $isBusinessDay(x)$, this assertion can be expressed as:

$$\forall x \in S, isBusinessDay(x)$$

4.4.4 Functions on Multisets

Some useful functions can be defined on multisets. The first of these is *filter*. This function takes a multiset A and a predicate $F(\cdot)$ and returns a new multiset that contains only the elements of A that satisfy F.

$$filter(A, F) = (\{x \in A \mid P(x) = F(x)\}, M(x) = M_A(x))$$

There are a suite of functions that are commonly used in functional programming which operate on `lists` that can be applied to multisets. These functions include:

```
take :: [a] -> Integer -> [a]
take ls n returns the first n elements of the list ls

drop :: [a] -> Integer -> [a]
drop ls n returns the list ls with the first n elements removed

head :: [a] -> a
head ls returns the first element in ls
```

```
tail :: [a] -> [a]
tail ls returns the list ls with the first element removed
```

4.4.5 Alternative Solutions

As mentioned, the requirements for sets and bags of APATRs are not clear. The need for two different kinds of collections is not obvious. It is possible that sets alone are sufficient, which would eliminate the need for multisets.

It could also be argued that multisets are unnecessarily complex. They could be replaced with *lists*, which could potentially be easier to interpret and work with.

4.5 Reasoning About Events

So far, the temporal objects of the model have been presented. What remains is an explanation of the mechanisms that are used to reason about the occurrence of events using those objects. This model takes inspiration from the temporal aspects of Lee's formalism [20]. It extends Lee's adaptation of the RU Calculus and re-interprets it to work with the temporal objects of this model: points, intervals, periods, multisets of periods, and collections of intervals.

4.5.1 Lee's Adaptation of the RU Calculus

As mentioned briefly in Chapter 2, Lee extends the work of Rescher and Urquhart [43]. The fundamental operator in this work is the 'realised' operator.

$$R_t\phi$$

This reads as the assertion that "ϕ is realised at time t" where ϕ is a first order logic formula and t is a point in time. In Lee's model, ϕ is typically a formula relating to that state of the contract, for example ϕ might represent the fact that an obligation has been discharged or an action has been performed. The $R_t\phi$ operator allows the user to query whether those facts have been realised at a given time point.

Lee includes the $RD_d\phi$ (realised during) and $RT_d\phi$ (realised throughout) operators to reason about the realisation of facts with respect to *spans*, which are the equivalent of intervals in his formalism. A fact is *realised during* a span if it is realised at one or more of

the points within the span. A fact is *realised throughout* a span if it is realised at no less than all of the points within the span.

Finally, Lee also includes RB_t (*realised before*) operator to assert whether a fact is realised before a given point in time.

4.5.2 Further Extension of the RU Calculus

Reinterpreting "Realised" with Respect to Periods

A fundamental principle of this model is that the time at which an event occurs or a fact is realised must inevitably be described using an arbitrary precision time reference. This is because the tools that humans use to measure time have limited precision, and are not capable of measuring the *point* at which something occurs. By extension, the time at which an event occurs or a fact is realised is described using a period in this model. The precision to which the period is specified depends on the instrument used to measure that event. This reconciles with the way that people interact with time in everyday life. A person may glance at their watch to confirm that their train is indeed leaving as scheduled, at 11:42. If this occurred on February 13, 1973, the time at which the train leaves the station could be described using the period $[1973, 2, 13, 11, 42]$. Meanwhile, an engineer at a high frequency trading firm may check the logs to find that a trade was carried out at `2019-07-03 19:37:43:456`. The time at which the trade occurred could be described using the period $[2019, 7, 3, 19, 37, 43, 456]$.

Consequently, in this model, the $R_t\phi$ operator becomes $R_p\phi$ and is re-interpreted to act on periods. Initially, this may be read as "ϕ is realised at period p." A question that arises immediately is: "what does *at* mean here?" In this model, *at* means *during*. That is to say, the operator reads as "ϕ is realised during period p." Since periods are used to represent the times at which facts are realised, a more verbose reading of this operator is "The period describing the time at which ϕ is realised *is during* ($\subseteq$) the period p."

The utility of this operator is immediately obvious. In contracts, periods can be used to specify simple deadlines, so this operator can be used to check whether some action has been performed *during* the deadline period. For example, take the following obligation and associated deadline:

> "The payment must be made on Sunday April 14, 2019"

The expression $R_p\phi$ can be used to check if this deadline is met where $p = [2019, 4, 14]$ and $\phi =$ payment-made.

Realised Before and Realised After

Lee's $RB_t\phi$ operator is re-interpreted to work with periods but maintains similar semantics. $RB_p\phi$ reads "ϕ is realised before or during the period p." ϕ is allowed to be realised during p to maintain consistency with Lee's operator. However, there could be cases where a stricter assertion is required (i.e. one that means "ϕ is realised strictly before the period p.") For this there is the *strict* realised-before operator: $SRB_p\phi$. Which has the stated meaning. The model has similar operators, realised-after ($RA_p\phi$) and strict realised-after ($SRA_p\phi$), to implement the same semantics but using the "after" relation. An example where the RB operator would be used is as follows:

> "The payment must be made no later than Sunday April 14, 2019"

The expression $RB_{p'}\phi'$ is used to check if this deadline is met where $p' = [2019, 3, 14]$ and $\phi' =$ payment-made. Alternatively we could have:

> "The payment must be made at any time up to but not including Sunday April 14, 2019"

In which case the expression $SRB_{p'}\phi'$ is used to check if this deadline is met where $p' = [2019, 3, 14]$ and $\phi' =$ payment-made.

The underlying interval relations in the realised-before and realised-after relations are maybe-before and maybe-after respectively. Meanwhile, strict realised-before and strict realised-after use the before and after relations.

Containment

For completeness, the *realised-contains* operator $RC_p\phi$ is included. A verbose reading of this operator is as follows "The period describing the time at which ϕ is realised *contains* ($\supseteq$) the period p." The utility of this operator is limited.

The only scenarios where this operator might be satisfied are scenarios in which the period that describes the time at which ϕ is realised is specified to a lower precision than p. The precision of the period that describes the occurrence of an event is determined by the precision of the clock that measures it. Smart contract designers that use this formalism must therefore be wary that the deadlines that they choose for certain actions are specified to a coarse enough precision such that they are not contained by the periods which describe the associated discharge events.

Periods and Intervals

Although periods and intervals in this model have the same underlying structure, they are intended to model distinct abstract objects. Periods are supposed to represent APATRs while intervals (constructed using the normal interval syntax) are used to represent continuous time intervals. For this reason, another set of operators are introduced to reason about the occurrence of events with respect to intervals. In many cases, these operators perform the exact same underlying operations as the operators presented above. They are kept separate for the sake of user friendliness and readability.

- The *Realised* operator for intervals is $R_i\phi$. This operator asserts that "The period describing the time at which ϕ is realised *is during* ($\subseteq$) the interval i."
- The *Realised-before* operator for intervals is $RB_i\phi$. This operator asserts that "The period describing the time at which ϕ is realised *is maybe before* ($<?$) the interval i."
- The *Realised-after* operator for intervals is $RA_i\phi$. This operator asserts that "The period describing the time at which ϕ is realised *is maybe after* ($>?$) the interval i."
- The *Strict Realised-before* operator for intervals is $SRB_i\phi$. This operator asserts that "The period describing the time at which ϕ is realised *is before* ($<$) the interval i."
- The *Strict Realised-after* operator for intervals is $SRA_i\phi$. This operator asserts that "The period describing the time at which ϕ is realised *is after* ($>$) the interval i."
- The *Realised-contains* operator for intervals is $RC_i\phi$. This operator asserts that "The period describing the time at which ϕ is realised *contains* ($\supseteq$) the interval i."

Iterative Operators for Multisets of Intervals

The requirements make it clear that it must be necessary to reason about the occurrence of events with respect to sets and bags of APATRs. With respect to contracts specifically, this might be because deadlines are given as choices of alternate times. Additionally, sets of APATRs may be useful for modelling recurring deadlines. For example, the following deadline can be modelled using a bag of alternate APATRs:

> "The payment must be made within 10 business days of the delivery date"

Let d be the period representing the delivery date. The bag can be represented using the multiset:

$$take(\{p \in \mathbb{P}_3 \mid [businessDay(p) \wedge p > d]\}, 10)$$

Of course, now what is needed is an operator to iteratively test whether the "payment-made" fact is realised during one or more of the periods in this multiset. This is the motivation for

the *weak iterative realised* operator, $WIR_m\phi$. This operator is equivalent to the following expression

$$\exists x \in M, p \subseteq x$$

Where p is the period that describes the time at which the fact is realised, and M is the multiset. This operator has a *strong* counterpart, $SIR_m\phi$ which is interpreted as

$$\forall x \in M, p \subseteq x$$

All of the period-to-interval operators presented earlier have weak and strong iterative counterparts in order to make those operators work with multisets of intervals.

4.6 Implementation

Inevitably, any smart contract formalism must be implemented using some computer programming language so that the formalism can be used to model and automate real smart contracts. Thus, creating a prototype implementation of a formalism is a key step in proving the viability of the formalism. Here, an outline is given of an implementation of this model in Haskell. Haskell was chosen as it allows for rapid prototyping of objects, and the syntax for expressing such objects is easy to interpret.

```
data tPoint = Point (Int, Int, Int, Int, Int, Int, Double)
                | Bot | Inf
```

The point is a tuple of six `Integer`s (to represent the year, month, day, hour, minute, and second) and one `Double` to represent the fractional seconds of the point. Alternatively a point can be `Bot` or `Inf` which represent $\perp$ and ∞ respectively. Observe that the choice of `Double` for the fractional seconds is the best attempt that can be made at making points continuous. Obviously, due to the nature of computer systems, this number will ultimately be fixed precision and thus these points are not truly infinitely precise. A compromise was made for the sake of ease of implementation and readability. Further work could be done to find a more faithful implementation of points.

The choice of

```
data tInterval = Interval (tPoint, tPoint)
type Period = tInterval
```

An interval is a pair of `Point`s, with the first being the inclusive lower bound and the second being the exclusive upper bound. A period is simply a type synonym on `Interval`. This

reflects the fact that periods are just a special kind of interval, but it enhances readability as it allows for more specific function typing. Binary relations and operations between intervals can be implemented as functions.

```
type tMSet = [tInterval]
```

Finally, the multiset of intervals is defined as a list of `Interval`s. This reflects the idea that multisets have similar functionality to lists.

Chapter 5

Evaluation

In this section, the model presented above is evaluated against the requirements identified by Clack and Vanca.

5.1 Arbitrary Precision Absolute Time References

In this model, arbitrary precision absolute time references are modelled using periods. Periods are are special intervals of *points* that correspond to absolute time quantities according to a widely recognised calendar and time system.

- It is possible to construct APATRs
- It is possible to perform addition and subtraction on these values at any suitable level of precision.
- It is possible to get the difference between two APATRs of the same precision. The result is a period delta.
- It is possible to associate and query properties with APATRs using predicates.
- It is possible to test two APATRs for equality and test their relative ordering (as well as numerous other relations).
- It is possible to create APATRs using addition and subtraction
- It is possible to get a single APATR representing the current date and time. The precision of this reference depends on the run-time system.

5.2 Sets and Bags of APATRs

In this model, multisets are used to model sets and bags of APATRs. Sets are special instances of multisets that satisfy the property that no element has multiplicity greater than

one and the elements are sorted. Multisets and bags are equivalent except for the fact that multisets in this model are ordered.

- It is possible to create a new set or bag of APATRs using a multiset
- It is possible to get the APATRs representing the beginning and end of a set that has been implemented using a multiset.
- It is possible to test multiset membership. In fact, quantifiers can be used to check if any or all of the members of a multiset satisfy some interval relation with another APATR.
- It is possible to get the union and intersection of two multisets. This produces another multiset.
- It is possible to test multiset equality.
- It is possible to bind a name to a multiset.
- It is possible to apply a filter to a multiset using either the *filter* function or the multiset builder syntax.

5.3 Continuous Time Intervals

Continuous time intervals are constructed fairly intuitively in this model, since we only deal with continuous time values. Intervals are just sets of contiguous points. We can achieve all of the desired semantics without having to deal with the two separate contexts of discrete and continuous time.

- It is possible to create a continuous time interval with two distinct points.
- It is possible to bind a name to an interval
- It is possible to create an aggregate collection of intervals using a multiset.
- It is possible to get the start and end point of interval.
- It is possible to take the intersection of two intervals. Set intersection works fine for this purpose.
- A range of interval relations are provided which allow the user to test whether an interval contains an APATR or whether an APATR occurs before or after the interval. Further relations are provided to allow the user to deal with the unique scenarios that arise due to the variable precsion of APATRs.

5.4 Deontic and Operational Requirements

Clack and Vanca note that temporal expressions are typically not found in isolation in contracts, but instead as annotations to the deontic and operational aspects of the contract. Those deontic and operational aspects include events, obligations, permissions, and prohibitions. In this model, a first step towards modelling the relationship between time objects and events and deontic objects was made through the reinterpretation of the RU Calculus. The additional operators that were introduced could be easily integrated back into Lee's model along with the temporal objects that were presented.

Events have an associated start and end times. In our model, the precise times at which an event begins and ends are points. However, in reality, it is impossible to measure the exact point at which these things occur. Instead, an event's start and end times must be represented as **intervals** to reflect the imprecision of the instruments used to measure those points.

Obligations have three associated points: the *start time*, *due time*, and *discharge time*. The start time is defined in the contract and can be specified as an exact point in time to reflect the fact that there is one distinct point at which the obligation becomes active. The due date is also defined in the contract, and would typically be specified to some arbitrary precision (days, hours, minutes etc.) Thus, the due time could be specified as an interval, since it is likely that if the due date is specified to some precision (e.g. day level precision), then the beneficiary of the obligation is indifferent to which specific point the obligation is satisfied, as long as it occurs on the specified day. Another possibility is that the due date time may actually comprise a set of alternative times at which the obligation can be satisfied. In this case, we can use a set of points to express the due time. Finally, the discharge time must be specified as an interval for the same reasons that event timings are expressed as intervals: time measuring instruments are imprecise. We can test if a party has not defaulted on an obligation by checking that the discharge time is a subset of the due time.

Permissions and prohibitions are typically active over continuous time intervals. These can be easily represented in our model. Since the start and end times of events are expressed as intervals, we can test to see if parties have committed prohibited actions by testing to see if the either of those intervals is a subset of the interval that defines the prohibition. We can do a similar operation for permissions.

Chapter 6

Conclusion and Future Work

6.1 Summary of Achievements

In Chapter 1, the goals for this project were listed. Now those goals are revisited, and the project is assessed based on how well each goal was met.

The primary objective was to motivate, present, and evaluate a new temporal formalism that can be used to model the temporal aspects of financial derivative contracts. The following steps were taken to motivate the new temporal formalism: First, the problem of contract formalisation was explained and put into context. Then, a specific focus was drawn to the problem of formalising the temporal aspects of smart contracts. Then, the temporal aspects of various formalisms were described. In addition, point based and interval based temporal logics were introduced. Then the requirements given Clack and Vanca were evaluated and updated, and then used to evaluate the various works that were presented. In doing so, deficiencies in these models were highlighted, helping to motivate the design decisions made in the new model.

Next, in Chapter 4, the new model was presented. The unique features of this model were explained and motivated with respect to the problems that were identified in the evaluation of the state of the art. It was shown how the temporal objects in this model could be integrated into an adapted version of the Rescher Urquhart calculus with the ultimate aim of enabling the expression of combined deontic and temporal semantics. The foundations of an implementation in Haskell were laid out as a proof of concept to show that the model has the potential to be implemented in practice. Finally the model was evaluated with respect to the set of requirements that was developed in Chapter 3.

6.2 Future Work

The most constructive work that could be done is to integrate this temporal formalism into a wider smart contract formalism that is already capable of modelling the requisite deontic and operational semantics. This work would consist of incorporating suitable constructors for the temporal objects into the formalism and developing operators to model all of the possible relationships and associations that can exist between the temporal objects and the various deontic "objects". For example, there would need to be a way to associate the period representing an obligation's deadline with the obligation itself. This combined formalism would then need to be fully evaluated.

In addition, an implementation of this formalism should be developed in some programming language. There are many questions that would arise during the development of such a system. One immediate difficulty would be deciding on an efficient representation for points (as discussed). In addition, the implementer would need to decide at what level of precision the "current time" should be made available to the system. Also, care must be taken to accurately model the quirks of the UTC system when modelling points and periods. Finally, addition and subtraction using points and time deltas and periods and period deltas must be made efficient without sacrificing correctness.

A theme of this project was that many of design decisions that were made were a *best-effort* attempt to satisfy the requirements *as interpreted*. There are numerous other equally justifiable and better solutions to this problems that were addressed by this project. The design decisions that were made in this work should be analysed and challenged and alternative solutions should be pursued and compared to the results presented in this dissertation.

Bibliography

[1] C. D. Clack and G. Vanca, "Temporal aspects of smart contracts for financial derivatives," *CoRR*, vol. abs/1805.11677, 2018.

[2] V. Patel, "Contract lifecycle management and the cfo: Optimizing revenues and capturing savings," *Aberdeen-Group, Boston, NY*, Apr. 2007. https://images.treasuryandrisk.com/treasuryandrisk/historical/SiteCollectionDocuments/aberdeen-contract-lifecycle.pdf/.

[3] ISDA, "The future of derivatives processing and market infrastructure," *ISDA Whitepaper*, Sept. 2016. https://www.isda.org/a/UEKDE/infrastructure-white-paper.pdf.

[4] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," *arXiv preprint arXiv:1608.00771*, 2016.

[5] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: essential requirements and design options," *arXiv preprint arXiv:1612.04496*, 2016.

[6] C. D. Clack, "Smart contract templates: Legal semantics and code validation," *Journal of Digital Banking*, vol. 2, no. 4, pp. 338–352, 2018.

[7] T. Hvitved, *Contract Formalisation and Modular Implementation of Domain-Specific Languages*. PhD thesis, The Faculty of Science, University of Copenhagen, 2011.

[8] A. Goodchild, C. Herring, and Z. Milosevic, "Business contracts for b2b.," *ISDO*, vol. 30, 2000.

[9] A. Boulmakoul and M. Sallé, "Integrated contract management," in *Proceedings of the 9th Workshop of the HP OpenView University Association*, 2002.

[10] S. P. Jones and J.-M. Eber, "How to write a financial contract," *Citeseer*, 2003.

[11] C. Molina-Jimenez, S. Shrivastava, E. Solaiman, and J. Warne, "Run-time monitoring and enforcement of electronic contracts," *Electronic Commerce Research and Applications*, vol. 3, no. 2, pp. 108–125, 2004.

[12] P. F. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal, "A unified behavioural model and a contract language for extended enterprise," *Data & Knowledge Engineering*, vol. 51, no. 1, pp. 5–29, 2004.

[13] Z. Milosevic, S. Gibson, P. F. Linington, J. Cole, and S. Kulkarni, "On design and implementation of a contract monitoring facility," in *Electronic Contracting, 2004. Proceedings. First IEEE International Workshop on*, pp. 62–70, IEEE, 2004.

[14] J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen, "Compositional specification of commercial contracts," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 6, pp. 485–516, 2006.

[15] C. Prisacariu and G. Schneider, "A formal language for electronic contracts," in *FMOODS*, vol. 7, pp. 174–189, Springer, 2007.

[16] S. Fenech, G. J. Pace, and G. Schneider, "Automatic conflict detection on contracts," in *International Colloquium on Theoretical Aspects of Computing*, pp. 200–214, Springer, 2009.

[17] G. Governatori, "Representing business contracts in ruleml," *International Journal of Cooperative Information Systems*, vol. 14, no. 02n03, pp. 181–216.

[18] G. Governatori and D. H. Pham, "Dr-contract: An architecture for e-contracts in defeasible logic," *International Journal of Business Process Integration and Management*, vol. 4, no. 3, pp. 187–199, 2009.

[19] D. Nute, "Defeasible logic," in *Handbook of logic in artificial intelligence and logic programming (vol. 3)*, pp. 353–395, Oxford University Press, Inc., 1994.

[20] R. M. Lee, "A logic model for electronic contracting," *Decision support systems*, vol. 4, pp. 27–44, Mar. 1988.

[21] G. Vanca, "The semantics of smart contracts used in banking and financial services," Master's thesis, Departament of Computer Science, UCL, Apr. 2018.

[22] A. C. Magazine, "Crontab quick reference." www.adminschoice.com/crontab-quick-reference.

[23] WDT.io Inc., "Crontab." https://crontab.guru/.

[24] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*, pp. 52–71, Springer, 1981.

[25] E. A. Emerson and J. Y. Halpern, ""sometimes" and "not never" revisited: on branching versus linear time temporal logic," *Journal of the ACM (JACM)*, vol. 33, no. 1, pp. 151–178, 1986.

[26] A. Pneuli, "The temporal logic of programs," in *FOCS 1977*, 1977.

[27] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-time systems*, vol. 2, no. 4, pp. 255–299, 1990.

[28] S. Konur, "A survey on temporal logics for specifying and verifying real-time systems," *Frontiers of Computer Science*, vol. 7, no. 3, pp. 370–403, 2013.

[29] F. Jahanian and A. K.-L. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Transactions on software engineering*, no. 9, pp. 890–904, 1986.

[30] J. S. Ostroff and W. M. Wonham, "Modelling, specifying, and verifying real-time embedded computer systems.," in *RTSS*, pp. 124–132, 1987.

[31] J. S. Ostroff, *Temporal logic for real-time systems*, vol. 40. Research Studies Press Advanced Software Development Series, 1989.

[32] J. Y. Halpern and Y. Shoham, "A propositional modal logic of time intervals," *Journal of the ACM*, vol. 38, no. 4, pp. 935–962, 1991.

[33] J. F. Allen, "Maintaining knowledge about temporal intervals," in *Readings in qualitative reasoning about physical systems*, pp. 361–372, Elsevier, 1990.

[34] J. F. Allen, "An interval-based representation of temporal knowledge.,"

[35] J. F. Allen and P. J. Hayes, "Moments and points in an interval-based temporal logic," *Computational Intelligence*, vol. 5, no. 3, pp. 225–238, 1989.

[36] R. Razouk and M. Gorlick, "Real-time interval logic for reasoning about executions of real-time programs," in *ACM SIGSOFT Software Engineering Notes*, vol. 14, pp. 10–19, ACM, 1989.

[37] "Date and time — representations for information interchange — part 1: Basic rules," standard, International Organization for Standardization, Geneva, CH, Feb. 2019.

[38] "Date and time — representations for information interchange — part 2: Extensions," standard, International Organization for Standardization, Geneva, CH, Feb. 2019.

[39] Wikipedia, "Multiset - Wikipedia, the free encyclopedia," 2019. https://en.wikipedia.org/wiki/Multiset.

[40] K. E. Iverson, "A programming language," 1962.

[41] "Haskell language." https://www.haskell.org/.

[42] C. Clack, C. Myers, and E. Poon, *Programming with Miranda*. Prentice Hall, 1995.

[43] N. Rescher and A. Urquhart, *Temporal logic*, vol. 3. Springer Science & Business Media, 2012.