TITLE OF INVENTION: DIGITAL ASSET MODELING

CROSS-REFERENCE

[0001] This application claims priority under 35 USC § 119 to U.S. Provisional Patent Application No. 62/329,888 (Atty. Dkt. 16-30013-US-PV, Client Dkt. DAH-2016011P) entitled Digital Asset Modelling Language Simulator, filed on April 29, 2016 in the United States Patent and Trademark Office (USPTO), the contents of which are herein incorporated by reference in their entirety.

TECHNICAL FIELD

[0002] The present disclosure relates to a digital asset modeling system and method for modeling, tracking and settling digital assets, obligations, and transactions.

RELATED ART

[0003] Existing closed, centrally administered ledgers utilized for settling assets, obligations, and transactions are considered opaque and error-prone. This makes oversight cumbersome, requires many duplicative processes and ledgers, and allows the potential for fraud. The first and currently largest alternative to the existing ledger architectures is represented by a distributed digital ledger called Bitcoin, which uses a blockchain data structure. A fundamental principle of Bitcoin's operation is that the system is set up as a peer-to-peer transaction mechanism that utilizes public-private key cryptography, has no central intermediary or central repository, and allows all participants in the network to hold and validate the integrity of a full copy of the ledger in real time. The Bitcoin blockchain was designed in order to create a trustless native asset, bitcoin, which could be exchanged with pseudonymous parties across the globe.

[0004] Current platforms built to support digital assets on top of Bitcoin-like or blockchain-like systems are not generally structured to provide comprehensive protection to financial institutions as may be required by law for many of their existing transaction businesses. These platforms may not have contemplated the regulatory regime for financial institutions and financial transactions in general. As a result, institutional investors have hesitated to enter the digital assets market and have avoided the use of distributed ledgers for their existing businesses.

[0004a] Throughout this specification the word "comprise", or variations such as "comprises" or "comprising", will be understood to imply the inclusion of a stated element, integer or step, or group of elements, integers or steps, but not the exclusion of any other element, integer or step, or group of elements, integers or steps.

[0004b] Throughout this specification the word "comprise", or variations such as "comprises" or "comprising", will be understood to imply the inclusion of a stated element, integer or step, or group of elements, integers or steps, but not the exclusion of any other element, integer or step, or group of elements, integers or steps.

SUMMARY

[0005] The embodiments disclosed herein provide mechanisms for adding flexibility to computer executed transactions of digital assets. The embodiments provide new data models and functions to allow computer systems executing the transactions to operate in a new and advantageous manner. Provided is an exemplary embodiment method of modeling a digital asset and its evolution with respect to the rights of a plurality of parties, the method comprising: providing an await function instance that executes no more than once using one of at least one choice defined therein for disposition of the digital asset with respect to the rights of at least one of the plurality of parties, said await function instance incorporated upon the consent of the affected parties to

fulfil a configured function instance associated with the at least one choice; providing an agree function instance that requires the consent of at least one of the plurality of parties to execute; and providing an append-only ledger for storing results of the executed function instances.

[0006] Provided is an exemplary embodiment method of modeling and/or recording a digital asset and its evolution, including the associated rights of a plurality of parties, the method comprising: determining an await function that includes at least one choice defined therein for disposition of the digital asset, wherein the at least on choice has an associated configured function and a respective choice-step condition. The method further comprises executing the await function instance no more than once, wherein upon receiving an indication of consent of parties required by the at least one choice and receiving a valid choice selection from the at least one choice, the method further comprises executing the associated configured function instance of the at least one choice, wherein the await function terminates with determination that the choice-step condition of the valid choice selection is satisfied; determining an agree function to create an agree record associated with the digital asset, wherein the method further comprises executing an agree function instance based on receiving an indication of consent of parties required for the agreement. The method further comprises sending, to an append-only ledger for storage, results of the executed function instances.

[0007] The methods of modelling and/or recording a digital asset and its evolution described above may be provided wherein the at least one of the plurality of parties whose respective rights are at stake is the same at least one of the plurality of parties whose consent is required. The method may be provided further comprising providing a delete function that requires the consent of the affected parties to invalidate an agree function or disable a non-executed await function, wherein the append-only ledger stores the results of the executed await, agree, and delete functions. The

method may be provided wherein the digital asset comprises at least one of cash and/or cash-valued payment, a fungible, equity, bond, commodity, future, right, or good. The method may be provided wherein the at least one choice of the await function is made by a delegate of the at least one of the plurality of parties. The method may be provided wherein the at least one choice of the await function is made by respective delegates of at least two of the plurality of parties. The method may be provided wherein the append-only ledger comprises a blockchain. The method may be provided wherein the append-only ledger may be queried for digital asset status based on pattern-matching. The method may be provided wherein the append-only ledger may be queried for digital asset status of all models in the ledger using queries based on top-level definitions. The method may be provided further comprising providing a delete function to render an active model inactive and no longer available for future transactions.

[0008] Provided is an exemplary embodiment method of interpreting a modeled digital asset and its evolution with respect to the rights of a plurality of parties, the method comprising: executing an await function instance no more than once using one of at least one choice defined therein for disposition of the digital asset with respect to the rights of at least one of the plurality of parties, said await function instance incorporated upon the consent of the affected parties to fulfil a configured function instance associated with the at least one choice; executing an agree function instance that requires the consent of at least one of the plurality of parties to execute; and storing the results of the executed function instances in an append-only ledger.

[0009] Provided is an exemplary embodiment method of interpreting a modeled digital asset and its evolution, including the associated rights of a plurality of parties, the method comprising: executing an await function instance no more than once, wherein the await function instance includes at least one choice defined therein for disposition of the digital asset, and the at least one

choice has an associated configured function and a respective choice-step condition, wherein upon receiving an indication of consent of parties required by the at least one choice and receiving a valid choice selection from the at least one choice, the method further comprises executing the associated configured function instance of the at least one choice, wherein the await function terminates with determination that the choice-step condition of the valid choice selection is satisfied. The method further comprises executing an agree function instance to create an agree record associated with the digital asset upon receiving an indication of consent of parties required for the agreement. The method further comprises sending the results of the executed function instances in an append-only ledger.

[0010] The methods of interpreting a modeled digital asset and its evolution described above may be provided wherein the at least one of the plurality of parties whose respective rights are at stake is the same at least one of the plurality of parties whose consent is required. The method may be provided further comprising executing a delete function that requires the consent of the affected parties to invalidate an agree function or disable a non-executed await function, and storing the results of the executed await, agree, and delete functions in the append-only ledger. The method may be wherein the digital asset comprises at least one of cash and/or cash-valued payment, a fungible, equity, bond, commodity, future, right, or good. The method may be provided wherein the at least one choice of the await function is made by a delegate of the at least one of the plurality of parties. The method may be provided wherein the at least one choice of the await function is made by respective delegates of at least two of the plurality of parties. The method may be provided wherein the append-only ledger comprises a blockchain. The method may be provided wherein the append-only ledger may be queried for digital asset status based on pattern-matching. The method may be provided wherein the append-only ledger may be queried for digital asset

status of all models in the ledger using queries based on top-level definitions. The method may be provided further comprising executing a delete function to render an active model inactive and no longer available for future transactions.

[0011] Provided is an exemplary embodiment digital system configured to interpret a modeled digital asset and its evolution with respect to the rights of a plurality of parties, the system comprising: at least one processor configured to execute an await function instance no more than once using one of at least one choice defined therein for disposition of the digital asset with respect to the rights of at least one of the plurality of parties, said await function instance incorporated upon the consent of the affected parties to fulfil a configured function instance associated with the at least one choice, and configured to execute an agree function instance within the at least one choice that requires the consent of at least one of the plurality of parties; and at least one storage device configured to store an interpreted result of the executed function instances in an append-only ledger.

[0012] Provided is an exemplary embodiment digital system configured to interpret a modeled digital asset and its evolution, including the associated rights of a plurality of parties, the system comprising: at least one processor configured to execute an await function instance no more than once, wherein the await function instance includes at least one choice defined therein for disposition of the digital asset, and the at least one choice has an associated configured function and a respective choice-step condition, wherein upon receiving an indication of consent of parties required by the at least one choice and receiving a valid choice selection from the at least one choice, the method further comprises executing the associated configured function instance of the at least one choice, wherein the await function terminates with the determination that the choice-step condition of the valid choice selection is satisfied. The at least one processor is further

configured to execute an agree function instance to create an agree record associated with the digital asset upon receiving an indication of consent of parties required for the agreement. The system further comprises at least one storage device configured to store an interpreted result of the executed function instances in an append-only ledger

[0013] The systems described above may be provided wherein the at least one of the plurality of parties whose respective rights are at stake is the same at least one of the plurality of parties whose consent is required. The system may be provided with the processor further configured to execute a delete function that requires the consent of the affected parties to invalidate an agree function or disable a non-executed await function, and to store the execution results of the await, agree, and delete functions in the append-only ledger. The system may be provided wherein the digital asset comprises at least one of cash and/or cash-valued payment, a fungible, equity, bond, commodity, future, right, or good. The system may be provided wherein the at least one choice of the await function is made by a delegate of the at least one of the plurality of parties. The system may be provided wherein the at least one choice of the await function is made by respective delegates of at least two of the plurality of parties. The system may be provided wherein the append-only ledger comprises a blockchain. The system may be provided wherein the append-only ledger may be queried for digital asset status based on pattern-matching. The system may be provided wherein the append-only ledger may be queried for digital asset status of all models in the ledger using queries based on top-level definitions. The system may be provided with the processor further configured to execute a delete function to render an active model inactive and no longer available for future transactions.

[0014] The embodiments disclosed herein may provide mechanisms for adding flexibility to computer executed transactions of digital assets. The embodiments may provide new data models

and functions to allow computer systems executing the transactions to operate in a new and advantageous manner.

BRIEF DESCRIPTION OF THE DRAWINGS

[0015] Illustrative, non-limiting exemplary embodiments may be more clearly understood from the following detailed description, particularly when taken in conjunction with the accompanying drawings, in which:

[0016] Figure 1 is a flowchart showing a language recognizer for the Digital Asset Modeling Language (DAML™) in accordance with an exemplary embodiment of the present disclosure;

[0017] Figure 2 is a hybrid analogical diagram showing state changes relative to actions in accordance with an exemplary embodiment of the present disclosure;

[0018] Figure 3 is a hybrid analogical diagram showing digitized states relative to the state changes of Figure 2 in accordance with an exemplary embodiment of the present disclosure;

[0019] Figure 4 is a hybrid analogical diagram showing recordation of the states of Figure 3 to a ledger in accordance with an exemplary embodiment of the present disclosure;

[0020] Figure 5 is a hybrid analogical diagram showing change of state, securing, and authorization of the ledger of Figure 4 in accordance with an exemplary embodiment of the present disclosure;

[0021] Figure 6 is a hybrid analogical diagram showing replication of the modifiable, secured and authorized ledger of Figure 5 in accordance with an exemplary embodiment of the present disclosure;

[0022] Figure 7 is a hybrid analogical diagram showing distributed consensus for the replicated ledger of Figure 6 in accordance with an exemplary embodiment of the present disclosure;

[0023] Figure 8 is a hybrid analogical diagram showing unnoticed propagation of invalid or inconsistent state within the distributed ledger of Figure 7 in contrast with an exemplary embodiment of the present disclosure;

[0024] Figure 9 is a hybrid analogical diagram showing noticed propagation of invalid or inconsistent state within the distributed ledger of Figure 8 in contrast with an exemplary embodiment of the present disclosure;

[0025] Figure 10 is a hybrid analogical diagram showing entry validations and prevention of invalid or inconsistent states within the ledger of Figures 8 and 9 in accordance with an exemplary embodiment of the present disclosure;

[0026] Figure 11 is a hybrid analogical diagram showing dynamic strengthening and extension of the validation process for the ledger of Figure 10 in accordance with an exemplary embodiment of the present disclosure;

[0027] Figure 12 is a hybrid diagram showing DAML™ code using the Await function with defined actions for single party in accordance with an exemplary embodiment of the present disclosure;

[0028] Figure 13 is a hybrid diagram showing DAML™ code using defined actions for real-world agreement among multiple parties within a conditional in accordance with an exemplary embodiment of the present disclosure;

[0029] Figure 14 is a hybrid diagram showing DAML™ code operating on native fungible digital assets in accordance with an exemplary embodiment of the present disclosure;

[0030] Figure 15 is a hybrid diagram showing DAML™ code with alternating actions by two parties versus the single party logic of Figure 12 in accordance with an exemplary embodiment of the present disclosure;

[0031] Figure 16 is a hybrid diagram showing DAML™ code combining multiple steps in accordance with an exemplary embodiment of the present disclosure;

[0032] Figure 17 is a hybrid diagram showing DAML™ code for an asset swap between two parties in accordance with an exemplary embodiment of the present disclosure;

[0033] Figure 18 is a hybrid diagram showing DAML™ code choice in accordance with an exemplary embodiment of the present disclosure;

[0034] Figure 19 is a hybrid diagram showing DAML™ code composition in accordance with an exemplary embodiment of the present disclosure;

[0035] Figure 20 is a hybrid diagram showing DAML™ ordered ledger entries in accordance with an exemplary embodiment of the present disclosure;

[0036] Figure 21 is a hybrid diagram showing a DAML-based ledger with ordered ledger entries in accordance with an exemplary embodiment of the present disclosure;

[0037] Figure 22 is a hybrid diagram showing a DAML-based ledger with ordered and timestamped ledger entries in accordance with an exemplary embodiment of the present disclosure;

[0038] Figure 23 is a hybrid diagram showing two exemplary DAML™ storage and ledger logic deployments across multiple parties in accordance with an exemplary embodiment of the present disclosure;

[0039] Figure 24 is a hybrid diagram showing a DAML™ transaction with ledger entries in accordance with an exemplary embodiment of the present disclosure;

[0040] Figure 25 is a hybrid diagram showing a DAML™ transaction with ledger entries and secrecy in accordance with an exemplary embodiment of the present disclosure;

[0041] Figure 26 is a hybrid diagram showing a DAML-based ledger including multi-party authorization of commitment of new ledger entry transactions in accordance with an exemplary embodiment of the present disclosure;

[0042] Figure 27 is a hybrid diagram showing a DAML-based two-tiered ledger including hash versus entry and details in accordance with an exemplary embodiment of the present disclosure;

[0043] Figure 28 is a hybrid diagram showing a DAML-based hash-centric public ledger tier or public log in accordance with an exemplary embodiment of the present disclosure;

[0044] Figure 29 is a hybrid diagram showing a DAML-based private and sharable private ledger in accordance with an exemplary embodiment of the present disclosure;

[0045] Figure 30 is a hybrid diagram showing a two tier DAML™ ledger with associated ledger entry identification and ledger entry liveliness tracking in accordance with an exemplary embodiment of the present disclosure;

[0046] Figure 31 is a hybrid diagram showing a two tier DAML™ ledger with block-oriented logic, and with associated ledger entry identification and block-centric liveliness tracking in accordance with an exemplary embodiment of the present disclosure;

[0047] Figure 32 is a hybrid diagram showing party to party sharing of certification or provable properties of private ledger entry properties in accordance with an exemplary embodiment of the present disclosure;

[0048] Figure 33 is a hybrid diagram showing hosted and non-hosted copies of a distributed ledger in accordance with an exemplary embodiment of the present disclosure;

[0049] Figure 34 is a hybrid diagram showing DAML™ ledger entry types for Agree, Await, and Delete commands in accordance with an exemplary embodiment of the present disclosure;

[0050] Figure 35 is a hybrid diagram showing usage of the DAML™ Agree command in accordance with an exemplary embodiment of the present disclosure;

[0051] Figure 36 is a hybrid diagram showing ledger entries that are instantiated in accordance with an exemplary embodiment of the present disclosure;

[0052] Figure 37 is a hybrid diagram showing DAML™ code of an agreement for external notification within an equity model in accordance with an exemplary embodiment of the present disclosure;

[0053] Figure 38 is a hybrid diagram showing DAML™ code using the Await command within a cube model in accordance with an exemplary embodiment of the present disclosure;

[0054] Figure 39 is a hybrid diagram showing a DAML™ ledger with delete commands in accordance with an exemplary embodiment of the present disclosure;

[0055] Figure 40 is a hybrid diagram showing a ledger algorithm with role-centric flows in accordance with an exemplary embodiment of the present disclosure;

[0056] Figure 41 is a hybrid diagram showing a ledger algorithm with role-centric flows and comments in accordance with an exemplary embodiment of the present disclosure;

[0057] Figure 42 is a hybrid diagram showing a party-centric ledger algorithm flow in accordance with an exemplary embodiment of the present disclosure;

[0058] Figure 43 is a hybrid diagram showing a ledger-centric ledger algorithm in accordance with an exemplary embodiment of the present disclosure; and

[0059] Figure 44 is a hybrid diagram showing a function to initiate a ledger transaction in accordance with an exemplary embodiment of the present disclosure.

[0060] Figure 45 is a schematic of a digital system to model and/or record a digital asset according to one example;

[0061] Figure 46 is a flow chart of a computer-implemented method of manipulating data structures to model and/or record a digital asset and its evolution;

[0062] Figure 47 is a flowchart of another computer-implemented method of manipulating data structures to model and/or record a digital asset and its evolution;

[0063] Figure 48 is a flowchart of additional steps to the methods described in Figures 46 and 47;

[0064] Figure 49 is a schematic diagram representing a contract template;

[0065] Figure 50 is a schematic diagram of interactions related to an active contract;

[0066] Figure 51 shows a representation of an await function instance;

[0067] Figure 52 is a schematic diagram of an inactive contract and a new contract; and

[0068] Figure 53 is an example of a processing device.

DETAILED DESCRIPTION

[0069] The present inventive concept will be described more fully with reference to the accompanying drawings, in which exemplary embodiments are shown. The present inventive concept may, however, be embodied in many different forms and should not be construed as being limited to the embodiments set forth herein. Like reference numerals may refer to like elements throughout this description. As used herein, the word "model" is defined as at least one bundle of agreement(s) or potential transaction(s), which, under certain governing rules such as may be provided by a Master Contract, for example, might or might not have the potential to represent a digitally-represented agreement or a legally binding contract.

[0070] As shown in Figure 1, a language recognizer for the Digital Asset Modeling Language™ (DAML™) is indicated generally by the reference numeral 100. The language recognizer is a routine executed under the control of a computer processor. DAML™ includes predefined statements that can be stored as a computer readable data structure. The language recognizer

includes a start block 110, which passes the control to an input block 112. The input block 112 receives computer executable source code, such as from a memory or communications channel, and passes control to a function block 116. The block 116 performs both lexical analysis and syntax analysis on the received source code with the help of monadic parser combinator block 114 defined for the DAML™ language, and when successful produces an abstract syntax tree 118 of the parsed DAML™ language, which, in turn, passes control to an end block 120. The abstract syntax tree 118 is stored as a data structure. The current exemplary embodiment of the DAML™ processing logic takes DAML™ abstract syntax tree structures and transforms it to a new DAML™ abstract syntax tree, and is based on each DAML™ ledger entry data structure storing serialized DAML™ abstract syntax tree data as a data structure. Alternate embodiments of the DAML™ processing logic can use alternative parser technologies to translate the DAML™ source code to DAML™ abstract syntax trees, such as, for example, recursive descent. Alternate embodiments of the DAML™ ledger entry can store data-centric representations of the DAML™ language expressions, such as, for example, by compiling the DAML™ abstract syntax trees further with the help of a type inference step and a defunctionalization step to replace the higher order functional constructions of the DAML™ language with first-order constructions.

[0071] In operation, a DAML™ recognizer, under the control of a computer processor, interprets functions and syntax suitable for modeling digital assets. An exemplary embodiment DAML™ recognizer implements all DAML™ core language features.

DAML™ comments are supported using a Haskell style such as:

```
-- this is a line comment
{-
This is a multi-line comment. And comments {- can be -- nested -}
-}
```

[0072] The primitive types include the following: Bool is the two Boolean values True and False. Text is sequences of Unicode characters. Integer is signed integers of arbitrary size. Decimal is floating point decimal numbers. Party is unique identifiers for legal entities. Time is absolute points in time with implementation-dependent precision. RelTime is nominal differences between points in time (inheriting the same precision). ContractId is an identifier of a concrete instance of a contract (the digital equivalent to the physical identity of the piece of paper that a concrete contract is printed on).

[0073] There is a composed type for function types, which are built using the -> operator as in Haskell. For example, "Integer -> Text" is for a function that takes one argument of type Integer and returns a Text value.

[0074] DAML™ additionally supports the following special types: Record is a record of labeled values whose access is checked at runtime. Choice is a choice offered to a party. In an exemplary embodiment, each choice is made by a party. In an alternate embodiment, a model contract can incorporate the notion of a party's delegate or and/or an authorization scheme such as a public key infrastructure (PKI) such that a party's delegate may make such choice. Agreement is an agreement about an off-ledger event that must happen. Contract covers the terms of a model or potential contract. Update is a description of a ledger-update, which creates and deactivates active models in the ledger. Scenario is a description of a contractual interaction of multiple parties, which DAML™ embodiments may use for unit testing. Assertion is an assertion that a certain Boolean condition evaluates to True. PureValue is a value wrapped such that it can be used as a side-effect-free step both in a ledger-update or a scenario.

[0075] The following words are keywords of the language: await, at, named, if, chooses, then, else, such, that, exercises, agree, agrees, with, on, let, in, create, commit, commits, scenario, update.

[0076] Expressions over primitive types are built from literals, built-in functions, function application, and lambda abstraction. The two Bool literals are True and False. Text literals are written using double-quotes and use the same escaping rules as Haskell String literals (cf. section 2.6 of the Haskell Report 2010). An exemplary text literal is "Hello world" denoting the string 'Hello world'. Integer literals are written as base-10 numbers, with a possibly prefixed - for negative numbers. Examples are 1024 and -1. A Decimal literal represents a floating point decimal number. It is distinguished from an integer literal by having a decimal point. For example, 1234.56789, 0.5 or 1.0. The general form of decimal literals is given by the regular expression [0-9]+\.[0-9]+. Note that DAML™ ignores trailing zeros in Decimal literals; e.g., 1.5 == 1.50 == 1.5000. Party literals are written as sequences of alphanumeric characters between single-quotes. Examples are 'CITI GROUP' and 'DA'. In an exemplary embodiment, it is assumed that alphanumeric characters are enough to form the unique identifiers for legal entities. In an alternate embodiment, a concrete standard (e.g., LEI) may be used for referencing legal entities.

[0077] Time literals are always interpreted as UTC times and are written as literals according to ISO-8061 and restricted to the subset that is accepted by the following regular expression: [0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}(:[0-9]{2}(\.[0-9]+)?)?Z. For example, 2007-04-05T14:30Z, 2007-04-05T14:30:00Z and 2007-04-05T14:30:00.0Z all denote the UTC time 14:30 on the 5th of April 2007. Time literals can have arbitrary sub-seconds precision. It is however implementation-defined how much of that precision can actually be represented. A RelTime literal represents the nominal differences between two absolute points in time. The nominal difference

between two timepoints is always the same, regardless of whether a leap second has been or will be introduced. A relative time is represented as a fraction of seconds, e.g., toRelTime (24.0 * 60.0 * 60.0) denotes the amount of seconds that constitute a day (see below for an explanation of toRelTime). While RelTime literals can specify arbitrary fractions of seconds, it is implementation-defined how much of that precision can actually be represented. However, all implementations guarantee that RelTime and Time have the same precision. In alternate embodiments, alternative literals such as '1d' may be used to denote the relative time between the start and the end of a single day in the maximum precision of the implementation.

[0078] There is no fixed means for writing literals of type ContractId, as the form of ContractIds depends on the execution model. Records are written using curly braces as in JavaScript. For example, {"x": 1, "y": 2} is a Record with two fields labeled x and y whose values are 1 :: Integer and 2 :: Integer. DAML™ embodiments currently do not track what fields are present in the types. Moreover, you can only use literal strings for the labels. DAML™ embodiments use {} for the empty record.

[0079] DAML™ embodiments support the standard Haskell operators ||, &&, not for Boolean disjunction, conjunction, and negation (cf. Chapter 9 of the Haskell Report 2010). DAML™ embodiments support the standard arithmetic operators +, -, * , /, ^ for addition, subtraction, multiplication, integer division, and exponentiation for Integers with the usual precedence. As in Haskell, they can be written in parentheses to use them in prefix notation. For example, (+) 1 2 is another way of writing 1 + 2. In contrast to Haskell, DAML™ embodiments require infix operators to be surrounded by space. This resolves the ambiguity of 2 - -2 to mean2 - (-2). See below for overloaded versions of - and + when working with absolute and relative time.

[0080] DAML™ embodiments define the set of Decimals to be all rational numbers d for which there exist integers n and k such that d == n / 10 ^ k. That is, Decimals are a small subset of decimals, which include terminating, non-terminating, repeating, and non-repeating rational and irrational numbers when written in decimal form. Decimals (with a capital "D") include only the terminating subset of decimals. The following explains how DAML™ supports working with Decimals. DAML™ embodiments overload the arithmetic operators +, -, * to perform addition, subtraction and multiplication of Decimals. None of these operators perform any rounding, as the set of Decimals is closed under addition, subtraction and multiplication.

[0081] DAML™ embodiments support rounding of Decimals to a given precision with the function round :: Integer -> Decimal -> Decimal where round prec d rounds the Decimal d to the nearest Decimal of the form n / 10 ^ prec and ties are resolved by rounding towards the Decimal with an even n, which is the also known as the Banker's rounding mode. For example,

round 0 2.5 == 2.0	round 0 (-2.5) == -2.0
round 0 3.5 == 4.0	round 0 (-3.5) == -4.0
round 0 3.2 == 3.0	round 0 (-3.2) == -3.0
round 0 3.8 == 4.0	round 0 (-3.8) == -4.0

[0082] There is no "right" rounding mode. DAML™ embodiments may add further rounding modes on demand.

[0083] Note that the set of Decimals is not closed under division; e.g., dividing two Decimals does not necessarily result in a Decimal. For example, 1.0 / 3.0 == 0.3333... is a rational number, but not a (terminating) Decimal. DAML™ embodiments therefore provide the function divD :: Integer -> Decimal -> Decimal -> Decimal to compute the result of the rational division of two decimals rounded to a given Integer precision. The rounding mode is the same as the one of round. DAML™ embodiments provide access to the remainder of this division with the function remD ::

Integer -> Decimal -> Decimal -> Decimal. The relation between divD and remD is such that the following laws hold: 1) forall prec x y. y * divD prec x y + remD prec x y == x. 2) forall prec x y. abs (remD prec x y) < abs y. 3) forall prec x y. sign (remD prec x y) = sign y * sign x.

[0084] DAML™ embodiments support the conversion of Decimals from and to Integer with the functions fromInteger :: Integer -> Decimal, and toInteger :: Decimal -> Integer, where toInteger d converts the result of round 0 d. DAML™ embodiments assume that all execution models internally represent RelTimes as Decimals at a fixed, but implementation-specific precision. DAML™ embodiments therefore support converting Decimals from and to RelTime with the functions fromRelTime :: RelTime -> Decimal, and toRelTime :: Decimal -> RelTime, where fromRelTime provides a the Decimal number of seconds corresponding to the RelTime and toRelTime d rounds the given number of d seconds to the implementation-specific precision of RelTime. Note that this implies toRelTime (fromRelTime dt) == dt for all dt.

[0085] DAML™ embodiments support a total order for all primitive types as follows. Bool: False < True. Text: implementation-defined. Integer: total ordering of integers. Decimal: total ordering of decimals. Party: implementation-defined. Time: absolute time is ordered along a linear axis of time. More formally, ∀ t1, t2 :: Time. t1 < t2 ⇔ (t1 - t2) < toRelTime 0 1. RelTime: relative time is ordered according to its Decimal representation. ContractId: implementation-defined.

[0086] DAML™ embodiments support equality for all primitive types such that a == b ⇔ not (a < b ∨ b < a). DAML™ embodiments are using the same infix operators as Haskell does, e.g., DAML™ embodiments use ==, /=, <=, >=, <, > to denote equality, inequality, less-or-equal-than, greater-or-equal-than, less-than, and greater-than, respectively (cf. Chapter 9 of the Haskell Report 2010). Additionally DAML™ embodiments support the operator (~) :: ContractId -> Contract ->

Bool, where coid ~ co means that coid is a contract-id that refers to an active contract in the ledger, and this active contract is an instance of co.

[0087] DAML™ embodiments support the concatenation of Text values using the (<>) :: Text -> Text -> Text operator. DAML™ embodiments support an overloaded operation toText that converts any value of primitive type to a textual value in an implementation-defined manner. DAML™ embodiments also support a show operation that converts any value of any type to a textual value in an implementation-defined manner. This is used for debugging during scenario development.

[0088] DAML™ embodiments overload the + and - operators such that (+) :: Time -> RelTime -> Time denotes shifting of a point in time by a relative time and (-) :: Time -> Time -> RelTime denotes the computation of the time difference between the first and the second argument. The relation between these operators is such that t1 + (t2 - t1) == t2 for all t1, t2 :: Time.

[0089] Record access is written using brackets as in JavaScript. For example, the expression r["x"] tries to access the field labeled "x" in the record r. Record values are constructed using curly braces and literal strings as in JavaScript. For example, {"x": 1, "y": 2} is a record with two fields "x" and "y".

[0090] DAML™ embodiments use lambda abstractions as in Haskell, but always annotated with types; e.g., \(a :: Integer) (b :: Integer) (c :: Text) -> toText (a + b) <> c. Function application is written as in Haskell using juxtaposition. It is left-associative and binds stronger than all infix operators.

[0091] DAML™ embodiments use non-recursive let-bindings as in Haskell, but multiple bindings have to be separated by a semicolon; e.g., let x = 4; y = 5; in x + y. The formal syntax is given by:

let var1 = expr1; var2 = expr2; ... varN = exprN; in expression. Recursive let-bindings are not allowed.

[0092] DAML™ embodiments use lazily evaluated if-then-else branching expressions as in Haskell. The formal syntax is: if condition then expression1 else expression2. For example, if owner == 'ACME' then "sell" else "buy".

[0093] The DAML™ is based on the assumption that the main purpose of models is to describe what agreements come into effect when and under what constraints. These agreements form the connection of models in the ledger with effects in the real world. The meaning of agreements may be determined by human interpretation and optionally in the context of an applicable legal system, as it is done for paper-based contracts.

[0094] An introductory DAML™ embodiment includes agreements, choices and models. The formal syntax and semantics of choices, ledger-updates, and models are provided. Alternate DAML™ embodiments provide further examples to help deepen the understanding of this modeling language.

[0095] In an exemplary embodiment, DAML™ embodiments specify agreements using the agree keyword, whose syntax is party1, ..., partyN agree textOfAgreement. For example: 'UBS', 'Alice' agree "'UBS' deposits 100 CHF on account 'CH42 1234 5'." is meant to denote the agreement between 'UBS' and 'Alice' that 'UBS' must deposit the 100 Swiss Francs (CHF) on the account 'CH42 1234 5' at the point in time when this agreement comes into effect.

[0096] DAML™ embodiments convert such agreements to templates of agreements using lambda abstraction. For example, the definition deposit = \(obligor :: Party) (owner :: Party) (amount :: Integer) (account :: Text) -> obligor, owner agree toText obligor <> " deposits " <> toText amount

<> " CHF on '" <> account <> "'."; allows one to write the above agreement as: deposit 'UBS' 'Alice' 100 "CH42 1234 5".

[0097] It is up to the writer of models to make sure that the textual description of an agreement is precise in the context of the legal system in which the models are interpreted. From the perspective of the model language, DAML™ embodiments do not interpret the legal text of agreements, but consider every agreement between parties to be an obligation of these parties.

[0098] DAML™ embodiments model permissions of parties by giving them choices. For example, the choice 'Alice' chooses account :: Text then deposit 'UBS' 'Alice' 100 account means that 'Alice' can choose an account on which 'UBS' must deposit 100 CHF. Models are groups of choices which are simultaneously available to (possibly multiple) parties. For example, iouSellSettle = \(obligor :: Party) (owner :: Party) (amount :: Integer) -> await { "settle": owner chooses account :: Text then deposit obligor owner amount account, "sell": owner chooses newOwner :: Party then iouSellSettle obligor newOwner amount }; is a template for an I-owe-you model (IOU) in CHF that can be settled or sold. The model iou 'Alice' 'UBS' 100 denotes a concrete IOU modeling that 'UBS' owes 'Alice' 100 CHF.

[0099] The previous iouSellSettle example provides intuition on how models are specified. In the following, DAML™ embodiments will give precise definitions for both the syntax and the semantics of choices and models. These definitions are non-trivial to explain because they are mutually recursive via a third concept, which DAML™ embodiments call a ledger-update. Intuitively, these three concepts are related as follows. 1. A model is a group of mutually exclusive choices. 2. A choice is a permission given to a specific party to perform a specific kind of ledger-update at a specific time. 3. A ledger-update is a description of how to update the ledger by creating

new agreements and models, exercising choices on existing models, deleting existing models, or asserting statements about the ledger's state at specific times.

[00100] In the following, DAML™ embodiments first give and explain the syntax of models, choices, and ledger-updates. Then, DAML™ embodiments explain how the state of the ledger is represented and how DAML™ embodiments transition between states using ledger-updates. DAML™ embodiments specify models using the following syntax. await identified as contractIdBinder named contractNameExpr { "choiceName1": choice1 , ... , "choiceNameN": choiceN }. Such an expression specifies a model that provides N choices labeled choiceName1 to choiceNameN to the controlling parties of these choices. The variable contractIdBinder is of type ContractId and can be used to refer to the identity of the concrete model instance in the definition of the choices 1 to N as well as in the contractNameExpr expression. ThecontractIdBinder is useful to relate multiple choices by giving them the unique contract-id as an argument. The variablecontractName is of type Text and is an arbitrary name for the model to facilitate debugging. The named contractName and the identified as contractIdBinder parts can be left out in case they are not needed.

[00101] DAML™ embodiments specify choices using the following syntax. controllingPartyExpr chooses valueBinder1 :: Type1, ..., valueBinderL :: TypeL at choiceTimeBinder such that booleanChoiceCondExpr then updateExpr. Such a Choice expression specifies that a choice is given to the party denoted by controllingPartyExpr to choose values of the required types at some time in the future, and if these values and the time at which they were chosen satisfy the booleanChoiceCondExpr, then ledger-update denoted by updateExpr is executed. The choice only succeeds if this ledger-update executes successfully.

[00102] Ledger-updates reflect transactions, are stored as a data structure and are constructed using one of the following built-in functions or using an update-block. create :: ContractOrAgreement -> Update. delete :: ContractId -> Contract -> Update. exercises :: Party -> Label -> Any -> .... -> Any -> ContractId -> Update. assert :: Bool -> Update. pure :: Any -> Update. Intuitively, DAML™ embodiments use create to create models or agreements, delete to deactivate active models, exercises to force parties to execute choices on active models, assert to assert statements about the current state of the ledger, and pure to construct ledger updates that just return a value but do not change the ledger.

[00103] An update-block allows one to execute multiple consecutive ledger-updates as one atomic ledger-update. Update-blocks are specified using the following syntax. update [updateStatement1 ~> binder1 , ... , updateStatementN ~> binderN , lastUpdateStatement]. DAML™ embodiments use the squiggly arrows ~> to name the results of the execution of individual update-statements. These names can then be used in later update-statements to refer to these results.

[00104] Models can be recursive, as shown in the previous iouSellSettle example. The way to specify such recursion is via the use of named top-level definitions, as explained later in the Section "DAML™ Programs". DAML™ embodiments reduce the syntactic overhead of specifying models using syntactic sugar. DAML™ embodiments allow for models and agreements to be used as updates, and DAML™ embodiments allow for records of updates to be used as updates.

[00105] When using a model c as an update, it's effect is the same as the one of create c. DAML™ embodiments add this feature as creating models is the most common update action. When using a record {"l1": upd1, ..., "lN": updN} as an update, then it's effect is the same one of

update [upd1 ~> v1 , ... , updN ~> vN , pure {"l1": v1, ..., "lN": vN}]. DAML™ embodiments add this feature as DAML™ embodiments often need to return results of many intermediate update actions.

[00106] DAML™ embodiments explain the semantics of models, choices, and ledger-updates as a transition system. The state of this transition system is a ledger, which is a finite map from ContractIds to models and a log of agreements that are in effect. The transitions are given by the interpretation of ledger-updates. Note that DAML™ embodiments specify for each interpretation both how the ledger is changed and what result value is returned by the ledger-update. DAML™ embodiments require these results when interpreting update-blocks, where later ledger-updates can refer to the results of earlier ledger-updates.

[00107] DAML™ embodiments interpret an expression create agreementExpr by evaluating agreementExpr, checking that it is indeed an agreement of the form party1, ..., partyN agree legalText, and then recording that this agreement is now in effect. The result of this ledger-update is the agreement itself.

[00108] DAML™ embodiments interpret an expression create contractExpr by evaluating contractExpr. Provided this evaluation succeeds with a model co, DAML™ embodiments store this model co under a freshly allocated contract-id coid. The result of this ledger-update is the contract-id coid.

[00109] DAML™ embodiments interpret an expression delete contractIdExpr contractExpr by first evaluating both contractIdExpr andcontractExpr. Provided this evaluation success with a literal contract-id coid and a model co, DAML™ embodiments then check thatcoid identifies an active model equal to the model co. Provided that is the case, DAML™ embodiments remove coid from the ledger. Note that DAML™ embodiments require the model co to be specified as part of

the delete to enable the static analysis of which parties are affected by this delete. The result of a delete is the empty record {}.

[00110] DAML™ embodiments interpret a ledger-update of the form partyExpr exercises "choiceLabel" with choiceValue1, ... choiceValueN on coid as follows. DAML™ embodiments first evaluate partyExpr. Provided this results in a literal party name actor, DAML™ embodiments lookup the active model associated with coid. Provided coid identifies an active model co, DAML™ embodiments mark it as inactive. Then, DAML™ embodiments lookup the choice identified by "choiceLabel" in co. Provided the actor is equal to the choice's controller, DAML™ embodiments exercise this choice; e.g., DAML™ embodiments first instantiate both the choice's condition and its follow-up ledger-update with the given choiceValues and the current time. Then, DAML™ embodiments check the choice's condition and, provided that succeeds, then interpret the choice's follow-up. The result of a ledger-update built using exercises is the result of the choice's follow-up. Note that this interpretation is potentially recursive, but it is guaranteed to terminate.

[00111] DAML™ embodiments interpret an expression assert booleanExpr by evaluating the booleanExpr, and then checking whether the result isTrue. If this is the case, then interpretation succeeds. Otherwise, it fails. The result of an assert ledger-update is the empty record {}.

[00112] DAML™ embodiments interpret an expression pure x by evaluating x and returning it as the result of this ledger-update. DAML™ embodiments can therefore use pure to construct side-effect free ledger-updates that return a specific result.

[00113] DAML™ embodiments interpret an update-block update [updateStatement1 ~> binder1 , ... , updateStatementN ~> binderN , lastUpdateStatement] by interpreting the updateStatements one after the other after substituting the results of the previous updateStatements

for the binders. The result of an update-block is the result of the lastUpdateStatement. The interpretation of the update-block fails if any of its statements fails; and all effects on the ledger are only applied, if the update-block succeeds. Update-blocks allow therefore to build atomic composite ledger-updates.

[00114] One can also define model templates that combine other model templates. For example, the definition option = \ (controller :: Party) (tlb :: Time) (tub :: Time) (next :: Contract) -> await { "exercise": controller chooses at t such that tlb <= t && t <= tub then next }; provides a combinator called option, which allows a party to enter a model during a specific time-interval.

[00115] In all previous examples, choices just lead to the creation of zero to many new models. The following example of an IOU that can be split and merged shows how to use an update-block in the "merge" choice to atomically delete (deactivate) the model merged-in and create the new IOU model over the larger amount. iouChf = \ (obligor :: Party) (owner :: Party) (amount :: Integer) -> await { "settle": owner chooses account :: Text then deposit obligor owner amount account , "sell": owner chooses newOwner :: Party then iouChf obligor newOwner amount , "split": owner chooses newAmount :: Integer such that 0 < newAmount && newAmount < amount then { "iou1": iouChf obligor owner newAmount , "iou2": iouChf obligor owner (amount - newAmount) } , "merge": owner chooses otherIou :: ContractId, otherAmount :: Integer then update [delete otherIou (iouChf obligor owner otherAmount) , iouChf obligor owner (amount + otherAmount)] };.

[00116] DAML™ embodiments can also use models to coordinate changes on other models by using choices that require specific parties to execute decisions. For example, the template below can be used to require a payer to transfer ownership of a payment until a certain time.

```
-- A template for a breach-of-contract that has to be negotiated between two
-- parties in front of Swiss court.
```

```
contractBreachedBy =
  \(defendant :: Party) (plaintiff :: Party) ->
     defendant, plaintiff agree
       toText defendant <> " has breached the contract, and " <>
       toText plaintiff <> " can sue " <> toText defendant <>
       " in any court of Switzerland according to Swiss law."
  ;
-- A template for a model requiring payment within a certain time via a
-- transfer of ownership of an IOU as defined above.
mustPayIouUntil =
  \ (payer      :: Party)
    (payee      :: Party)
    (payment    :: Contract)
    (maxPayTime :: Time)
  ->
    await
     { "pay":
       payer chooses paymentId :: ContractId
       such that
         -- a check that 'paymentId' refers to an active 'payment' model
         paymentId ~ payment
       then payer exercises "sell" with payee on paymentId
       -- punitive clause/choice becomes available to the payee after the
       -- time that was given to the payer to pay.
     , "breach":
       payee chooses at tbreached
       such that
         maxPayTime <= tbreached
       then contractBreachedBy payer payee
     };
```

A combinator that uses iouChf payments could then defined as follows.

```
payInChfUntil =
  \ (payer      :: Party)
    (payee      :: Party)
    (obligor    :: Party)
    (amount     :: Integer)
    (maxPayTime :: Time)
  ->
    mustPayIouUntil payer payee (iouChf obligor payer amount) maxPayTime;
```

[00117] DAML™ embodiments call a group of top-level definitions a DAML™ program. DAML™ embodiments currently separate these definitions using semi-colons due to the following local parsing ambiguity: a = b c = d would be parsed as a = b c with a parse failure for the following = d. DAML™ embodiments therefore use a trailing semi-colon for each top-level definition as follows. a = b; c = d to obviate the need for semi-colons.

[00118] Scenarios are descriptions of how multiple parties interact via a model or models stored in a ledger. DAML™ embodiments include a special notation for scenarios in the language because they serve as vital documentation and test-cases for model templates.

Here is an example scenario for the IOU definition in the previous section.

```
days = \(x :: Integer) -> toRelTime (fromInteger (x * 24 * 60 * 60));
createAndSettleIou =
  scenario
    [ commit (iouSellSettle 'UBS' 'UBS' 100) ~> ubsIou
    , 'UBS' commits 'UBS' exercises "sell" with 'Alice' on ubsIou ~> aliceIou
    , assert (aliceIou ~ iouSellSettle 'UBS' 'Alice' 100)
    , pass (days 10) ~> now
    , 'Alice' commits 'Alice' exercises "settle" with "CH42 1234 5" on aliceIou ~> settled
    , assert (settled == (
        'UBS', 'Alice' agree "'UBS' deposits 100 CHF on 'CH42 1234 5'."))
    ];
```

[00119] As for update-blocks, DAML™ embodiments use squiggly arrows to bind the results of the scenario actions; and the result of a scenario is the result of its last step. The types of the bound variables in the above example are the following. ubsIou :: ContractId; aliceIou :: Record; now :: Time; settled :: Record;. Note that the form of the aliceIou record will be {"iou": contractId} and the one of the settled record will be {"settle": agreement}. This is determined from the labeling of the follow-ups in the "sell" and "settle" choices in the iou definition above.

[00120] DAML™ embodiments can pattern match into the records of the results of the scenario steps. For example 'Alice' exercises "settle" with "CH42 1234 5" on aliceIou["iou"] ~> {"settle" : agreement}. This binds the "settle" entry of the result record of the step to the variable name agreement. DAML™ embodiments can pattern match records to arbitrary depth, e.g., this is a valid pattern: {"foo" : {"bar" : {"baz" : varName}}}.

[00121] Patterns do not need to be complete; that is, labels that are present in the result record can be omitted in the pattern. A pattern match against a label that is not present in the result record will cause an error (e.g., a runtime error if interpreted). Shadowing of variable names is determined by the order of their label keys. In the previous example the label key of the variable varName is ["foo", "bar", "baz"]. This variable would shadow one with a label key of ["foo", "bar"]. In the example {"a" : varName, "b" : varName} varName binds the entry accessed by ["b"], because the key ["b"] comes after the key ["a"].

[00122] The default interpretation of a scenario is as follows. Starting with an empty ledger the steps of a scenario are executed in order.

[00123] Committing ledger-updates. An expression of the form party1, ..., partyN commit updateExpr denotes that the parties party1, ..., partyN jointly agree that they want to commit the ledger-update denoted byupdateExpr. This succeeds if the updateExpr can successfully be

interpreted as a state transition on the current ledger. DAML™ embodiments require an explicit specification of the parties that agree to the commit, as these are the only parties that are allowed to be obligable in the interpreted ledger-update. Alternate embodiments may specify the concept of obligable parties.

[00124] Controlling Time. A pass relTimeExpr ~> newTimeBinder step advances the current scenario time by relTimeExpr and binds the new scenario time to newTimeBinder. One can use this feature to determine the initial time of the scenario as follows: pass (toRelTime 1.0) ~> now.

[00125] Expressing Expectations. A assert booleanExpr step evaluates the Boolean expression and fails the scenario if that expression is false. It has no effect otherwise. The mustFail keyword can decorate a scenario step to indicate that it is supposed to fail. Such a step fails if the inner step does not, and vice versa. For example, it makes sense to state that a model cannot be settled twice.

```
mustFailExample = scenario
  [ 'UBS' commits (iouSellSettle 'UBS' 'Alice' 100) ~> iouId
  , 'Alice' commits 'Alice' exercises "settle" with "CH12 345 6" on iouId
  , mustFail ('Alice' exercises "settle"  with "CH12 345 6"on iouId)
  ]
;
```

[00126] Debugging. The trace textExpr step evaluates the Text expression and creates a scenario step that does not change the ledgers in any way, but contains the evaluated text in the description of the step, so this can be used for debugging a scenario and looking into values using the toText function. Example: trace ("gergely's trace" <> toText (42 + 123)).

[00127] One can annotate any expression with a textual expression using description annotations. These are written using the following syntax. {@ DESC textExpr @} someOtherExpression andItsArguments. Description annotations bind as far right as possible, and you'll have to use parentheses if you only want to annotate a function itself, e.g.,

```
\(f :: Integer -> Integer) (arg :: Integer) ->
  ({@ DESC "the function's description" @} f)
  ({@ DESC "the argument's description" @} arg)
```

[00128] DAML™ embodiments use this method as this requires fewer parentheses in the common case of annotating whole models or choices. Note that in the case of multiple annotations on the same expression, the inner-most annotation is kept and the other ones are ignored. In particular, DAML™ embodiments use description annotations to abbreviate models using non-legally-binding human-readable text for GUI purposes. For example, DAML™ embodiments can introduce named iou models as follows.

```
iou =
 \ (obligor :: Party) (owner :: Party) (amount :: Integer)
 ->
  {@ DESC
    toText obligor <> " --(" <> toText amount <> ")--> " <> toText owner
  @}
  await
   { "sell": owner chooses newOwner :: Party then {"iou": iou obligor newOwner
amount }
   , "settle":
       owner chooses account :: Text
       then deposit obligor owner amount account
   }
;
traceExample =
```

```
  scenario
    [ 'Bank' commits (iou 'Bank' 'Alice' 1) ~> alice
    , 'Bank' commits (iou 'Bank' 'Bob'   2) ~> bob
    , trace ("Bob's contract: " <> toText bob)
    ]
;
```

[00129] Description annotations on await keywords are remembered by the interpreter and used when printing the final ledger. For example, when finishing with a scenario, DAML™ embodiments can have this output.

```
final ledger:
  [ contract 0c created at 1970-01-01T00:00:00Z 'Bank' --(1)--> 'Alice'
      [ "sell":
        'Alice'
        chooses newOwner5 :: Party
        then { "iou": iou newOwner5 'Bank' 1 }
      ]
  , contract 1c created at 1970-01-01T00:00:00Z 'Bank' --(2)--> 'Bob'
      [ "sell":
        'Bob'
        chooses newOwner6 :: Party
        then { "iou": iou newOwner6 'Bank' 2 }
      ]
  ]
```

[00130] Here DAML™ embodiments can see the short description of the IOU, that shows that the Bank owns Alice $1 and to Bob $2.

[00131] Many models require choices by multiple parties, but are indifferent to the order in which these choices are made. Such models can be specified with the language features described up to here. However, the size of these specifications is exponential in the number of parties that

must make a choice, as DAML™ embodiments must enumerate all possible orderings of choices. To avoid such an exponential blowup, DAML™ embodiments introduce explicit support for parallel choices by multiple parties.

[00132] DAML™ embodiments explain the support for parallel choices as follows. DAML™ embodiments first provide an introductory example. DAML™ embodiments then specify the formal syntax both for parallel choices and decisions on individual parallel choices. Finally, DAML™ embodiments explain the semantics of parallel choices.

[00133] The following example contract models a proposal for an option to swap two sellable contracts between two parties.

```
optionalSwap =
 \ (alice     :: Party)
   (aliceGood :: Contract)
   (bob       :: Party)
   (bobGood   :: Contract)
 ->
   await
    { "swap":
      {| "alice": alice chooses ca :: ContractId such that ca ~ aliceGood
       , "bob"  : bob   chooses cb :: ContractId such that cb ~ bobGood
       |}
      then
      { "bob's":   alice exercises "sell" with bob  on ca
      , "alice's": bob   exercises "sell" with alice on cb
      }
    , "alice cancels": alice chooses then {}
    , "bob cancels":   bob chooses   then {}
    };
```

[00134] In contrast to previous examples, the await choice's follow-up is guarded by two parallel choice-steps. The first step models alice's choice to pre-commit her good and the second step models bobs choice to pre-commit his good. These two steps are only pre-commits, as both alice and bob could still exercise a choice on their individual goods as long as the second party has not provided his good. As it is only an option to swap, DAML™ embodiments provide both alice and bob with a choice to cancel the swap. Both of these choices remain available as long as only alice or bob exercised their part of the "swap" choice. Note that the actual swapping of goods happens atomically once the second party has made its pre-commit choice.

The formal syntax of choices with parallel choice-steps is the following.

```
{| "choiceStep1":
   controllingPartyExpr1
   chooses
      valueBinder1_1 :: Type1_1, ..., valueBinder1_L :: Type1_L
    at choiceStepTimeBinder1
   such that
    booleanChoiceStepCondExpr1
   then followUpResultBinder1 <- followUpExpr1
 , ...
 , "choiceStepN":
   controllingPartyExprN
   chooses
      valueBinderN_1 :: TypeN_1, ..., valueBinderN_M :: TypeN_M
    at choiceStepTimeBinderN
   such that
    booleanChoiceStepCondExprN
   then followUpResultBinderN <- followUpExprN
|}
such that
```

```
  booleanChoiceCondExpr
 then followUpExpr
```

[00135] The scope of the value binders is such that the per-choice-step bound values can be referenced in the corresponding per-choice-step Boolean conditions; and all bound values can be referenced in the choice's Boolean condition and all the choice's follow-ups. Note that the group of choice-steps must be non-empty and all the steps must be labeled differently. DAML™ embodiments chose to use a different kind of parentheses, the {| and |} variants, to ensure that records and parallel choices are clearly distinguishable. DAML™ embodiments thereby keep the syntax unencumbered for later generalizations of records.

[00136] DAML™ embodiments extend the formal syntax of decisions such that one can not only reference the choice's label, but also the choice-steps' label. This allows specifying the decision of a party to execute one of its choice steps. The concrete syntax is the following. partyExpr exercises "choiceLabel" "choiceStepLabel" with choiceStepValue1, ..., choiceStepValueN on contractIdExpr. For example, the following scenario creates a optionalSwap of ious and exercises it.

```
optionalSwapTest =
 scenario
  [ 'UBS' commits (iouSellSettle 'UBS' 'Alice' 100) ~> aliceIou1
  , 'CS'  commits (iouSellSettle 'CS'  'Bob'   160) ~> bobIou1
  , commit
     ( optionalSwap 'Alice' (iouSellSettle 'UBS' 'Alice' 100)
                    'Bob'   (iouSellSettle 'CS'  'Bob'   160)
     ) ~> optSwapId1
  , 'Bob' commits   'Bob'   exercises "swap" "bob"   with bobIou1
     on optSwapId1 ~> optSwapId2
  , 'Alice' commits 'Alice' exercises "swap" "alice" with aliceIou1
     on optSwapId2 ~> { "alice's": aliceIou2
```

```
                    , "bob's":   bobIou2
                    }
        , assert (aliceIou2 ~ iouSellSettle 'CS'  'Alice' 160)
        , assert (bobIou2   ~ iouSellSettle 'UBS' 'Bob'   100)
        ];
```

[00137] The semantics of exercising parallel choices is the following. A choice c guarded by a single choice-step behaves the same way as a normal choice whose condition is the conjunction of the choice-step's condition and the condition of c. For choices guarded with more than one choice-step, exercising a decision actor exercises "choiceLabel" "step_i" with v1, ..., vN at time now on "step_i" of a choice c guarded by N choice-steps

```
{| "step_1": step_1,
 , ...
 , "step_i":
   ctrl chooses x1 :: type1, ..., xN :: typeN at t such that choiceStepCond
 , ...
 , "step_N": step_N
|}
such that
  choiceCond
then followUps
```

[00138] This works as follows. DAML™ embodiments first check whether actor controls step_i, then DAML™ embodiments check types of the values v1, ..., vN match the expected types, and finally DAML™ embodiments check whether the choice-step condition of step_i is satisfied. If all of these checks succeed, then DAML™ embodiments delete the current model, and create a new model, which records the decision for choice-step step_i.

[00139] More concretely, let's assume that the model c is the one pointed to by optSwapId1 in the above optionalSwapTestscenario. Then, the step 'Bob' exercises "swap" "bob" with bobIou1

on optSwapId1 ~> optSwapId2 executed at time t will mark optSwapId1 as inactive, and introduce a new contract-id optSwapId2 pointing to c after [exercising "swap" "bob" at t with bobIou1] where after is a special keyword marking a list of pending decisions that must be applied to a model.

[00140] Alternate DAML™ embodiments will allow the specification of pending decisions in the input-syntax of DAML. These features are fully specified and implemented in the reference semantics. However, they are not necessarily supported in all execution models. DAML™ embodiments therefore state for each feature in what execution models it is supported.

[00141] For example, HyperLedger with stakeholder ratification does not currently support the DAML™ features of Point-Wise Observables or Tabular Contract Query Language. Financial contracts often depend on external, public measurements like the closing price of a share, or the announcement of a dividend payment. DAML™ embodiments call such external, public measurements observable data. There are many models for formalizing observable data. DAML™ embodiments explore several of them in the daml-design-considerations.md document. Here, in this section, DAML™ embodiments explain the support for a simple model where 1. each party can publish their own named data-feeds of timestamped immutable data values, and 2. the models can depend on pointwise observations of these data-feeds.

[00142] DAML™ embodiments do not mandate that these data-values are published on the ledger. Instead, DAML™ embodiments assume that there is some means that allows parties to communicate values of data-feeds and to prove that a data-feed attained a certain value at a certain timepoint. Conceptually, the state of a data-feed corresponds at any time to a partially-defined, piecewise-constant function. As the state of a data-feed evolves the corresponding function becomes more and more defined, but never changes the value at an already defined timepoint. More formally, DAML™ embodiments can represent the state of a data-feed of values of type a

as a non-empty list of type type Feed a = [(Time, a)] where the timepoints are strictly monotonic. Let feed = [(t_0, v_0), (t_1, v_1), ..., (t_n, v_n)] be a non-empty list of type Feed. Then the corresponding partially-defined, piecewise-constant function is the function that maps the timepoint t to

1. v_i if there exists a consecutive pair of timed values (t_i, v_i) and (t_(i+1), v_(i+1)) in feed such that t_i <= t < t_(i+1),
2. v_n if t_n == t, and
3. undefined, if none of the above two conditions is satisfied.

[00143] Note that this definition implies that (t_n, v_n), the last timed value in feed, only defines the value of the corresponding function for t == t_n. For all t > t_n, the corresponding function is undefined.

[00144] As stated before, DAML™ embodiments do not fix a publication scheme for data-feeds to be used in a distributed ledger setting. However, one of the reasons for choosing the above definition is that it allows for a straightforward publication scheme. Namely, a publishing party can publish new values to a data-feed by signing them together with a hash-based link to the previous publication. For example, the following definitions exemplify publishing a data-feed at two consecutive timepoints t0 and t1 > t0.

```
hash_0 = hash(t_0, v_0);
feed_0 = sign('PublisherPk', ("feedName", hash_0))
hash_1 = hash(hash_0, (t_1, v_1));
feed_1 = sign('PublisherPk', ("feedName", hash_1))
...
```

[00145] The example assumes that the bytestrings corresponding to the hashes hash_0 and hash_1 are either inlined in the published messages or distributed via a content-addressable filesystem like IPFS. Obviously, there are many ways to improve the efficiency of such a publication scheme (e.g., building blocks of timed values, or using authenticated stream-broadcast

protocols like TESLA, or the like). DAML™ embodiments expect the majority of these efficiency improvements to be compatible with the assumptions on the definedness and immutability of data-feeds. Therefore, DAML™ embodiments do not further pursue the details of these publication schemes for data-feeds in this document.

[00146] In the reference implementation of the model language, DAML™ embodiments provide the following two functions to manage the querying of data-feeds and the publication of new timed values to party-specific data-feeds.

```
observeFeedAt :: Party -> Text -> Time -> Any
publishToFeed :: Party -> Text -> Any -> Scenario Unit
```

[00147] DAML™ embodiments use the expression observeFeedAt publisher feedName t to query the value that the data-feed feedName published by publisher has at time t. If the feed's value is not defined at t, then the evaluation of this expression fails. This will be the case if t is before the first published data-point of the feed feedName or after the last published data-point.

[00148] DAML™ embodiments use the expression publishToFeed publisher feedName expr to denote a scenario step that publishes the value ofexpr to the feed feedName of the publisher. The timestamp of this published value is the current scenario time, which implies that one cannot publish data into the future.

[00149] The following example demonstrates the interplay between observeFeedAt and publishToFeed.

```
seconds = \(t :: Integer) -> toRelTime (fromInteger t);
observeSixSmi = observeFeedAt 'SIX' "SMI";
publishToSixSmi = \(value :: Integer) ->
  scenario
    [ pass (seconds 0) ~> now
    , publishToFeed 'SIX' "SMI" value
```

```
    , pure now
    ];
publicationAndObservationTest =
  scenario
    [ publishToSixSmi 7000 ~> t0
    , assert (observeSixSmi t0 == 7000)
    , mustFail (assert (observeSixSmi (t0 - seconds 1) == 7000))
      -- ^ fails as the "SMI" feed of 'SIX' is undefined before 't0'
    , mustFail (assert (observeSixSmi (t0 + seconds 1) == 7000))
      -- ^ fails as the "SMI" feed of 'SIX' is not defined after the current
      -- scenario time, which is at this point equal to 't0'
    , pass (seconds 1) ~> t1
    , mustFail (assert (observeSixSmi (t0 + seconds 1) == 7000))
      -- ^ Fails as the last published value in the "SMI" feed of 'SIX' is
      -- at timestamp 't0'.
    , publishToSixSmi 8000
    , assert (observeSixSmi t1 == 8000)
      -- Now this succeeds as the value of the "SMI" feed of 'SIX' has been fixed
      -- at the current scenario time. DAML™ embodiments require this explicit fixing, as
      -- otherwise the observed value at a specific timepoint could change
      -- non-deterministically. With the current solution, DAML™ embodiments avoid this
      -- non-determinism as undefined values cannot be observed in
      -- choices of models.
    ];
```

[00150] The restriction that every feed must be published explicitly at every time is inspired from a distributed ledger standpoint, where every extension of the signed hash-chain must be published explicitly. However, with the above primitives this might prove to be a bit cumbersome in test scenarios. DAML™ embodiments therefore envision that alternate DAML™ embodiments

could introduce a function fixAllFeeds :: Scenario Unit that publishes the last value of all feeds as the value timestamped with the current scenario time, if such a function is necessary.

[00151] Model templates can be parameterized over data-feeds. The support for data-feeds described above is minimal. DAML™ embodiments nevertheless expect that it provides significant expressivity for formulating model templates with choices that depend on these external data-feeds. The key idea for formulating such model templates is to represent time-dependent rates as functions of type Time -> Decimal and time-dependent conditions as functions of type Time -> Bool. The caller of such a template can then use the given time value to inspect a data-feed.

[00152] Note that this decision means that functions in the language are not completely pure. They do depend on the state of the ledger (due to the ~ function), and on the state of the data-feeds (due to the observeFeedAt function). This is acceptable as DAML™ embodiments have a strict evaluation semantics, which ensures that it is always clear when DAML™ embodiments evaluate an expression. Evaluation semantics for alternate DAML™ embodiments may eagerly evaluate applications and arguments and stop once they encounter an unsaturated application, an await, or a choice.

[00153] Tabular Contract Query Language provides support for querying a contract-based ledger. The querying is based on syntactic pattern-matching of data structures, which DAML™ embodiments contemplate to play a central role in reading from a contract-based ledger. The language features described in this section serve as a basis with respect to how pattern-matching behaves in the context. This understanding may be used in the development of GUIs and the development of contract-specific automation for choice execution.

[00154] An execution environment, embodied on a computer system, stores models specified in DAML™ in a ledger. DAML™ embodiments want to provide the means to query

such a database for contracts active in the ledger and their parameters. For example, a DAML™ embodiment could query the ledger for how many active "IOU" contracts it currently contains. Or how many "IOU" contracts there are, where the obligor is named 'UBS'. A GUI might want to present the results of a query for all active "IOU"s in the ledger in tabular form as follows.

ContractId	CreatedAt	Amount	Obligor	Owner
0xffeeafa	2007-04-05T14:30Z	23	UBS	Bob
0xa123001	2016-03-02T12:00Z	1000	CS	Alice

[00155] Since contracts are not stored using fixed data schemas, but as data structures representing abstract syntax trees (ASTs), such a table cannot be provided by the underlying database implementation of the ledger. DAML™ embodiments provide the facility to query the ledger for active models with given parameters by means of syntactic pattern-matching of the AST representations of the active models in the ledger. This concept is probably best understood by looking at an example.

```
testQuery =
  scenario
    [ 'Bank1' commits (iouSellSettle 'Bank1' 'Alice' 100) ~> iou1
    , 'Bank1' commits (iouSellSettle 'Bank1' 'Bob'   20)  ~> iou2
    , 'Bank2' commits (iouSellSettle 'Bank2' 'Bob'   40)  ~> iou3
    , traceMatchingContracts (iouSellSettle ?obligor ?owner ?amount)
      -- This will output the following trace, where <iouN> refers to the
      -- value of the 'iouN' variable.
      --   Found 3 matching contracts:
      --   1. contract <iou1> with
      --      { "obligor": 'Bank1'
      --      , "owner": Alice
      --      , "amount": 100
      --      }
      --   2. contract <iou2> with
```

```
    --     { "obligor": 'Bank1'
    --     , "owner": Bob
    --     , "amount": 20
    --     }
    --   3. contract <iou3> with
    --     { "obligor": 'Bank2'
    --     , "owner": Bob
    --     , "amount": 40
    --     }
    -- DAML™ embodiments can also fix some values and thereby filter by equality.
  , traceMatchingContracts (iouSellSettle ?obligor 'Alice' ?amount)
    -- This will output the following trace.
    --   Found 1 matching contracts:
    --   1. contract <iou1> with
    --     { "obligor": 'Bank1'
    --     , "owner": Alice
    --     , "amount": 100
    --     }
  ];
```

[00156] DAML™ embodiments provide the function traceMatchingContracts with the syntax, traceMatchingContracts contractPatternExpr to run a query against the interpreter ledger in a scenario. A contractPatternExpr is an expression containing pattern variables. A pattern variable is distinguished from a normal variable by a leading '?' character. A simple example of a pattern is ?a + 2. This pattern would syntactically match 3 + 2 for ?a = 3, but this pattern would not match 3 + 9. The following table gives further examples:

Pattern	Expression	Match
(?a + ?b)	(1 + 2)	{"a": 1, "b": 2}
(?a + ?a)	(1 + 2)	{}
(?a + days ?b)	(1 + days 2)	{"a": 1, "b": 2}

```
(\a -> a + ?b)          (\x -> x + 2)    {"b": 2}
(\a -> a + ?a)          (\x -> x + 2)    {"a": 2}
```

[00157] DAML™ embodiments can create patterns by calling functions with pattern variable arguments. For example, iouSellSettle ?obligor ?owner ?amount is a pattern that matches models created using the iouSellSettle model template.

[00158] As the above example shows, the traceMatchingContracts accepts an expression of type Contract with pattern-variables. The resulting contract pattern is then matched against all active models and the results are reported as part of the interpretation. DAML™ embodiments do not yet expose the results in the Scenario result, as DAML™ embodiments cannot yet represent a list of results.

[00159] The two pattern variables ?ContractId, ?ChoiceTime are reserved pattern variable names and cannot be used in a model pattern. They match the contractIdBinder of a model and the choiceTimeBinder in a choice, respectively. If a model is parameterized by either of the two variables their values will be reported by the interpreter in the same way as a normal pattern variable.

[00160] Can all models in a ledger be pattern-matched? Given a concrete ledger one might ask whether all of its active models can be retrieved using a pattern-matching query. This is certainly true, as one can always query for each model itself using the model's AST without any variables as the query. So, one may reformulate the question: given a concrete DAML™ file defining the template for all models with which a company is interested in dealing, can all models that affect this company be pattern-matched using queries based on top-level definitions from this DAML™ file? This does not hold in general, as there can be nested await statements yielding active models for which there is no corresponding top-level definition. However, if DAML™

embodiments require that all awaits only occur in top-level definitions structured as follows, then all of the resulting models can be matched.

```
contractTemplateDef = \(param1 :: ty1) ... (paramN :: tyN) ->
  let abbrev1 = body1;
      ...
      abbrevM = bodyM;
  in await
   { ....
   }
```

[00161] This restriction does not limit expressivity. Alternate DAML™ embodiments may therefore investigate automatic translations of nested awaits into this form. Once DAML™ embodiments have this translation, DAML™ embodiments can always use pattern-matching to switch between a per-contract-template-tabular representation of models, and the native AST-based representation.

[00162] Alternate embodiment Proposed Language Extensions. The following section contain fully specified extensions not currently implemented in an exemplary execution model.

[00163] Must-choose obligations examples are parsed and executed. Many models require parties to make certain choices within a given time-bound. DAML™ embodiments can represent such obligations with the model language described up to here by using punitive clauses as shown in the 'mustPayIouWithin' example above. Adding all these punitive clauses becomes however rather cumbersome, which is why DAML™ embodiments add explicit syntactic and semantic support for must-choose obligations.

[00164] DAML™ embodiments explain the support for must-choose obligations in three steps. DAML™ embodiments first give an introductory example. Then, DAML™ embodiments specify the formal syntax of must-choose obligations; and finally DAML™ embodiments present

their semantics. The following contract models the obligation to execute the "sell" choice on another model until a certain maximal time.

```
mustPayUntil =
 \ (seller :: Party)
   (buyer  :: Party)
   (good   :: Contract)
   (maxSellTime :: Time)
 ->
   await
     { "pay":
       seller must choose cid :: ContractId until maxSellTime
       such that
         cid ~ good
       then
       { "payment": seller exercises "sell" with buyer on cid
       }

       -- DAML™ embodiments allow the buyer to forfeit a payment to illustrate in a later
       -- example that breached 'must choose' obligations lead to freezing
       -- the whole contract.
     , "forfeit": buyer chooses then {}
     }
;
```

[00165] These language elements are the must choose obligation and the until time-bound on the fulfillment of this obligation. The following scenario illustrates the use of the mustPayUntil contract template.

```
testSuccessfulSale =
  let iouFor = \(owner :: Party) -> iouSellSettle 'Bank' owner 100;
      aliceMustPayBobUntil = mustPayUntil 'Alice' 'Bob' (iouFor 'Alice');
```

```
in
  scenario
    [ -- create IOU's and payment obligations for 'Alice'
      pass (days 0) ~> t0
    , 'Bank' commits create (iouFor 'Alice') ~> aliceIou
    , 'Alice' commits create (aliceMustPayBobUntil (t0 + days 2)) ~> mustPay
      -- Alice is not obligable because she can choose the contract.
      -- However, the only kind of contracts she can choose are contracts, in
      -- which she becomes obligable. That makes her de facto obligable.
      -- demonstrate a successful fulfillment of a payment obligation
      -- after one day.
    , pass (days 1)
    , 'Alice' commits 'Alice' exercises "pay" with aliceIou on mustPay ~> {"payment":
bobIou}
    , assert (bobIou ~ iouFor 'Bob')
    ]
;
```

[00166] The formal syntax for choice-steps with support for both optional choices and must-choose obligations is:

```
 controllingPartyExpr1
  [chooses | must choose ]
     valueBinder1_1 :: Type1_1, ..., valueBinder1_L :: Type1_L
    at choiceStepTimeBinder1
    after t0
    until t1
   such that
    booleanChoiceStepCondExpr1
```

where t0 and t1 are expressions of type Time that do not reference any of the choice's bound values, and [chooses | must choose] means that one of the two keyword(s) on either side

of the | must be used. Both the after t0 and theuntil t1 constraints are optional for optional choices. For must-choose obligations only the after t0 constraint is optional.

[00167] DAML™ embodiments explain the semantics of the extended choice-steps in two parts. After t0 until t1 time constraints are evaluated. Then this embodiment defines when a 'must choose' obligation is breached and what the consequences of such a breach are.

[00168] Any choice or must-choose-obligation with an after tMin constraint can only be taken at a time t >= tMin. Any choice or must-choose-obligation ch with an until tMax constraint can only be taken at a time t < tMax. DAML™ embodiments call tMax the horizon of ch. DAML™ embodiments decided to interpret after tMin until tMax as a time-interval of the form { t | tMin <= t < tMax}to ensure that the intervals associated to two constraints of the form after t0 until t1 and after t1 until t2 are guaranteed to be disjoint. Thereby DAML™ embodiments simplify avoiding corner-cases where two choices are available at the same time. DAML™ embodiments decided to make the interval closed for its minimum time to ensure that a choice ch constrained by after 0 has the same behavior as the choice ch without the after 0 constraint.

[00169] A must-choose-obligation ch is breached at time t iff t is larger or equal to the horizon of ch. A model instance containing must-choose-obligations is breached at time t iff one of its must-choose-obligations is breached at time t. DAML™ embodiments freeze breached models by disabling all choices for all parties.

[00170] The motivation for completely freezing breached models is the following. DAML™ embodiments designed must-choose-obligations to be used in models where there is no default behavior for resolving breaches. If there was a default behavior, DAML™ embodiments could and would encode it by giving optional choices for handling the breach to the counter-parties. DAML™ embodiments expect that breaches of must-choose obligations in a model c are resolved by proposing a resolution model that asks all stakeholders of the model c for approval and then

deletes model c jointly with creating other models to compensate for the damages caused by the breach. The following example illustrates such a must-choose-obligation breach and its resolution.

```
testMustChooseObligationBreachResolution =
  let iouFor = \(owner :: Party) -> iouSellSettle 'Bank' owner 100;
      aliceMustPayBobUntil = mustPayUntil 'Alice' 'Bob' (iouFor 'Alice');
  in
    scenario
      [ -- create IOU's and payment obligations for 'Alice'
        pass (days 0) ~> t0
      , 'Bank' commits create (iouFor 'Alice') ~> aliceIouId
      , 'Alice' commits create (aliceMustPayBobUntil (t0 + days 2)) ~> mustPayId
        -- demonstrate that contracts are frozen as soon as one of their
        -- 'must choose' obligations is breached
      , pass (days 2)
        -- 'Alice' is too late with her payment.
      , mustFail ('Alice' exercises "pay" with aliceIouId on mustPayId)
        -- contracts with breached 'must choose' obligations are frozen
      , mustFail ('Bob' exercises "forfeit" on mustPayId)
        -- Demonstrate resolution of breached contract
        --
        -- Let's assume that 'Alice' and 'Bob' agreed out of ledger that
        -- 'Alice' can resolve the breached mustPayId contract by paying both
        -- the original payment and a 100% fine in two days after 'Bob's
        -- acceptance of the reparation proposal.
      , 'Alice' commits create (
          await
            { "accept": 'Bob' chooses at t then
              { "deleted": delete mustPayId (aliceMustPayBobUntil (t0 + days 2))
              , "payment1": aliceMustPayBobUntil (t + days 2)
              , "payment2": aliceMustPayBobUntil (t + days 2)
```

```
            }
          , "bob rejects": 'Bob' chooses then {}
          }
        ) ~> proposedResolution
      , 'Bob' commits 'Bob' exercises "accept" on proposedResolution
        ~> { "payment1": payment1, "payment2": payment2 }
       -- Alice immediately pays the first installment.
      , 'Alice' commits 'Alice' exercises "pay" with aliceIouId on payment1
      ]
  ;
```

[00171] Alternate DAML™ embodiments include extensions of the model language that DAML™ embodiments might want to add in alternate execution environments. They may include Descriptive Literals for Relative Time like '1d' to denote one day in relative time. Analogously, for other common units of measuring relative time. Mustache Templates for Legal Agreements: The explicit syntax for describing legal text is quite verbose. Provided is a lightweight alternative in the style ofMustache templates. For example, DAML™ embodiments may use an agreement as follows.

```
seller, buyer agree
  "{{seller} has mown the lawn of {{buyer}} between {{tbought}}
  and {{tbought + 1d}}, and if this was not the case, then {{buyer}} can sue
  {{seller}} according to the Swiss OR."
```

[00172] Public Choices explain the whenever some partyMakingChoiceBinder at choiceTimeBinder prefix for choices. DAML™ embodiments allow the definition of newtypes with a light-weight syntax that defines explicit conversion functions, and introduces lifted versions of all functions on the representation type. These newtypes are intended to reduce the number of opportunities for confusion in the positional calling convention.

```
newtype Amount = Integer
```

```
fromAmount :: Amount -> Integer
toAmount   :: Integer -> Amount
newtype Account = Text
fromAccount :: Account -> Text
toAccount :: Text -> Account
```

[00173] Type + Operator Extensions include a group of extensions that DAML™ embodiments envision to be useful, which can all be described as an additional set of types and functions using these types. DAML™ embodiments currently envision the following ones. Banking Calendar Support since banking models may depend heavily on trade-location-specific calendar conventions. DAML™ embodiments may want to provide custom operators with a well-defined semantics to capture these conventions. Times are usually in local exchange timezone. Many options / futures expire on things like 3rd Friday of month at 08:30. special treatment is required during business holidays where dates usually move before the holiday. Using an observable of something like businessDayOrBefore(t, "Europe/Zurich") might thus be useful.

[00174] Simple Observables are usable since many models need access to market data and other observable values of the physical reality where there is an objective definition of how they are measured. If the models are executed between participants sharing a common definition of observable values then an operator of the form obs :: Text -> Time -> Integer would be useful and sufficient to look up the values of the observable at specific times. Note that that this lookup would fail for times lying in the future. In case the model parties want to delegate the publication of observable values to third-parties, then DAML™ embodiments may use cryptographic receipts as described in the next section.

[00175] Cryptographic Receipts may be used to support binding of model evolution with off-ledger cryptographic statements; e.g., for certified market-data. Simple Key-based Receipts use signatures directly in the choice conditions. Policy-based Receipts are based on Proof-

Carrying Authorization, and allow for elaborate policies to be used to check receipts provided in choice conditions.

[00176] Turning to Figure 2, a rotatable 2x2x2 multi-faced puzzle cube is indicated generally by the reference numeral 200. For ease of description, this cube is offered as an example somewhat analogous to a smart model to represent the potential states of the model. The state of the cube is defined as the ordered colors of the four blocks showing on each of three visible faces. In an initial state, all four sections of each respective one of the cube's six faces are a respective one of six colors such as Red (R), White (W), Blue (B), Yellow (Y), Green (G), and Black (B). A first transaction transitions the cube through a top 90-degree clockwise rotation. A second transaction transitions the cube through a right 90- degree clockwise rotation.

[00177] Turning now to Figure 3, an example of representation of the states to be recorded for the cube of Figure 2 are indicated generally by the reference numeral 300. The initial state is {B, B, B, B, R, R, R, R, Y, Y, Y, Y, …}; the state after the top 90-degree clockwise rotation is {R, R, B, B, G, G, R, R, Y, Y, Y, Y, …}; and the state after the right 90- degree clockwise rotation is {R, W, B, W, R, G, R, G, Y, B, Y, R, …}.

[00178] As shown in Figure 4, the states of the cube as recorded to an append-only ledger in chronological order are indicated generally by the reference numeral 400. Here, the initial state of {B, B, B, B, R, R, R, R, Y, Y, Y, Y, …} is appended first; the second state of {R, R, B, B, G, G, R, R, Y, Y, Y, Y, …} is appended next; and the third state of {R, W, B, W, R, G, R, G, Y, B, Y, R, …} is appended last.

[00179] Turning to Figure 5, change semantics, security and authorization, measures for the append-only ledger are indicated generally by the reference numeral 500. Such measures may include marking a previously recorded state of the smart model as superseded, creating a

cryptographic hash and/or signature tied to each new state of the smart model; and using a Merkle tree or block-chain to ensure that each ledger committed to the ledger is built from a provable origin of references to previous ledger entries, follows verifiable commitments to previously defined validations rules, enables later sharing of hidden provable properties, and is tamper-proof.

[00180] Turning now to Figure 6 replicated copies of the ledger are indicated generally by the reference numeral 600. The replicated copies may be used to make the ledger robust to localized failures.

[00181] As shown in Figure 7, a ledger with approval and consensus distribution among the replicated copies of the ledger 600 is indicated generally by the reference numeral 700. Consensus facilitates maintenance of a single version of the distributed ledger.

[00182] Turning to Figure 8, the ledger 700 with the latest approved entry appended is indicated generally by the reference numeral 800. As shown, the ledger 800 includes apparent unoticed errors for the front face of the cube. This is not an acceptable state, and yet is not excluded without additional algorithms that validate the semantic correctness of the ledger entries. Such situations of error or inconsistent states inclusion in the ledger is not a current exemplary embodiment of the DAML™ ledger storage.

[00183] Turning now to Figure 9, the ledger with errors is indicated generally by the reference numeral 900. Here, the same error state, which was apparently noticed and committed, had become the distributed consensus on all nodes. This is not an acceptable state, and yet is not excluded without additional algorithms that validate the semantic correctness of the ledger entries. Such situations of error or inconsistent states inclusion in the ledger is not a current exemplary embodiment of the DAML™ ledger storage.

[00184] As shown in Figure 10, a ledger with validation of semantic correctness of ledger entries to avoid the entry of incorrect ledger entries indicated generally by the reference numeral 1000. Here, the validation against the higher rule or master model yielded an invalidity conclusion, and one or more parties blocks the ledger entries from entering the ledger because of this failed validation. In case of correct validation, the ledger entries can be committed with the inclusion of the "validating logic" within the ledgers log. The current exemplary embodiment of the DAML™ storage and ledger logic has one or more parties rejecting erroneous or semantically incorrect proposed ledger entry transaction. The current exemplary embodiment of the DAML™ storage and ledger logic has parties maintaining hash trees basic logic in the form of a blockchain allowing the full history of validation of each ledger transaction to be revisited and rechecked for auditing purpose.

[00185] Turning to Figure 11, a ledger where validation algorithms are define within the ledger, strengthening and extending the capabilities of the validation of ledger entries, is indicated generally by the reference numeral 1100. Here, a validation algorithm applied to each instance of the distributed ledger may depend on algorithms defined within previous ledger entries, and the resulting action and validation are applied. This makes validation stronger and extensible. The current exemplary embodiment of the DAML™ storage uses the formal property of the DAML™ to validate the later execution of DAML™ functions, uses previously stored DAML™ digital asset based ledger entries as a basis to validate DAML™ execution logic, and validate the include of new DAML-based ledger entries.

[00186] Turning now to Figure 12, an instance of DAML™ code is indicated generally by the reference numeral 1200. This DAML™ code defines variables of type color (an example party define defined type) for each of the four sections of each of the six faces (24 total), which will

indicate one of the six colors for each section. This code further includes a DAML™ "await" function, which awaits a party's choice of rotating either a top side or a right side, for example. Here, each choice produces a new "cube" while consuming an older one, and the definition of cube2 includes its validation rules.

[00187] As shown in Figure 13, an instance of DAML™ code is indicated generally by the reference numeral 1300. This code includes a DAML™ "await" function that awaits party1's choice ("chooses") "such that" if each side is a single color, "then" the parties "agree" that the "Cube is solved". Here, DAML™ allows parties to agree on statements and their interpretations external to the ledger.

[00188] Turning to Figure 14, an instance of DAML™ code is indicated generally by the reference numeral 1400. This code includes a DAML™ "await" function that awaits party1's choice ("chooses") "such that" if each side is a single color, "then" the parties "agree" that the "Cube is solved" and a prize is transferred ("transferAsset") from party2 to party1. Thus, DAML™ allows parties to operate on native fungible digital assets (e.g., assets in HyperLedger).

[00189] Turning now to Figure 15, an instance of DAML™ permitting alternating moves by two parties is indicated generally by the reference numeral 1500. This DAML™ two-party alternating moves example is different than that of Figure 12, in which only one party could choose one of the two shown moves.

[00190] As shown in Figure 16, an instance of DAML™ code combining multiple steps is indicated generally by the reference numeral 1600. This is a multiple-moves example with a "such that" condition that must be met in order for the validation to be accepted and allow the update to take place.

[00191] Turning to Figure 17, exemplary DAML™ code for an asset swap between two parties is indicated generally by the reference numeral 1700. This may be extended via recursion to a swap of cubes between multiple parties.

[00192] Thus, DAML™ ledger entry may be agreements, "await" (active models) waiting for parties to make one or more choices, and/or operations on native fungible assets. DAML™ agreements tie parties to "real world" data or “real world” activity and/or promises. DAML™ active models and operations on digital assets are both data and algorithmic in that they "contain" agreements that are data centric, and they allow active model exercise and asset operations that are algorithmic. Validation rules for an active model exercise are defined "within" each active model where they are like blockchain scripts, but in effect with a more sophisticated language, and new active models may create new validation rules.

[00193] Turning now to Figure 18, exemplary DAML™ code entry types are indicated generally by the reference numeral 1800. An active model, once exercised, always becomes an inactive model. That is, only one choice can be taken within a choice selection, and once taken the choice selection is no longer available.

[00194] As shown in Figure 19, exemplary DAML™ code choice is indicated generally by the reference numeral 1900. Composition allows each model step to be as sophisticated as desired. Here, in addition to the multiple moves await for "Top 180" of FIG. 16, another await for "Swap cubes" is presented. An update section swaps two cubes by creating active swapped cubes and deleting the previous cubes by making them inactive.

[00195] Turning to Figure 20, an exemplary DAML-based ledger with ordered ledger entries is indicated generally by the reference numeral 2000. A ledger is an ordered set of ledger entries shown here as X, Y, Z. The current exemplary embodiment of the DAML™ ledger storage

is append-only sequential ordering where a strict "one dimensional" order is maintained. Alternate embodiment of the DAML™ storage can be used to achieve better scaling by using a less strict and yet still logically correct sorting of ledger entries (e.g. through the use of direct acyclic graphs).

[00196] Turning now to Figure 21, another exemplary DAML-based ledger with ordered ledger entries is indicated generally by the reference numeral 2100. While the strict one-dimensional ordering of Figure 20 is sequential, local versus global ordering allows for other sequential ordering possibilities. Here, topological sorting or a direct acyclic graph (DAG) technique may be utilized for scaling to larger scenarios and more complex possibilities.

[00197] Turning to Figure 22, an exemplary DAML-based ledger with ordered and timestamped ledger entries is indicated generally by reference numeral 2200. This ledger contains an ordered and timestamped set of ledger entries shown here as X, Y, Z. The current exemplary embodiment of the DAML™ ledger storage is an append-only ledger where temporal information is maintained with each monotonically increasing timestamped ledger entry.

[00198] Turning to Figure 23, two exemplary DAML™ storage and ledger logic deployments across multiple parties are indicated generally by the reference numeral 2300. Here, one (right) has centralized logic as in Figure 22, but the other (left) has distributed logic. The centralized embodiment of the DAML™ storage and ledger logic is centralized around a key business party (e.g., an exchange), where parties can still maintain their own version of DAML™ storage and ledger logic. The distributed embodiment of a DAML™ storage and ledger logic deployment is distributed to support business processes that are not centralized around a key business party. This and like embodiments of the DAML™ storage can be used to achieve larger scaling by using weakly increasing timestamps (e.g., where unrelated ledger entries have no temporal dependencies).

[00199] Turning now to Figure 24, an exemplary DAML™ transaction with ledger entries, stored as data structures, and secrecy is indicated generally by the reference numeral 2400. Both centralized (left) and distributed (right) ledger embodiments are shown. Here, parties C and D cannot see certain ledger entries, while parties A and B can. The ledger visible to a party may be effectively masked, using known cryptographic techniques, for example, such that the ledger transaction shows ledger entries visible only to authorized parties, such as to support secrecy. Here, a ledger transaction may include multiple ledger entries added as a "group" where transaction logic ensures "all or nothing" execution logic for the group. Secrecy and partial visibility are provided since not all parties involved in a transaction need or are authorized to see all ledger entries of that transaction, as a transaction may include ledger entries that are private to a subset of transaction participants. Transaction and entry insert/append authorization is supported where multiple parties need to authorize the transaction for it to be entered or committed.

[00200] Turning now to Figure 25, an exemplary DAML™ transaction with ledger entries and secrecy is indicated generally by the reference numeral 2500. Here, party B cannot see certain ledger entries, while party A can. The ledger visible to a party may be effectively masked such that the ledger transaction shows ledger entries visible only to authorized parties, such as to support secrecy. Here, a ledger transaction may include multiple ledger entries added as a "group" where transaction logic ensures "all or nothing" execution logic for the group. Secrecy and partial visibility are provided since not all parties involved in a transaction need or are authorized to see all ledger entries of that transaction, as a transaction may include ledger entries that are private to a subset of transaction participants. Transaction and entry insert/append authorization is supported where multiple parties need to authorize the transaction for it to be entered or committed.

[00201] As shown in Figure 26 an exemplary DAML-based ledger including multi-party authorization of commitment of new ledger entry transactions is indicated generally by the reference numeral 2600. Here, authorization is provided by the validating an affected party. Authorization from one or more validating party may guarantee that all new ledger entries committed to the ledger meet proper and previously agreed upon ledger logic. Authorization from the affected parties may be to commit to agree to future business variants and scenarios defined within ledger entries. That is, one or more parties consents to immediate and possible future ledger execution impacts of the proposed DAML-based ledger transaction. Likewise, one or more partiesies consents to immediate and possible future impact on cross-party relations of the proposed DAML-based ledger transaction. The current exemplary embodiment of the DAML™ ledger has one or more parties consenting to DAML™ ledger execution logic, and has one or more parties consenting to changes to legal rights and obligations between two or more parties following the interpretation of the proposed DAML™ ledger transaction within the context of previously agreed external legal agreements. Alternate embodiments of the DAML™ ledger may have the authorization certified by legal authorities to allow a DAML™ ledger transaction to define legally binding relations between two or more parties without the necessity to work within a previously external agreement framework. Alternate embodiments of the DAML™ ledger may have the authorization certified by multiple legal authorities to allow a DAML™ ledger transaction to define legally binding relations between two or more parties across two or more legal jurisdictions. The current exemplary embodiment of the DAML™ ledger has one or more parties validating the correctness of the DAML™ ledger execution logic.

[00202] As shown in Figure 27, an exemplary DAML-based two-tiered ledger including hash versus entry and details is indicated generally by the reference numeral 2700. Here, all parties

have the anonymized transactions and hash values, but party B does not have some details that party A has. The hash keys are tied to ledger entries. A hash is associated with each ledger entry, but the hashes do not reveal ledger entry details. Hashes help to provide proof of ledger entry details without revealing them. A two-tiered ledger includes a ledger or log of hash data, and a ledger of ledger entries. Both ledger tiers include a transaction structure and/or an optional block structure.

[00203] As shown in Figure 28, an exemplary DAML-based hash centric public ledger tier or public log is indicated generally by the reference numeral 2800. Here, the public ledge provides provably authorized references. This public ledger is a ledger of "hashes" that hides ledger entry data. The hash of each ledger entry is the anonymized reference to that ledger entry. This provides the transaction commitment status, where the parties authorize the transaction based on their private ledger, but the transaction commit is in the public ledger. The private ledger commit depends on the public ledger commit. This particular embodiment implements only hashes, and has no finite life cycle nor cross-entry dependencies. That is, the public ledger is a log where the entries are hash values associated with detailed ledger entries stored in private ledgers, and the public ledger entries are anonymized references to the detailed private entries. A ledger transaction commitment is authorized by one or more parties within the public ledger, and this public ledger commitment status determines the commitment status of the transactions of the private ledger. The current exemplary embodiment of the DAML™ ledger does not include any life cycle of its entries (the hash values), nor cross-entry dependencies, and can therefore be seen as a log. Alternate embodiments of the DAML™ ledger may include additional logic, such as homomorphic encryption of certain ledger details, and implement further ledger logic.

[00204] As shown in Figure 29, an exemplary DAML-based private and sharable private ledger tier is indicated generally by the reference numeral 2900. Here, the private ledger includes an authorized private and sharable private ledger. The ledger entry data may be shared on a "need to know basis." Privacy is maintained since parties do not receive, nor see, nor store ledger entries to which they are not "participants". The private and public ledgers are linked through a hash of ledger entries. Authorization and transaction commitment apply to both, including "Merkle tree based proofs" on Merkle trees that are built and shared by a party (e.g. validating party), securely distributed, and encrypted; where the need to know basis may be used by a party with a partial view of the private ledger.

[00205] The private ledger provides certifications or “proofs of correctness” of properties associable to the public ledger to other parties on a need-to-know basis based on the relation of the requesting party to the requested private ledger properties. The current exemplary embodiment of the certification and proof scheme of public ledger entries through private data properties is based on a Merkle signature scheme. Alternative embodiments of the certification and proof scheme are possible, for example based on PKI and a proof carrying scheme.

[00206] As shown in Figure 30, an exemplary two tier DAML™ ledger with associated ledger entry identification and ledger entry liveliness tracking is indicated generally by the reference numeral 3000. The current exemplary embodiment of these two ledger entry properties are built upon the hash structures that reside within the private ledger tier, where the hash tree/graph is used to associate unique and relevant cryptographic hash values with ledger entries while capturing the acyclic dependency between the DAML™ (and possible non-DAML) based private ledger entries, while incrementally updating a Merkle tree structure is used to track the liveliness (active versus inactive status) of each ledger entry. Alternative embodiments of the

ledger entry identification and ledger entry liveliness tracking may target scalability by addressing the non-uniform nature of ledger entry data via capturing properties of the physical partitioning model within the hash value assignment logic and within the liveliness tracking structure.

[00207] As shown in Figure 31, an exemplary two tier DAML™ ledger with block-oriented logic, and with associated ledger entry identification and block-centric liveliness tracking is indicated generally by the reference numeral 3100. The current exemplary embodiment to track ledger entry liveliness is to regroup ledger transactions as blocks of multiple transactions, and incrementally update the private ledger Merkle tree with each new block of transactions. Alternative embodiments of block-oriented ledger entry liveliness tracking may target scalability by addressing the non-uniform nature of ledger entry data via capturing properties of the physical partitioning model within the hash value assignment logic and within the block-oriented liveliness tracking structure.

[00208] As shown in Figure 32, an exemplary party to party sharing of certification or provable properties of private ledger entry properties is indicated generally by the reference numeral 3200. The current exemplary embodiment to share certification or provable properties of private ledger entry properties is to provide on a need to know basis Merkle signature scheme based certification of specific ledger entries, optionally supported by partial deeper private ledger data tied to the ledger entry hash value. Here, authorization of ledger liveliness is based on a "Merkle signature."

[00209] Turning to Figure 33, hosted and non-hosted copies of a distributed ledger are indicated generally by the reference numeral 3300. Here, public versions are in the left column and private versions are in the right column. Hosted copies of both the public and private versions are in the top row for parties A1, A2, etc., while non-hosted copies are in the bottom two rows for

party B and party C, respectively. That is, party ledgers may be hosted, or they may be managed by the respective parties.

[00210] Turning now to Figure 34, exemplary DAML™ ledger entry types for Agree, Await, and Delete commands are indicated generally by the reference numeral 3400. There are the three main types of DAML™ ledger entries. Here, Agree generally means that one or more parties agree on a real world change, or a real world promise among specified parties. Await generally means that execution awaits a permit or promise later or in the future of a "predefined" change to the ledger. Delete generally means that a previously entered active model is inactivated (such as deactivate/annihilate/invalidate). Ledger entries may also include native cryptographic assets defined by the supporting ledger. These are separate ledger entries which can be both referenced and created by DAML™ ledger entries of type await (see Figure 14). In operation, an Await may act as an offer, while a Chooses and/or Agree may act as an acceptance.

[00211] As shown in Figure 35, exemplary usage of the DAML™ Agree command is indicated generally by the reference numeral 3500. Here, the syntax and semantics are fully up to the parties to agree "off ledger", but all parties to an agreement need to authorize it. In greater detail, Agree means that one or more parties agree on a real world change. The syntax, etc. is fully up to the parties to agree "off ledger". All parties to an agreement need to authorize it. Under some circumstances, such agreement or active model may embody a legal court-enforceable contract if the authorization had legal standing and other lawful requirements were met. For example, a master contract may be used to give such an agreement legal status. Otherwise, master contracts may impart other features to agreements. An agreement or active model history cannot actually be deleted, yet an agreement may supersede a previous one and thereby inactivate the

previous one. Thus, the function Delete makes a model inactive, but need not remove its history. "Ledger logic" does not automatically know the scope of real world effects.

[00212] Turning to Figure 36, ledger entries are "instantiated" entries indicated generally by the reference numeral 3600. Here, the code in the left column represents a non-instantiated model template for a cube, the middle column represents data values to be instantiated into the template, and the right column represents an instantiated active model including the data values.

[00213] Turning now to Figure 37, exemplary DAML™ code of an agreement for external notification within an equity model, where the parties agree that they are making and receiving notification of an event in the format for that event, is indicated generally by the reference numeral 3700. In alternate embodiments, agreement may be used in many other DAML™ applications.

[00214] As shown in Figure 38, exemplary DAML™ code using the Await command within a cube model is indicated generally by the reference numeral 3800. Here, Await waits for a permit or promise of a later/future "predefined and pre-authorized" change to the ledger. These are the logic "choices" that are offered by an await entry. Each choice produces a new "cube" or active model while consuming an older one making that an inactive model. In this example, the definition of the cube2 model is also its validation ruleset.

[00215] Turning to Figure 39, a DAML™ ledger with delete commands is indicated generally by the reference numeral 3900. The delete command is used to transform an active model into an inactive model such as by applying deactivate/annihilate/invalidate to any previous active entry. Here, the earlier (upper) delete references an await, while the later (lower) delete references an agree.

[00216] A bundle or block of transactions is executed as a single step, all or nothing, including the following three parts.

1. Transactions within a bundle are created, defined or derived to contain:
    a. A set of one or more of:
        i. Deactivation of previous ledger entries;
        ii. Transient entries that are created and deactivated within the same transaction; and/or
        iii. Creation of new ledger entries.
    b. For each entry:
        i. Which parties need to authorize it (affected parties);
        ii. Which parties see this entry (to which party is the entry visible); and/or
        iii. Justification/proof of reasoning for b(i) and b(ii) as needed to support the soundness of each party's validation checking.
2. Get authorization from affected parties for content.
3. Commit transaction to ledger with authorizations.

[00217] In the first part, a transaction and/or ledger entry may be proposed and/or created by a single party, or jointly proposed and/or created by multiple parties. A transaction may be communicated as a data-centric set of transaction entries, or as a DAML™ script that generates transaction entries with optional supporting salt codes. Each transaction entry can be independent of the past and unrelated to previous ledger entries, or it can be dependent on one or more previous ledger entries. Proposing parties provide the basis of the validation of their proposed ledger transaction if this was not previously visible to affected parties. For example, entries having no dependency on the past do not require reference to prior entries for justification.

[00218] Turning now to Figures 40 and 41, a ledger algorithm with role-centric flows is indicated generally by the reference numerals 4000 and 4100. The ledger algorithm is distributed and repeats the following operational steps, in sequence or optionally in parallel.

[00219] At step and/or node 1, a ledger transaction is initiated. The ledger transaction proposal is created, or optionally derived from interpreting existing ledger entries. A party creates a "ledger entry update" that specifies the expected transaction. Optionally, more than one party

can agree together on creating a ledger entry update. The impact on the ledger caused by the ledger entry update, which may also be referred to as the "Core Transaction", is computed by the party's DAML™ engine system.

[00220] At step and/or node 2, the proposed transaction may include outputs, inputs, authorizations, and proofs/justification. The outputs may include a set of one or more new ledger entries. There may be any number of types of ledger entries, but the principal ones are destruction of ledger entries (DAML™ delete), and creation of new ledger entries such as DAML™ agreements or DAML™ awaits with choice logic. The inputs may include a set of zero or more references to existing active ledger entries. For each transaction ledger entry, authorization information may include which parties need to authorize it within this transactions (affected or validating parties), which parties see this entry (to which party is the entry visible), optionally which parties are needed to authorize the derived action, and supporting Merkle signatures tied to ledger entries needed to justify the transaction.

[00221] At step and/or node 3, the ledger transaction is authorized. Here, requests for authorization are created and sent to concerned parties. The requests are created and only sent to parties from whom authorization is needed, and each request contains a copy of transaction ledger entries, excluding those for which the party has no visibility.

[00222] At step and/or node 4, the ledger transaction is coordinated. Digital signatures from the authorization parties are provided. These are generated per ledger transaction or per Merkle signature tied to ledger entries within a Merkle tree of the ledger transaction or possibly within the Merkle tree of a block of transactions. Each party checks correctness of the transaction, ledger entries and visible Merkle tree hashes before providing their authorization.

[00223] At step and/or node 5, the ledger transaction is stored. Each party involved in transactions stores new ledger entries in the private tier of their ledger, but without yet committing these to private storage.

[00224] At step and/or node 6, the ledger transaction is committed. The shared public tier of the ledger with a distributed consensus algorithm (e.g., HyperLedger) stores and commits the new transaction with its Merkle tree hashes and authorizations but without transaction entry details.

[00225] At step and/or node 7, the ledger transaction is committed to private tier of the ledger storage. Each party involved in a new transaction can commit its previously stored transaction entry details in the public tier of the ledger.

[00226] The current exemplary embodiment of the DAML™ ledger has a dedicated party that validates each new DAML™ ledger entry transaction proposal (e.g., an exchange entity). The current exemplary embodiment of the DAML™ ledger has a new DAML™ ledger entry transaction being proposed by one party and has the proposing party and an additional dedicated party consenting to the impact of the new transaction on the ledger and on relations with other parties. Multi-party consent is done by successive entry of DAML™ ledger transactions by different parties in such a manner as to build DAML™ logic combining the multiple parties' consents. Alternate embodiments of the DAML™ ledger can allow any number of parties to consent to the impact of a new DAML™ transaction on the ledger and on relations with other parties.

[00227] Turning to Figure 42, a party-centric ledger algorithm flow is indicated generally by the reference numeral 4200. Here, function block 4210 initiates the ledger, including a private shared tier 4212 of the ledger and a public tier 4214 of the ledger, and passes control to decision

block 4220. Decision block 4220 waits for party action, and then passes control to function blocks 4230 and/or 4240. Function block 4230 initiates a ledger transaction on the private shared ledger 4212, and passes control back to the decision block 4220 to await further party action. Function block 4240 authorizes the ledger transaction on the public ledger 4214, and passes control back to the decision block 4220 to await further party action.

[00228] Turning now to Figure 43, a ledger-centric ledger algorithm is indicated generally by the reference numeral 4300. Here, function block 4310 initiates the ledger, including a private shared tier 4312 of the ledger and a public tier 4314 of the ledger, and passes control to a decision block 4320. The decision block 4320 passes control to one or more of function blocks 4330, 4340, 4350, 4360, and/or 4370 based on the ledger event, which each implement their particular function and pass control back to the decision block 4320 to await the next ledger event. Here, the function block 4330 authorizes the ledger transaction; the function block 4340 coordinates the ledger transaction; the function block 4350 stores the ledger in the private shared ledger 4312; the function block 4360 commits the ledger transaction to the public tier 4314 of parties ledger; and/or the function block 4370 commits the storage state to the private tier of the parties ledger.

[00229] As shown in Figure 44, an exemplary function to initiate a ledger transaction is indicated generally by the reference numeral 4400. At input block 4410, a party initiates a new DAML™ update expression based on a business intent message (BIT), and passes control to a function block 4420. The function block 4420 uses the DAML™ engine to compute the impact of the proposed DAML™ update on the ledger in the form of a core transaction (CT) 4412, and passes control to a function block 4430. The function block 4430 initiates and coordinates the core transaction on the ledger. The core transaction 4412 enumerates expected changes, and provides hooks to digitally sign ledger entries. The basic proposed transaction details may include new

ledger entries (outputs that will become active models), and ledger entries to be removed (inputs that will become inactive models). In addition, each transaction specifies which party must authorize each ledger entry, and which party can "see" each ledger entry (to which party are the ledger entries visible), as well as justification in the form of a cryptographic hash of the BIT, and supporting Merkle proofs.

[00230] A DAML™ engine may further include a DAML™ simulator that provides the ability to observe and interact with DAML™ Modeling activities. DAML™ is a new type of contract specification language (CSL). Users may inspect the developed DAML™ code by running the scenarios and observing how the DAML™ models are used to progress in a scenario. They may use this visibility to observe how their input is absorbed into the DAML-based ledger, for example.

[00231] The DAML™ simulator may be deployed together with the DAML™ engine onto a host accessible to the user. The DAML™ simulator may be deployed as a binary, with the DAML™ Modeling code as source code. This source code may be visible in the simulator. Thus, a user may view the Modeling of their business processes within DAML, but may not modify this installation. The DAML™ simulator includes a User Interface (UI) portion. Access to the UI may be limited via HTTP, HTTPS, or the like.

[00232] In operation, the DAML-based ledger properties assure process continuity between successive ledger operations. They are defined as fixed, and pre-agreed upon among the parties, but they can also be managed incrementally. In the current exemplary embodiment they are implemented with a "ledger of core primitives", and is then evolved as with a version management by one or more key parties (e.g., an exchange entity), allowing backwards compatibility of ledger semantics. Although variable semantics are contemplated, exemplary embodiments use fixed

semantics for existing ledger entries because new versions of these rules might impose on parties the onus to migrate ledger entries from a previous version to a new version. New version function semantics can be introduced for future ledger entries as long as they maintain existing semantics of older ledger entries. Such new version functions may be tracked by version or date, for example. Note, however, that exemplary embodiments may depend on external legal agreements that may be subject to change, and should therefore be tracked by revision date.

[00233] Parties agree on ledger entries and their representation as data. Ledger entries define what parties can agree through the media of the ledger. These are defined with lexical, grammar, binary, and other representations. Exemplary embodiments of DAML™ may use agree, await, delete, transfer and other HyperLedger™-compatible digital asset ledger primitives. Semantics of previous and future ledger entry interpretation, and real world agreements. Ledger entry interpretation semantics define a deterministic way to algorithmically interpret or evaluate a ledger entry. Semantics of a ledger entry may affect previous and future ledger entries.

[00234] In DAML, the "delete" entry invalidates or deactivates previous ledger entries in DAML, but the auditable history remains. Interpretation of the execution of an “await choice” provides the "rights" to delete the "chosen await entry", and the right to create zero or more new ledger entries that meet the constraints supporting a ledger entry update.

[00235] In agreements, semantics of ledger entry may affect future ledger entries. In DAML, an "await" ledger entry authorizes a later “choice” to create ledger entries that match given properties. Future interpretation may depend on future ledger entry data. In DAML, await choice selection configurations may include templates that optionally depend upon zero or more parameters to be provided in one or more future ledger entries.

[00236] The semantics of a ledger entry may affect the physical world. In DAML, an "agree" ledger entry ties multiple parties to a "real world" agreement. Semantics of ledger entry update declaration define a deterministic way to algorithmically create new ledger entries from a ledger entry update declaration. In DAML, an update statement and generalized follow-up have such semantics.

[00237] Authorization rules may be used in an algorithm that computes which parties need to agree to the creation of a new ledger entry. These take into account effects of a ledger entry being derived from the interpretation of one or more existing ledger entries, or whether the ledger entry is created independently of the past. This takes into account that continuity in the “face of secrecy" can be assured by designating trusted parties as validating authorization parties and the use of Merkle signatures on Merkle trees of active ledger entries..

[00238] Semantics of validating parties are enforced. Ledger entry interpretation semantics can define validating parties with the purpose of assuring "no double trade" by assuring that the correctness of ledger logic and the "continuity" of the Merkle tree based logic. A central concept is that validating parties may be defined within DAML. In an exemplary embodiment, that also means that if party were to become a validating party, while not having been previously, and if this party were not to have enough ledger details to assure validation, it would need to be receiving the additional ledger details from another party, and be able to show the consistency of its new view on the ledger within the instituted secrecy rule, or it would not be permitted to authorize the proposed transaction.

[00239] Semantics and rules of secrecy are used in an algorithm that computes which parties may see a new ledger entry. This takes into account authorization needs of ledger entry. Moreover, salt and Merkle tree implementation rules define a deterministic way to compute one or more hash

keys of a ledger entry or of a ledger entry update. These rules will produce the same hash given the same ledger entry and salt value, and the same sequence of hashes given the same ledger entry update and same sequence of salt values. One hash per ledger entry is defined by the update.

[00240] Hashes with cryptographic properties do not allow the hashes to be used to derive information on the ledger entries that they derive from. For example, DAML™ engine implementation is not compromised by hashes usable only for confirmation of validity.

[00241] In a transaction, ledger entries are added as a "group" with "all succeed" or "all fail" atomicity properties. That is the notion of a DAML™ ledger transaction. Multiple parties need to authorize the transaction for it to be entered (committed), but not all parties involved in a transaction need to see all all ledger entries of the transaction. This is because a transaction may include ledger entries that are private to a subset of transaction participants. While parties are agreeing on transactions, they are technically authorizing ledger entries within the transaction. Thus, their digital signatures are tied to individual ledger entries. The "transaction" logic further ensures "all or nothing" execution logic.

[00242] Using DAML, there are three types of ledger entries. The delete type can be said to deactivate/annihilate/invalidate a previous entry by transitioning it to an inactive model. The await type embodies a permit or promise for later or future "predefined" change(s) to the ledger. The agree type requires that multiple parties each agree to one or more steps, conditions, or results.

[00243] An await entry offers the "choices" that the authorizing parties "bind themselves to allow consequences to the choices of the await." An agree entry requires that one or more parties agree on a real world change, and although its result can be deactivated, the agreement itself cannot be deleted. "Ledger logic" does not know scope, and therefore it would be indeterminable if such scope could be "deleted".

[00244] By default, all parties to an agreement need to authorize it. The agreement might supersede a previous one. An agreement is typically "eventful", but is not required to be. The syntax and the interpretation of an agreement is left entirely up to the parties to agree "off ledger". An exemplary embodiment ledger records such off ledger agreements, but does not attempt to interpret them. Under particular circumstances, such an agreement leading to an active model may meet the requirements of a legally enforceable contract in a given jurisdiction if that was the intention of the parties and their respective authorizations had legal standing. In general, the ledger does not care whether a given agreement is legally enforceable, and an exemplary embodiment makes no distinction between a general agreement and one meeting the standards of a legally enforceable contract. Where desired, the present inventive concept envisions that a master contract may be used to give DAML™ agreements legal status as contracts in particular jurisdictions.

[00245] All ledger entries, code, declarative statements, data structures and the like discussed above can be stored in a non-transient computer readable storage media. Functional steps described herein can be accomplished by computer code executed on a processor. The various data manipulations described above can be accomplished on stored data structures to create transformed data structures that are processed by a computer processor in a different manner. The various functions, such as the await function, of the embodiments allow a computing system to operate in a new manner to accomplish transactions and provide advantages not found in the prior art. The various flow chart steps can be accomplished by software modules executed on a computer processor. The cylinders illustrated in the drawing can represent data structures, such as databases storing records, which are manipulated in the described manner to allow a computing system to operate on the data and transform the data.

[00246] An example of a digital system 5000 that performs one or more of the computer implemented methods described herein is illustrated in Figure 45. The system 5000 may facilitate modelling and/or recording of a digital asset and its evolution, including the associated rights of a plurality of parties. As an example, the plurality of parties may include first user 5001 and second user 5003. The first user 5001 is associated with a first node 5002 and the second user 5003 is associated with a second node 5004. The first 5002 and the second node 5004 may be in communication with a third node 5007. This allows messages, notifications, indications, choices, authorization, etc. from the first and second nodes (and operated by respective users) to the third node 5007.

[00247] The first node 5002, second node 5004 and third node 5007 may be an electronic device, such as a computer, tablet computer, mobile communication device, computer server, computer terminal, etc. Such an electronic device may include a processing device, a data store and a user interface. Examples of a user interface include a keyboard, mouse, monitor, touchscreen display, etc.

[00248] The third node 5007, having a processing device 5008, is in communication with at least one storage device 5009. In some examples, the storage device 5009 is a data store 5011, such as a hard disk, magnetic disk, solid state storage, for an append-only ledger. In some examples, the storage device for an append-only ledger is in the form of a peer-to-peer distributed ledger 5013. The peer-to-peer distributed ledger 5013 may be associated with one or more processing devices 5015 to receive and record data. In some examples the peer-to-peer distributed ledger 5013 includes a blockchain.

[00249] The system may also include another user 5017 who may be one of the parties involved. The another user 5017 may use a delegate, such as second user 5003 (and the corresponding second node 5004) to interact with the system 5000.

[00250] It is to be appreciated that the first node 5002, second node 5004, third node 5007, and data storage 5009 may be in communication with each other through one or more networks that may include one or more of a local area network, wide area network. In some examples, this may include a secure local area network. In some examples, the communication may include a communications network that includes the internet.

[00251] Figure 46 illustrates an example of a method 100 of manipulating data structures to model and/or record a digital asset and its evolution. At least part of this method may be performed at the third node 5007. This includes providing 1200 an await function instance that executes no more than once using at least once using one of at least one choice defined therein for disposition of the digital asset with respect to the rights of at least one of the plurality of parties, said await function instance incorporated upon the consent of the affected parties to fulfil a configured function instance associated with the ate least one choice.

[00252] In some examples, this step 1200 may be initiated upon receiving indication(s) of consent that is sent 5100 from the nodes 5002, 5003 associated with the parties 5001, 5017.

[00253] The method 100 may also include providing 1300 an agree function instance that requires the consent of at least one of the plurality of parties to execute. In some examples, this step 1300 may be initiated upon receiving indication(s) of consent associated with the agree function that is sent 5100 from the nodes 5002, 5003 associated with the parties 5001, 5017.

[00254] The method 100 may also include providing access 4000 to an append-only ledger for storing results of the executed function instances. In turn, the at least one storage device 5009 may store 5200 the results in an append-only ledger.

[00255] In some examples, the system 5000 may perform a method of interpreting a modelled digital asset and its evolution with respect to the rights of a plurality of parties. This may include executing an await function instance of the await function 1200 described above and executing an agree function instance of the agree function 1300 described above. The results of these executed function instances may then be stored in the append-only ledger.

[00256] Another example of a method 6000 performed by the system 5000 will now be described with reference to Figure 47. At least part of this method 6000 may be performed at the third node 5007. This shows a computer implemented method of manipulating data structures to model and/or record a digital asset and its evolution, including the associated rights of a plurality of parties. The method 6000 includes determining 6100 an await function that includes at least one choice defined therein for disposition of the digital asset, wherein the at least one choice has an associated configured function and a respective choice-step condition.

[00257] The method 6000 further includes executing 6200 the await function instance no more than once, wherein upon receiving an indication of consent of parties required by the at least one choice and receiving a valid choice selection from the at least one choice, the method further comprises executing the associated configured function instance of the at least one choice, wherein the await function terminates with determination that the choice-step condition of the valid choice selection is satisfied.

[00258] It is to be appreciated that either one and/or both steps 6100 and 6200 may be initiated from receiving an indication of consent of a party or parties required by the at least one

choice and/or receiving a valid choice selection that is sent 6250 by the nodes 5002, 5004 associated with the parties 5001, 5003. In some examples, the indication of consent may include, either explicitly or implicitly, the choice selection (and vice versa).

[00259] The method 6000 further includes determining an agree function to create an agree record associated with the digital asset, wherein the method further comprises executing an agree function instance based on receiving consent of parties required for the agreement. This agree function may be initiated by receiving an indication of consent required for the agreement that is sent 6350 from the nodes 5002, 5004.

[00260] The method 6000 further includes sending 6400, to an append-only ledger for storage, results of the executed function instances. In turn, the storage device 5009 stores 6450 the results in the append-only ledger.

[00261] Some further examples of the methods 100, 6000 may further comprise determining 6700 a delete function that, when executed, requires the consent of the affected parties to invalidate an agree function or disable a non-executed await function. This step 6700 may be in response to receiving an indication of consent of the affected parties to invalidate an agree function or disable a non-executed await function that is sent 6750 from the nodes 5002, 5003 associated with the parties 5001, 5017. The method further includes sending 6800, to an append-only ledger for storage, the results of the executed await, agree, and delete functions. In turn, the at least one storage device 5009 may store the results in the append-only ledger.

[00262] The above method and system may be applied to a variety of digital assets. In some examples, such digital assets may be representative or associated with tangible assets (e.g. iron ore, wheat, coffee, gold, etc.). For example, modern exchanges for highly traded commodities, such as fungible commodities are represented as a digital asset. As an example, gold can be traded

digitally between parties using (digital) tokens without physical exchange of the gold itself. In other examples, the digital asset may be representative of other rights, such as a right for payment in a negotiable instrument (such as a banknote). Similar to traditional banknotes, that may incorporate anti-counterfeiting solutions such as seals, watermarks, magnetic strip, digital assets also require security features to maintain the integrity of the digital asset. This may include incorporating means to prevent double spending and keeping an immutable record that can be used for auditing.

[00263] A non-limiting example of how digital asset can be modelled and evolved will now be described. In this example, the digital asset will include associated rights (which may be in terms and conditions of the relationship between parties). In some examples, some of these terms and agreements may be (at least partially) standardized or in a set of agreements between parties. In a trading situation, for example, a master agreement might detail the relationship between a dealer and its custodian, spelling out when and how the custodian will act and describing the responsibilities of each. In healthcare, the master agreement could be a risk sharing agreement between a hospital provider and the outpatient care vendors. The operation of the master agreement may, in part, be facilitated by the method (e.g. the evolution of the digital asset).

[00264] The relationship between the parties, in some examples, may be considered a contract (in the sense that there is an offer and acceptance of the parties to an agreement that can include terms and conditions). Since there may be a large volume of digital asset and that some digital assets may be similar or share similar characteristics to each other, it may be beneficial to include templates that include, at least in part, information (such as the terms and conditions) relevant to the digital asset. A contract template contains the agreement statement, all the applicable parameters, and the choices that can be made in acting on that data. It specifies

acceptable input and the resulting output. Therefore it describes the change of state that will occur when the contract is implemented. A contract template is not in effect until it is signed by the parties involved. A contract template contains placeholders rather than actual names, amounts, dates and so on.

[00265] Figure 49 illustrates a schematic diagram representing a contract template 7001. One party (such as owner 7003) has some currency. The owner 7003 can make a choice to:

i. Deposit the money in a bank 7007 (first choice 7011); or

ii. Sell it to a second party 7005 (presumably in exchange for goods or services) (second choice 7013).

[00266] In the first choice 7011, the money goes into the bank account at the bank 7007 and in the second choice 7013, the money goes to the second party 7005. As noted above, the contract template may contain placeholders rather than actual names, amounts, dates, etc. as these may be determined at a later stage when parties provide their choice and/or consent. This allows the contract template to be used multiple times (and/or modified as appropriate). It is to be appreciated that contract templates may be stored at a library in a data store for retrieval. In some examples, the contract templates may formalize or define at least part of the rights of the parties (if they enter into the contract as discussed below).

[00267] An active contract may be an instance of the contract where the placeholders have been replaced with actual data. It has been accepted and is in effect and binding on at least one of the participants. An active contract may be a modified and/or executed example of the contract template described above.

[00268] Figure 50 illustrates a schematic diagram of an example of an active contract 7101, where the owner 7003 details have been replaced with a specific person, Alice 7103. Similarly,

the second party 7005 is specified as Bob 7105. The bank 7007 may be defined in the contracts template or a specific bank may be defined in the active contract 7101. Thus in the active contract, Alice has the money and she has two choices:

i. Deposit the money in a bank 7007 (first choice 7011); or

ii. Sell it to Bob 7105 (second choice 7013).

[00269] In the method described above, this state of an active contract may be provided by the await function (or more specifically an instance of the await function). This is illustrated in Figure 51 that shows the await function instance 7201 that includes two choices 7011, 7013 for Alice 7103.

[00270] If Alice 7103 chooses the first choice 7011 (that is, the "Payout" option), then the code determines what happens next is the execution of an associated configured function 7211. Alice's choice and the bank's agreement to deposit money in Alice's account (either by previous agreement when opening Alice's bank account or a temporal agreement) may be indicative of consent of the parties to Alice's choice. In this example, the associated configured function is for the bank to deposit money in Alice's account. A respective choice-step condition may include the condition of receiving confirmation that the bank 7007 has made a deposit to Alice's account.

[00271] If Alice 7103 chooses the second choice 7013 (that is, the "Sell" option), then the code determines that a new instance 7223 of a contract is created based on the contract template, but this time, Bob is the one with the currency and the option to deposit and sell. That is, the associated configured function for the second choice 7013 is to create the new instance of a contract 7223. The respective choice-step condition may include an indication that the new instance 7223 of a contract is created. Alternatively, or in combination, another respective choice-

step condition may include Bob 7105 having made a valid choice or that the Bank 7007 having deposited money to Bob's account.

[00272] When the contract is used and a choice has been made, it is no longer available for use. That is, a user cannot sell the same asset again or deposit the same currency twice. This may be achieved by the await function terminating with the determination that the choice-step condition of the valid choice is satisfied.

[00273] The choice-step conditions may allow parallel choice-steps whilst preventing double spending. Importantly, this may allow subsequent parallel choices to be validly selected if an earlier choice-step condition of an earlier choice step is not or cannot be satisfied or fulfilled. For example, say Alice 7103 initially chooses the first choice 7011 but the bank 7007 is unable to deposit the money in Alice's account (for example, Alice's account has been closed). Thus Alice has made a choice but the choice step condition cannot be satisfied. Thus this remains an active contract (and performed through an active await function) so that Alice can make an alternative choice such as the second choice 7013. This prevents a situation where Alice is stuck with a choice that cannot be fulfilled.

[00274] Once a contract has be "used", the await function instance 7201 is terminated (i.e. when a valid choice is made and the respective choice-step condition is satisfied). Thus that used contract becomes an inactive contract and can no longer be acted on (i.e. Alice cannot make a choice again). This prevents double spending. A feature of the system is that items are not deleted, so that the contract (or information stored on the ledger associated with contract) is not deleted from append-only ledger. This maintains the integrity of the system and may assist review and auditing. Figure 52 illustrates an inactive contract 7101' (that corresponds to a terminated await function instance) where Alice had made a valid choice 7013 and therefore can no longer make

any other choices.. The result of this is that Bob can make a choice in a new await function instance 7223, such as to sell to Charlie 7106 or to deposit to a bank 7007.

[00275] However, there are some circumstances where one or more of the defined choices are not desirable or cannot be chosen. This may include circumstances where it is necessary to record relevant information for a digital asset in relation to an event that was not anticipated or could be modelled. This can be "real world" events that affect the digital asset. Thus the system also includes steps to allow real world events to be recorded.

[00276] This may include an agree function (or instances thereof) to allow an agree record associated with the digital asset to be recorded on the append-only ledger. The agree function may be executed upon receiving consent of the parties required for the agreement. For example, the parties required may be those parties who's rights are affected or at stake.

[00277] In some examples, the agree function may be associated with an agree function instance and/or the choice-step condition. For example, part of a choice may include performing a "real world" or "off ledger" action, and performance of this may require the agree function instance to record this. With reference to the above example, when the bank deposits money into Alice's or Bob's account, this may result in a deposit slip or receipt for that payment outside the system. Thus Alice 7103 and the bank 7007, or Bob 7105 and the bank 7007 may need to provide indication that they both agree the deposit was made, and this indication may be used as consent to record this as an agree record on the ledger. In other examples where the digital asset is representative of tangible goods, the agree function may be used to record the delivery of goods, such as both parties agreeing that a quantity of coal was delivered.

[00278] A delete function instance (and delete function instance) may be used to disable or invalidate a previous entry recorded on the append-only ledger with the consent of the affected

parties. This function, in conjunction with the agree function, may allow the system some flexibility to correct errors without deleting information from the append-only ledger record.

[00279] The above method and system may allow users to code for a specific contract or contract templates and execute the contracts to see the results of various choices. This may allow the user to drill-down to see the various contract options (i.e. choices, which may in turn lead to further contracts and additional choices). This allows a user to, in effect, go down any path, interacting with the modelled contracts in any valid way and seeing the results. In some examples, this may also allow simulation of contracts of one or more users and the interactions. Thus in some examples, the system and method may be adapted to be an analysis tool.

[00280] Figure 53 illustrates an example of a processing device 5008 that may be associated with the nodes 5002, 5004, 5007. The processing device 5008 includes a processor 9310, a memory 9320 and an interface device 9340 that communicate with each other via a bus 9330. The memory 9320 stores instructions and data for implementing the methods 100, 3900, 6000 described above, and the processor 9310 performs the instructions (such as a computer program) from the memory 9320 to implement the methods. The interface device 9340 may include a communications module that facilitates communication with the communications network and, in some examples, with the user interface and peripherals such as data store 5009 as well as other nodes. It should be noted that although the processing device 5008 may be independent network elements, the processing device 5008 may also be part of another network element. Further, some functions performed by the processing device 5008 may be distributed between multiple network elements. For example, multiple processing devices 5008 may perform the method in a secure local area network.

[00281] While the inventive concept has been described by way of example with respect to non-limiting exemplary embodiments; other alternatives, modifications, and variations will be apparent to those of ordinary skill in the pertinent art based on the teachings disclosed herein. Accordingly, the scope of the appended claims is intended to include all such alternatives, modifications and variations on the exemplary embodiments set forth herein, as well as equivalents thereof that fall within the scope and spirit of the present disclosure.

CLAIMS

1. A computer-implemented method of authorizing a modeled digital asset and its evolution with respect to rights of a plurality of parties, the method comprising:

determining the rights of the plurality of parties with respect to the digital asset, wherein the digital asset is disposable through execution of a first function instance, the first function instance including at least a first choice defined therein for disposition of the digital asset;

receiving, by a computer associated with a first authorizing party, a proposed transaction configured to update an append-only ledger maintained by a plurality of distinct nodes of a computer network, the proposed transaction including one or more proposed ledger entries resulting from the execution of the first function instance, wherein the computer is a private node entitled to receive information pertaining to the proposed transaction;

receiving, by the computer, an authorization request from a requesting node,

receiving, by the computer, consent of the first authorizing party to authorize immediate and future ledger execution impacts of the proposed transaction;

digitally signing, by the computer after receiving the consent of the first authorizing party, the at least one of the one or more proposed ledger entries to provide authorization of the at least one of the one or more proposed ledger entries;

sending, by the computer, the authorization of the at least one of the one or more proposed ledger entries to the requesting node to allow execution of the first function and storing of the results of the executed first function as a committed transaction in an updated append-only ledger; and

receiving at least part of the updated append-only ledger.

2. The computer-implemented method of claim 1 wherein the proposed transaction includes per-ledger-entry authorizing data that identifies, for each of the one or more proposed ledger entries, one or more ledger-entry authorizing nodes that must authorize the respective proposed ledger entry.

3. The computer-implemented method of claim 2, further comprising validating that each of the one or more proposed ledger entries is authorized by each ledger-entry authorizing node required to authorize the respective proposed ledger entry.

4. The computer implemented method of claim 2, further comprising digitally-signing only the proposed ledger entries required for the computer according to the per-ledger-entry authorizing data.

5. The computer implemented method of claim 1, wherein the append-only ledger includes a public ledger shared amongst the plurality of nodes of the computer network, the public ledger storing a cryptographic representation of the committed transaction.

6. The computer-implemented method of claim 5, wherein the cryptographic representation of the committed transaction comprises a hash of the committed transaction.

7. The computer-implemented method of claim 1, wherein the proposed transaction, when committed, creates a new active model of the digital asset recorded to the private shared ledger.

8. The computer implemented method of claim 1, further comprising:

determining a private subset of the plurality of nodes entitled to receive the proposed transaction and the digitally signed ledger entries; and

transmitting a readable-version of the proposed transaction and the digitally-signed ledger entries only to the private nodes.

9. The computer implemented method of claim 8, further comprising:

validating that an authorizing subset of the plurality of nodes have authorized the immediate and future ledger execution impacts of the proposed transaction;

executing the first function instance after the validating; and

storing results of the executed first function instance as a committed transaction in a private shared ledger replicated by each of the private nodes.

10. A system for authorizing a modeled digital asset and its evolution with respect to rights of a plurality of parties, the system, associated with a first authorizing party of the plurality of parties, comprising:

one or more processors; and

memory storing instructions that when executed by the one or more processors, cause the one or more processors to:

determine the rights of the plurality of parties with respect to the digital asset, wherein the digital asset is disposable through execution of a first function instance, the

first function instance including at least a first choice defined therein for disposition of the digital asset;

receive a proposed transaction configured to update an append-only ledger maintained by a plurality of distinct nodes of a computer network, the proposed transaction including one or more proposed ledger entries resulting from the execution of the first function instance, wherein the system is a private node entitled to receive information pertaining to the proposed transaction;

receive an authorization request from a requesting node;

receive consent of the first authorizing party to authorize immediate and future ledger execution impacts of the proposed transaction;

digitally sign, after receiving the consent of the first authorizing party, the at least one of the one or more proposed ledger entries to provide authorization of the at least one of the one or more proposed ledger entries;

send the authorization of the at least one of the one or more proposed ledger entries to the requesting node to allow execution of the first function and storing of the results of the executed first function as a committed transaction in an updated append-only ledger; and

receive at least part of the updated append-only ledger.

11. The system of claim 10 wherein the proposed transaction includes per-ledger-entry authorizing data that identifies, for each of the one or more proposed ledger entries, one or more ledger-entry authorizing nodes that must authorize the respective proposed ledger entry.

12. The system of claim 11, wherein the instructions further cause the one or more processors to:

validate that each of the one or more proposed ledger entries is authorized by each ledger-entry authorizing node required to authorize the respective ledger entry.

13. The system of claim 11, wherein the instructions further cause the one or more processors to:

digitally-sign only the ledger entries required for the computer according to the per-ledger-entry authorizing data.

14. The system of claim 10, wherein the append-only ledger includes a public ledger shared amongst the plurality of nodes of the computer network, the public ledger storing a cryptographic representation of the committed transaction.

15. The system of claim 14, wherein the cryptographic representation of the committed transaction comprises a hash of the committed transaction.

16. The system of claim 10, wherein the proposed transaction, when committed, creates a new active model of the digital asset recorded to the private shared ledger.

17. The system of claim 10, wherein the instructions further cause the one or more processors to:

determine a private subset of the plurality of nodes entitled to receive the proposed transaction and the digitally signed ledger entries; and

transmit a readable-version of the proposed transaction and the digitally-signed ledger entries only to the private nodes.

18. The system of claim 17, wherein the instructions further cause the one or more processors to:

validate that an authorizing subset of the plurality of nodes have authorized the immediate and future ledger execution impacts of the proposed transaction;

execute the first function instance after the validating; and

storing results of the executed first function instance as a committed transaction in a private shared ledger replicated by each of the private nodes.

19. A non-transitory computer-readable medium storing instructions for authorizing a modeled digital asset and its evolution with respect to rights of a plurality of parties, the instructions, when executed by the one or more processors, associated with a first authorizing party of the plurality of parties, cause the one or more processors to:

determine the rights of the plurality of parties with respect to the digital asset, wherein the digital asset is disposable through execution of a first function instance, the first function instance including at least a first choice defined therein for disposition of the digital asset;

receive a proposed transaction configured to update an append-only ledger maintained by a plurality of distinct nodes of a computer network, the proposed transaction including one or more proposed ledger entries resulting from the execution of the first function instance, wherein the one

or more processors are part of a private node entitled to receive information pertaining to the proposed transaction;

receive an authorization request from a requesting node;

receive consent of the first authorizing party to authorize immediate and future ledger execution impacts of the proposed transaction;

digitally sign, after receiving the consent of the first authorizing party, the at least one of the one or more proposed ledger entries to provide authorization of the ledger entries;

send the authorization of the at least one of the one or more proposed ledger entries to the requesting node to allow execution of the first function and storing of the results of the executed first function as a committed transaction in an updated append-only ledger; and

receive at least part of the updated append-only ledger.

20. The non-transitory computer-readable medium of claim 19 wherein the proposed transaction includes per-ledger-entry authorizing data that identifies, for each of the one or more proposed ledger entries, one or more ledger-entry authorizing nodes that must authorize the respective proposed ledger entry.