Socotra Delopment

Applied Contract Theory

– Concepts and Practice –

October 12, 2021

Socotra

Contents

Acronyms

AuxData Auxiliary Data: A set of key value pairs which can be connected to a insurance related object in the system.

Part I
Concepts

We discuss, at a high level, how we can structure insurance contracts and what kind of properties those contracts should have. Then we try to apply these ideas, also at a high level, to some practical, actual problems.

Chapter 1
Theory

Science is a means whereby learning is achieved, not by mere theoretical speculation on the one hand, nor by the undirected accumulation of practical facts on the other, but rather by a motivated iteration between theory and practice.

Abstract Before we begin the process of discussing actual software, we consider how we might structure a financial contract in the insurance industry, what might be the fundamental operations on those contracts, and what properties those operations should satisfy.

1.1 Contracts

Consider an insurance contract, such as medical insurance. If we wanted describe such contracts generally, in code, we might come up with a simple descriptive framework like that below:

```
type Name = String

data Interval = Interval Integer Integer

data Contract =
    SimpleContract Name Interval (M.Map String String) |
    CompoundContract Contract Contract |
    And Contract Contract |
    Zero
```

There would be a standard $Name$ that would designate the particular insurance product offered by the insurance company. In addition the contract would need to have an associated $Interval$ to define the date that the contract coverage starts and the date that coverage ends. This interval would be a closed open interval with integers representing the start and end times as unix timestamps. Since we are trying to be general we would also need a way to describe attributes of policies. For medical insurance, these attributes could be numerical like co-pay amounts or deductible amounts, but they could also be non numerical attributes like a customers primary care physician or information on family members covered by the contract. All of this information I collect as a map from strings to strings in $SimpleContract$ above. I might also want to structure contracts in some way. For instance, my medical

insurance might be issued by an insurance company as two distinct contracts, one for domestic and one for international medical insurance. I could describe this contract as the contract $And\ domestic\ international$. We can get still fancier in how we structure contracts. Socotra currently likes to structure its contracts like trees. The most general attributes and time intervals are specified at the root of the contract tree and the details of the contract get more specific as one moves towards the leaves. Here is an example of a empty contract structured in the Socotra style:

```
peril :: Contract
peril = SimpleContract "collision" (Interval 1 100) M.empty

exposure :: Contract
exposure = SimpleContract "vehicle" (Interval 1 100) M.empty

policy :: Contract
policy = SimpleContract "auto" (Interval 1 100) M.empty

example :: Contract
example = CompoundContract policy (
            CompoundContract exposure (
                CompoundContract (And peril peril) Zero))
```

Don't worry about the contract nomenclature of policies, exposures, and perils right now, just note that the constructor $CompoundContract$ is sufficient to support any structuring need, tree or otherwise. Lastly, I have added a $Zero$ contract; this is the minimal contract that a customer can have and is conceptually the contract that all customers start off with and from which we build more complicated contracts.

1.2 Charges

In pricing of a contract it's necessary to have a way of delineating all of the charges associated with the contract. We can define this idea as

```
type Refundable = Bool

data Units =
    Currency |
    CurrencyPerMonth |
    CurrencyPerYear |
    CurrencyPerInstallment

data Charge=
    Premium Interval Double Units Refundable |
    Tax Interval Double Units Refundable |
    Commission Interval Double Units Refundable |
    Fee Interval Double Units Refundable
```

We will want to describe the origin of each charge, whether it is a premium, tax, commission or other fee. For each charge we will also need to track the interval over which the charge is assessed, some numerical currency value with its units, and lastly

a boolean value indicating whether the charge is refundable or not. At the moment the framework does not track what the numerical amount of a charge represents in units or the refundability of the charge. So the definitions above are more complete than in practice. That does not mean the concepts are optional however. Tracking the units and refundability of charges will eventually be required to properly price policies over the full range of possible charges.

1.3 Observables

With contract structure defined and a basic idea of charges we can start to compute observables. Observables are any external, objective values which are of interest to both parties to the contract. In a functional language observables will be calculated by evaluating recursively down the structure of the contract using pattern matching. In an object oriented scenario one would accomplish the same objective using a object or functional based visitor pattern. At Socotra with a Java backend, the second approach is very roughly the one that is taken.

One important observable is $rate$ which is defined as

```
rate :: Contract -> [Charge]
```

which takes a contract and calculates all of the charges associated with the contract. At socotra the $rate$ function takes the form of a function written in liquid or a plugin written in JavaScript. Other observables are

```
underwrite :: Contract -> (Bool, String)
```

to provide an underwriting decision and a possible rejection reason on the contract. Then there is $invoice$

```
invoice :: Contract -> [Invoice]
```

which takes a contract and greedily generates all of the invoices that will be payable on the contract. The greedy behavior of the $invoice$ function may seem contrived but Socotra actually generates invoice information greedily and, as we will see later, we will use this behavior as a convenience in describing the necessary properties of $invoice$ functions in general.

Before, we get there though, given what we have so far, we can make a few statements on the properties that we can expect from out observable functions. Given contracts, c1, c2, c3, we expect that observables will be commutative and associative

$$Obs(c1 \text{`}And\text{`} c2) = Obs(c2 \text{`}And\text{`} c1)$$
$$Obs(c1 \text{`}And\text{`} c2) \oplus Obs(c3) = Obs(c1) \oplus Obs(c2 \text{`}And\text{`} c3)$$

We also want all observations to comply with the meaning of the $Zero$ contract

$$Obs(Zero \text{`}And\text{`} c) = Obs(c) = Obs(c \text{`}And\text{`} Zero)$$

1.4 Modifications

Once a policy has been created, it will continue to undergo modifications over its lifetime. Managing these modifications is a large part of what the Socotra API does. There are three base functions which comprise any modification. These are:

```
extend :: Interval -> Contract -> Contract

reduce :: Interval -> Contract -> Contract

override :: Interval ->  Map String String -> Contract ->
    Contract
```

$extend$ moves the upper bound of the contract interval into the future maintaining the existing attributes of the contract. In the Socotra API, this extension is, in code and in conversation, variously referred to as one of the functions $create$, $renew$, or $reinstate$. $reduce$ moves the upper bound of the contract interval to a new value, t', such that $contract.start \leq t' < contract.end$. In the Socotra API, this function is variously referred to as one of the functions $cancel$ or $lapse$. Lastly, the $override$ function should be roughly associated with the Socotra concept of an endorsement. The meaning of $override$ is very close to that of relational override in mathematics. Here is an example of its use:

$$\{make : GM, value : 5000\}$$
$$`override`\{make : Ford, model : F150, value : null\}$$
$$= \{make : Ford, model : F150\}$$

This example contains an update, an addition, and a deletion. And you will find that this simple example carries over to endorsement code that you will eventually see, although in the actually code base the updates, addition, and deletes are much more explicit.

You may rightly wonder, especially if you are familiar with the Socotra framework, why I have abstracted this way over existing modifications like create, renew, endorse, cancel, and reinstate. I have done this point out the compositional nature of modifications which are currently implemented as one off behaviors. Take for instance a contract from t_1 to t_3 which we plan on endorsing with added attributes, while also changing the end date to some $t_2 < t_3$. This operation can be, and is, expressed monolithicly and non reusably in the framework as:

$$endorse\ (Interval\ t_1\ t_3)\ policy_{t_1}^{t_3}\ fields\ \{newEnd = t_2\}$$

or it can be expressed compositionally as

$$reduce\ (Interval\ t_2\ t_3)\ \$\ override\ (Interval\ t_1\ t_3)\ fields\ policy_{t_1}^{t_3}$$

There is all the difference in the world between writing monolithic functions for every modification and variation there of, and writing three functions that achieve the same effect through functional composition.

Now that we have our basic modification combinators, I am going to use them to define which behaviors must be true for our code to be correct. There is no particular order to the properties they are merely ideas which are important or non obvious. I will state the properties in Socotra terminology and write the details in generalized terminology. That way the idea will not get lost in abstraction.

Cancellation reinstatement inversion Under an observation, a contract that is canceled followed by a full reinstatement must equal the original contract.

$$extend\ (Interval\ t_2\ t_3)\ \$reduce\ (Interval\ t_2\ t_3)\ c = c$$

If we designate the left hand side of the equation as c'. Then we certainly want $rate\ c' = rate\ c$ Invoicing is slightly more complicated and it turns out we would like to following to be true:

$$\sum\{i.amount | i \in invoicec' if\ i.status \neq writtenOff\} = \sum\{i.amount | i \in invoicec\}$$

After a cancellation followed by a reinstatement, the amount of all invoices that have not been written off must be equal to the amount of the invoices that originally existed.

Invoice invariance on cancellation When a partial cancellation happens on a policy, all invoiced periods that are fully within the remaining, uncanceled range must not incur any modifications to their total amount or component amounts.

$$is = invoice\ contract_{t_1}^{t_3}$$
$$is' = invoice\ \$\ reduce\ (Interval\ t_2\ t_3)\ contract_{t_1}^{t_3}$$
$$\{i \in is : within\ i\ (Interval\ t_1\ t_2)\} = \{i' \in is' : within\ i' (Interval\ t_1\ t_2)\}$$

This should make intuitive sense. If one has a monthly policy and its is canceled after one month, barring cancellation charges, the amount calculated and owed for the first month should not change.

Endorsement commutativity Under observation, renewals and reinstatements commute with simple endorsements

$$\begin{aligned} c_{t_1}^{t_4} &= extend\ (Interval\ t_3\ t_4)\ \$\ override\ (Interval\ t_2\ t_3)\ fields\ c_{t_1}^{t_3} \\ &= override\ (Interval\ t_2\ t_4)\ fields\ \$\ extend\ (Interval\ t_3\ t_4)\ c_{t_1}^{t_3} \end{aligned}$$

Additionally, two endorsements will commute when the field maps of the two endorsements are disjoint or when the endorsement intervals are disjoint.

One can get more involved in creating more intricate equalities, but from the simple examples the main points should be clear. If a contract has a content and particular start and end dates. The modification path that got it to those start and end

dates should, in most cases, not effect observables calculated on the contract. If a contract is reduced, in most cases, observables related to the remaining part of the policy should not be affected.

1.5 Intervals

You may have noted that contracts, charges, and modification functions all use intervals in their definition. As such, intervals are unsurprisingly a pervasive concept throughout framework objects as well as the database. There interval end points are commonly expressed with the variable names $start\_timestamp$ and $end\_timestamp$, and, in the framework, intervals and their operations are expressed with the $Duration$ and $Durations$ classes. You should be familiar with the common interval predicates, like $overlaps()$ and $contains()$, and also the standard interval operations, like $union()$, and $intersect()$. Below are some of the predicates and operations that you will find can simplify your work and clarify your code.

MODULE $IntervalOps$

EXTENDS $Integers$, $Reals$

$max(a,\ b) \triangleq$ IF $a \geq b$ THEN a ELSE b
$min(a,\ b) \triangleq$ IF $a \leq b$ THEN a ELSE b

Interval Predicates

Making decisions on intervals is facilitated by a fundamental set of predicate combinators over intervals. The fundamental interval combinators sufficient for all interval computations at *Socotra* are listed below. Most of them are implemented in the framework Duration class. As others predicates become necessary that class is the best place to add new ones.

$notOverlaps(chA,\ chB) \triangleq\ chB.end\_ts \leq chA.start\_ts$
$\quad \lor\ chB.start\_ts \geq chA.end\_ts$

$overlaps(chA,\ chB) \triangleq\ \neg notOverlaps(chA,\ chB)$

$starts(chA,\ chB) \triangleq\ chA.start\_ts = chB.start\_ts \land chA.end\_ts < chB.end\_ts$

$contains(chA,\ chB) \triangleq\ chA.start\_ts \leq chB.start\_ts \land chB.end\_ts \leq chA.end\_ts$

$equals(chA,\ chB) \triangleq$
$\quad \land\ chA.start\_ts = chB.start\_ts$
$\quad \land\ chA.end\_ts = chB.end\_ts$

$width(ch) \triangleq\ ch.end\_ts - ch.start\_ts$

$within(tm,\ ch) \triangleq\ ch.start\_ts < tm \land tm < ch.end\_ts$

RECURSIVE $overlapsAny(\_,\ \_)$
$overlapsAny(ch,\ chs) \triangleq$
$\quad$ IF $chs = \{\}$ THEN FALSE

```
    ELSE LET tmp ≜ CHOOSE x ∈ chs : TRUE
         IN IF overlaps(ch, tmp) THEN TRUE
            ELSE overlapsAny(ch, chs \ {tmp})

RECURSIVE startsAny(_, _)
startsAny(ch, chs) ≜
  IF chs = {} THEN FALSE
  ELSE LET tmp ≜ CHOOSE x ∈ chs : TRUE
       IN IF starts(ch, tmp) THEN TRUE
          ELSE startsAny(ch, chs \ {tmp})
```

Interval Functions

There are three fundamental inverval functions used commonly in the framework and specified below.

(1) subtract: Interval → Interval → Set Interval, returns the parts of the first interval that do not overlap with the second interval. And will be a set of cardinality between 0 and 2.

(2) union: Interval → *Inteval* → Set Interval, returns the combination of the first and second intervals as long as the intervals overlap. And will be a set of cardinality between 0 and 1.

(3) intersect: Interval → Interval → Set Interval, returns the part of the first and second interval that overlap.

The fundamental interval functions are implemented in the framework's Durations class along with some additional common interval functions.

```
subtract(chA, chB) ≜
  CASE notOverlaps(chA, chB) → {chA}
  □OTHER →
    IF within(chB.start_ts, chA) THEN
      IF within(chB.end_ts, chA) THEN {
          [chA EXCEPT !.start_ts = chA.start_ts, !.end_ts = chB.start_ts],
          [chA EXCEPT !.start_ts = chB.end_ts, !.end_ts = chA.end_ts]
          }
       ELSE
          {[chA EXCEPT !.start_ts = chA.start_ts, !.end_ts = chB.start_ts]}
     ELSE
      IF within(chB.end_ts, chA) THEN
          {[chA EXCEPT !.start_ts = chB.end_ts, !.end_ts = chA.end_ts]}
       ELSE
          {}

union(chA, chB) ≜
  IF overlaps(chA, chB) ∨ chA.end_ts = chB.start_ts ∨ chB.end_ts = chA.start_ts
   THEN {[chA EXCEPT !.start_ts = min(chA.start_ts, chB.start_ts),
                     !.end_ts = max(chA.end_ts, chB.end_ts)]}
  ELSE {}

intersect(chA, chB) ≜
  IF notOverlaps(chA, chB)
```

```
    THEN {}
    ELSE {[chA EXCEPT !.start_ts = max(chA.start_ts, chB.start_ts),
                      !.end_ts = min(chA.end_ts, chB.end_ts)
          ]}

extend(chA, chB) ≜
    [chA EXCEPT !.start_ts = min(chA.start_ts, chB.start_ts),
                !.end_ts = max(chA.end_ts, chB.end_ts)]

subtractAll(chs, chB) ≜ UNION {subtract(chA, chB) : chA ∈ chs}

intersectAll(chs, chB) ≜ UNION {intersect(chA, chB) : chA ∈ chs}

overlapsAll(chs, int) ≜ {ch ∈ chs : overlaps(ch, int)}
```

* Modification History
* Last modified *Mon Oct* 11 10:40:53 *PDT* 2021 by *marco*
* Last modified *Wed Jul* 14 21:11:27 *PDT* 2021 by *ASUS*
* Created *Mon Dec* 28 09:50:41 *PST* 2020 by *ASUS*

Chapter 2
Practice

As a practitioner, you want the simplest possible strategy; the one that has the smallest amount of side effects; the minimum possible hidden complications . . . Your strategy to survive isn't the same as ability to impress colleagues.

Abstract In the previous chapter we discussed contracts at a very abstract level. Here we move from the previous abstract discussion closer to the specific realization of those abstractions in the framework.

2.1 The Simplified Schema

In the previous chapter we described a contract as the combination of a product name, an interval, and a set of attributes. In the actual Socotra implementation, contracts are called policies. Each policy has an associated interval and a set of attributes, which are called characteristics. The idea of a product name is expanded into the concept of a product. This product is best thought of as metadata describing the policy. You can think of a product as a parallel to the familiar idea of metadata in the database world. In the database world metadata is a set of relation and field definitions. Records in the database are then instances of those relations and field definitions. In the Socotra object model the product is the definition of possible attributes (characteristics) and product behaviors. A policy is then, like a database record, a specific data instance that complies with the definitions set out in the product (metadata).
In the framework, policies are contracts that are instances of what I have called a $CompoundContract$ in the previous chapter. In particular Socotra has decided to organize policy data in the form of a three level tree. The root of the tree is called the policy. The nodes at second level of the tree are called exposures and the nodes at the third level of the tree are called perils. Attributes become more specialized as one moves down the tree towards the perils. The span of an interval at any node in the tree is always contained within the interval of its parent. And the interval of the policy, at the root, defines the total time span of the contract itself. [1]

[1] Trees are useful data structures to assist a computer in sorting and searching. As technologists we are proud to have trained ourselves to be comfortable thinking in and manipulating trees. But must users are not computers nor have they been trained to think in tree structures. In the real world, it is a classic design failure to expose users to trees and I have seen that design choice lose to simpler

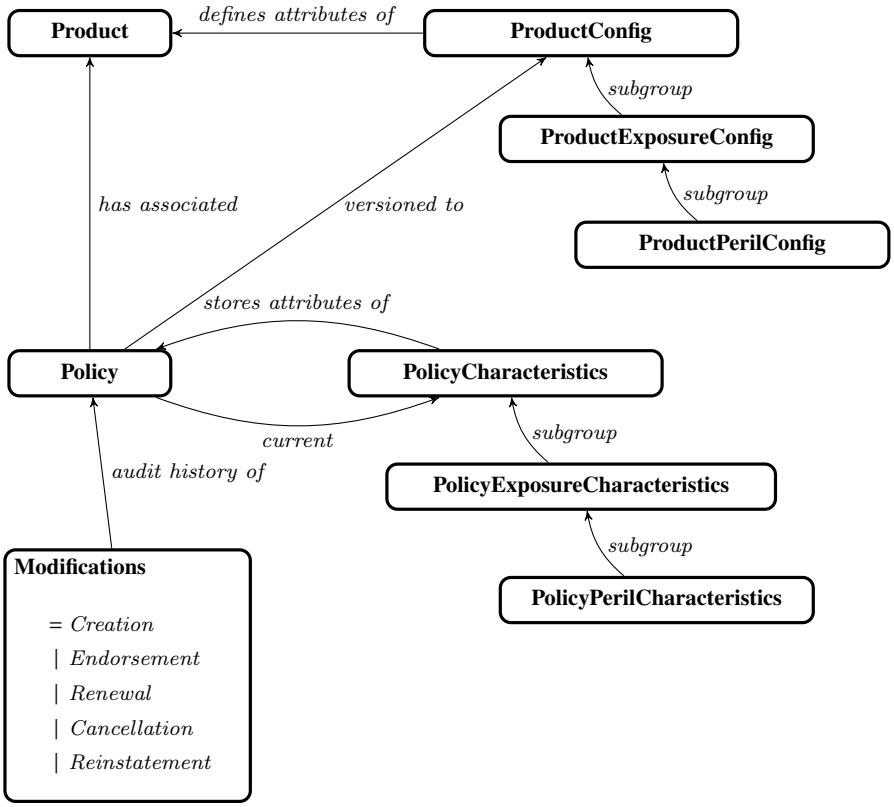

Fig. 2.1 A simple, overview of the Socotra data schema

Lastly, in addition to policy and product data, the framework organizes it's data to track change history. You remember from the previous chapter that the operations $extend$, $reduce$, and $override$ can be applied to contracts. The applications of these functions are tracked in the frameworks policy_modification table. In the policy_modification table the first change to a policy is tracked as a create modification and further modifications are tracked in order from there. This whole organization of data with labeled relationships on the data is shown in stylized form in image 2.1.

designs in the market over and over again. Trees are always the wrong data structure to expose to non technical users. There are no exceptions to this rule. Store that knowledge in your future design toolbox.

2.2 Policies

Now that we have talked a bit about policies and modifications, below is a data structure overview of the same. The mathematical notation my not be that familiar to you and if so don't be too put off. As with all mathematical notation in this book, there are comments to go with the notation and the comments cover all of the important points. If you are interested you should be able to go from the comments to the notation and that might be useful where you feel there are ambiguities. The mathematics is always correct and exists specifically to remove any ambiguity.

MODULE $Policy$

EXTENDS $Integers$, $Product$, $PolicyCtx$

CONSTANT $NoModification$

Socotra has several ways of generating invoices. It can generate invoices when a modification is accepted, on a schedule like once a month, and, for premium reporting, on client request. To give a sense that there is some variety, I have defined a incomplete set of the billing policies below.

$BillingPolicy \triangleq$ {"BySystemOnFinalize", "ByClientOnRequest"}

It is useful to see policies has having just two states. They start off in the $OffRisk$ state and transition to $OnRisk$ with their first issued modifications. Policies can also transition back to $OffRisk$ when they are fully cancelled.

$PolicyState \triangleq$ {"OnRisk", "OffRisk"}

The linear lifecycle of a $Modification$ from start to issue is:

(a) Draft - also commonly referred to as Create

(b) Accepted - also commonly referred to as $Finalized$

(c) Issued.

There are a few other states which pop up with modifications as well but are less often seen. Expired is particular to reinstatements which have an associated lifetime. If the reinstatement is not issued before its expiration. Then it expires. Invalidation moves a modification from the accepted state back into the draft state. And rescind completely discards a modification.

$ModificationState \triangleq$ {"Created", "Finalized", "Issued", "Expired", "Invalidated", "Rescind"}

$TimeRange \triangleq policyMinTs \,..\, policyMaxTs$

There are more modification types than are listed below. These are just three that will represent the extend, reduce, override possibilities.

$ModificationType \triangleq$ {"Endorsement", "Renewal", "Cancellation"}

Modifcations all have the basic structure below. It is convinient to think of modifications as a elaborated time interval and the action of applying a modification to a policy as an interval extension or reduction. Note that in the database the $policy\_modification$ table does not include an $end\_timestamp$. This absence turns out to be very problematical as in many places in the framework it is a crucial piece of information too have handy. Maybe this data mishap will get fixed in the future.

$Modification \triangleq$ [
  $type : ModificationType$,

$state : ModificationState,$
$start\_timestamp : TimeRange,$
$end\_timestamp : TimeRange,$
$product\_revision : 0 \,..\, maxRevision$
]

$InvModification(mod) \triangleq$
$\land\ mod.start\_timestamp \leq mod.end\_timestamp$

In retrospect, the simple policy representation below is not all that good for a discussion of policies, so here are some of facets of policy data to be familiar with. $original\_start\_timestamp$ will be the start date when the policy issued, and will be $\leq$ to the $effective\_start\_timestamp$. The $effective\_end\_timestamp$ is always the date when the policy ends. The policy state is not explicitly tracked in policy data. But implicitly when the $effective\_start\_ts = effective\_end\_ts$ then the policy is off policy and when $effective\_start\_ts > effective\_end\_ts$ then the policy is on policy.

The $pending\_modification$ is also not part of the policy data. Rather, here, it represents an important concept in dealing with policies that a policy must have either no modifications in the accepted state or one modification in the accepted state. An accepted modification acts like a global lock on the policy. No modifications can advance into the accepted state when another modification holds that "accept lock". The existing accepted modification must either be issued or invalidated for other modifications to be issued.

$Policy \triangleq$ [
$original\_start\_ts : TimeRange,$
$effective\_end\_ts : TimeRange,$
$billing\_policy : BillingPolicy,$
$state : PolicyState,$
$revision\_start\_timestamps :$ SUBSET $((0 \,..\, maxRevision) \times TimeRange),$
$renewal\_start\_timestamps :$ SUBSET $(TimeRange),$
$pending\_modification : Modification \cup \{NoModification\}$
]

* *Modification* History
* Last modified *Mon Oct* 11 14:08:06 *PDT* 2021 by *marco*
* Last modified *Mon Jul* 13 20:22:48 *PDT* 2020 by *ASUS*
* *Created* Sat *Jun* 27 14:05:18 *PDT* 2020 by marcderosa

2.3 Charges

There are currently five types of charges in the Socotra framework premiums (represented as PerilCharacteristics), taxes, commissions, fees, and holdbacks. All these charges are represented by the same basic structure, a closed-open interval, the data fields $[start\_timestamp, end\_timestamp)$, and a currency amount. It is always assumed that $start\_timestamp \leq end\_timestamp$. Moreover if we have a description of the policy coverage times as a set of intervals, then the range of a charge must

always fall inside the range of one of those intervals.

$$\forall c \in Charge \cdot \exists i \in CoverageIntervals \cdot i.contains(c)$$

This requirement is not strictly required to bill properly but it is established behavior, that has a natural interpretation. Every charge can be directly traced to a period of contracted coverage.
Amounts can have different interpretations depending on the type of charge and how it is calculated. Premium amounts are always calculated from an original rate, usually in units of currency per month. Usually, then, tax, commission, and rate fee amounts are calculated from the premium amounts by multiplying them by a percentage. When the charges, above, are scaled to a new time period, the amount is converted back into a rate with units of currency per time and multiplied by the new time period. It's a bit round about and round trip conversion looses precision but that is the legacy process. Flat fee and holdback amounts are not interpreted as being derived from a rate. As one might expect their amounts remain constant no matter what their associated interval is and the associated interval just indicates over what period of coverage the charge was assessed.
Since the charges associated with a policy change over time, all charges have additional structure to track their change history. Premium, tax, commission, and holdbacks are all modeled as temporal objects containing the temporal fields $issued\_timestamp$ and $replaced\_timestamp$. More commonly, you may have seen these fields in other frameworks by the names $valid\_from$ and $valid\_to$. This simple structure is used to record an audit history and allow easy determination of the coverage at any time, t, with a filter of the form

$$issued\_timestamp \leq t < (replaced\_timestamp || infinity)$$

The history of fees is recorded in a much less convenient form. Each fee is connected to a linear list of fee_versions, where the latest fee_version describes the currently assessed charges. Needless to say, a fee must have at least one version. To find the coverage at time, t, one has to calculate

$$min\{v \in Version : t < (v.replaced\_timestamp || infinity)\}$$

2.3.1 Fees in Detail

To give a better feeling for fees and how they behave under modification to a policy consider the following graphical descriptions.

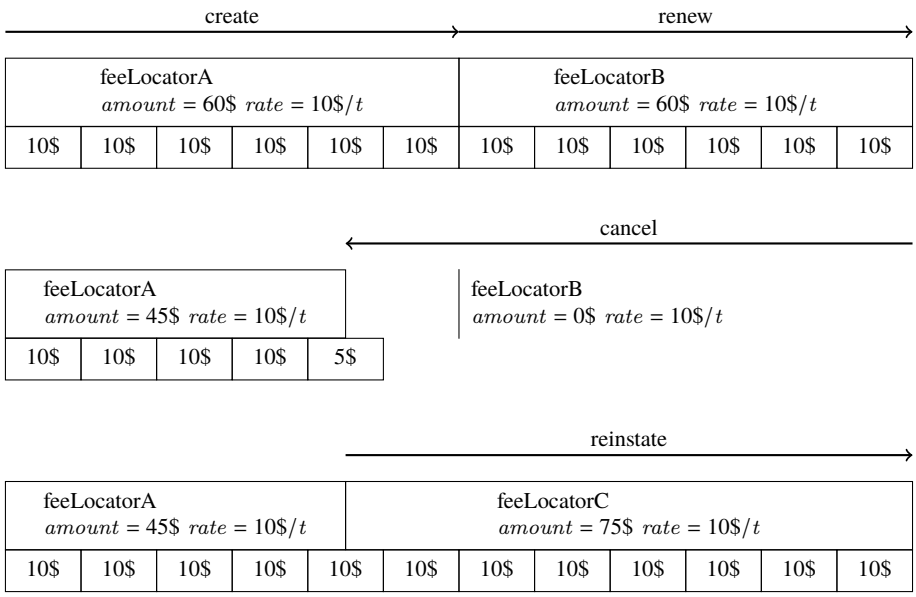

Fig. 2.2 Standard Socotra fixed rate fees: As the extent of the fees (top boxes) change during modifications, the amounts of the fees correspondingly change and fees are projected into schedules (bottom boxes) proportional to the fee / schedule intersection. Fees that fall outside the policy range on a cancellation are discarded by setting their amounts to zero. On reinstatement new fees are created to replace the discarded fees.

Below is Socotra mathematical specification which details the maintenance of flat fees as modifications are applied to a policy. The important points are captured in the comments.

MODULE $FeesFlatCtx1$

EXTENDS $Integers$
CONSTANTS $maxTime$

$ModType \triangleq$ {"Create", "Renew", "Endorse", "Cancel", "Reinstate"}
$Time \triangleq 0 \,..\, maxTime$

$Fee \triangleq$ [
  $pc : Int$,
  $amount : \{4\}$,
  $start\_ts : Time$,
  $end\_ts : Time$
]

$Modification \triangleq$ [
  $start\_ts : Time$,
  $end\_ts : Time$,
  $type : ModType$

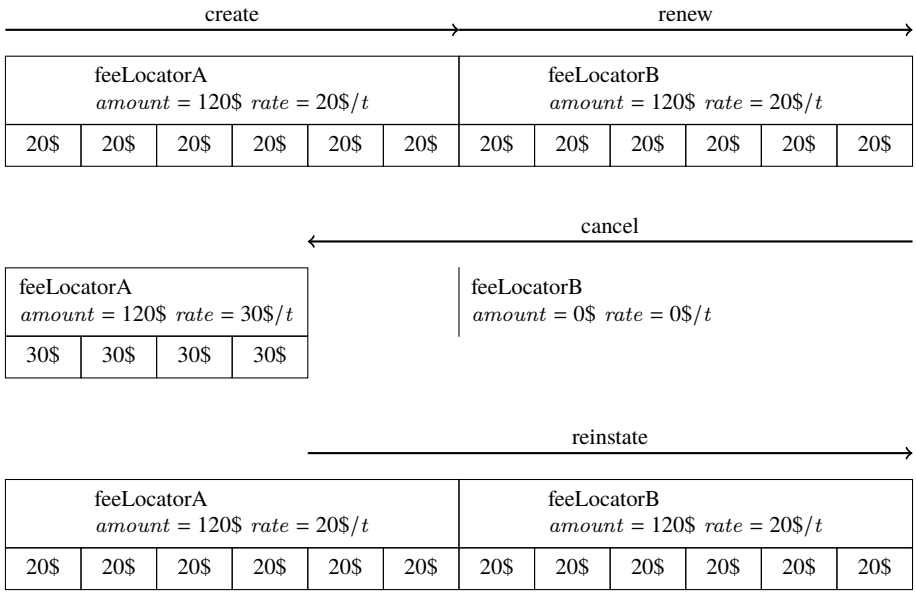

Fig. 2.3 Standard Socotra fixed amount fees: As the extent of the fees (top boxes) change during modifications, the amounts of the fees stay fixed and fees are projected into schedules (bottom boxes) proportional to the fee / schedule intersection. Fees that fall outside the policy range on a cancellation are discarded by setting their amounts to zero. On reinstatement discarded fees are re-extended.

]

$Interval \triangleq$ [
$start\_ts : Time$,
$end\_ts : Time$
]

* *Modification* History
* Last modified *Thu Jul* 15 16:47:34 *PDT* 2021 by *ASUS*
* Created *Wed Jul* 14 19:52:48 *PDT* 2021 by *ASUS*

MODULE $FeesFlatMch1$

EXTENDS $FeesFlatCtx1$, $IntervalOps$, TLC, $Sequences$, $FiniteSets$

CONSTANTS $maxSteps$
VARIABLES $policy$, chs, $fees$, $cancelQ$, pc

In this specification we consider the behavior of a single flat fee as it is acted on by cancellations and reinstatements. The flat fee is modeled by list where the first element records the fee state as it was initially created. Policy modifications change this initial state by appending records to the list. The last element then records the latest state of the fee. As the width of the flat fee changes we would like the amount to stay constant. The only exception to this expectation is when the width is equal to zero. In the zero width case modifications have move the fee outside the policy range and setting it to zero removes all charges associated with it.

$oneOf(set) \triangleq$ CHOOSE $x \in set$: TRUE

$leastOf(pchs) \triangleq$ CHOOSE $x \in pchs : (\forall\, ch \in pchs : x.start\_ts \leq ch.start\_ts)$

Do an interval subtraction. If the result is the empty set then return a zero width fee.

$safeSubtract(fee,\ mod) \triangleq$ LET $tmpSet \triangleq subtract(fee,\ mod)$
IN IF $tmpSet = \{\}$
THEN $[fee$ EXCEPT $!.amount = 0,\ !.end\_ts = fee.start\_ts]$
ELSE $oneOf(tmpSet)$

Try to expand a fee extent stoping when we either hit the max extent of the fee or a gap in the policy coverage.

$extendOnReinstate(latest,\ first,\ mod) \triangleq$
IF $latest.start\_ts = latest.end\_ts$
THEN LET $is \triangleq intersect(first,\ mod)$
IN IF $is = \{\}$
THEN $latest$
ELSE $oneOf(is)$
ELSE IF $latest.start\_ts < latest.end\_ts$
THEN CASE $latest.end\_ts < mod.start\_ts \rightarrow latest$
$\square\, latest.end\_ts = mod.start\_ts \rightarrow$
LET $is2 \triangleq intersect(first,\ [start\_ts \mapsto latest.start\_ts,$
$end\_ts \mapsto mod.end\_ts])$
$tmp \triangleq Assert(Cardinality(is2) = 1,$ "cardinality should be 1")
IN $oneOf(is2)$
$\square$OTHER $\rightarrow Assert($FALSE, "bad case"$)$
ELSE $Assert($FALSE, "bad if"$)$

The policy characteristics set has one or more non overlapping coverage intervals, and characteristic is contained within the policy bounds. For the special case where the width of the policy is zero, there should be one zero width characteristic.

$invPolicy \triangleq$
$\land\ policy \in Interval$
$\land\ policy.start\_ts \leq policy.end\_ts$
$\land\ policy.start\_ts = policy.end\_ts \implies Cardinality(chs) = 1$
$\land\ \forall\, ch \in chs : ch.start\_ts \leq ch.end\_ts$
$\land\ \forall\, ch1,\ ch2 \in chs :\ \lor\ ch1 = ch2$
$\lor\ notOverlaps(ch1,\ ch2)$
$\land\ \forall\, ch \in chs : contains(policy,\ ch)$

If there are no unreinstated cancellations then the policy must have a non-zero width.

$$invQ \triangleq (cancelQ = \{\} \land pc \neq 0) \implies policy.start\_ts < policy.end\_ts$$

A flat fee must always be contained within the bounds of the initial fee version. If the policy is fully cancelled then the amount of the fee must be zero, though a zero amount fee does not imply that the policy is fully cancelled.

$$\begin{array}{l} invFees \triangleq \\ \quad \land \text{LET } latest \triangleq fees[Len(fees)] \\ \quad\quad \text{IN} \quad \land width(latest) = 0 \equiv latest.amount = 0 \\ \quad\quad\quad\quad \land width(latest) \neq 0 \equiv latest.amount = 4 \\ \quad\quad\quad\quad \land latest.start\_ts \geq Head(fees).start\_ts \\ \quad\quad\quad\quad \land latest.end\_ts \leq Head(fees).end\_ts \\ \quad\quad\quad\quad \land width(latest) \neq 0 \implies (\\ \quad\quad\quad\quad\quad\quad \lor latest.end\_ts \leq policy.end\_ts \\ \quad\quad\quad\quad\quad\quad \lor latest.start\_ts \geq policy.start\_ts) \end{array}$$

When a policy is fully cancelled then all of its flat fees must also be fully cancelled. If a policy has an extent and any of its policy characteristics overlap the initial state of the fee, then the latest state of the fee must have an extent and its amount must be equal to the amount of the initial state. Lastly, if latest state of a fee is zero then it must be the case that it does not overlap any of the policy characteristics.

$$\begin{array}{l} invPolicyFees \triangleq \\ \quad \text{LET } latest \triangleq fees[Len(fees)] \\ \quad \text{IN} \quad \land policy.start\_ts = policy.end\_ts \implies latest.amount = 0 \\ \quad\quad\quad \land (width(policy) > 0 \quad \land \exists\, ch \in chs : overlaps(ch,\, Head(fees))) \\ \quad\quad\quad\quad\quad\quad \implies latest.amount = 4 \\ \quad\quad\quad \land latest.amount = 0 \implies \neg(\exists\, ch \in chs : overlaps(ch,\, latest)) \end{array}$$

A new version of a fee should always differ in some way from the version that it replaces.

$$\begin{array}{l} invRepeatFree \triangleq \\ \quad \text{LET } len \triangleq Len(fees) \\ \quad \text{IN} \quad len \geq 2 \implies \neg(\land fees[len-1].start\_ts = fees[len].start\_ts \\ \quad\quad\quad\quad\quad\quad\quad\quad \land fees[len-1].end\_ts = fees[len].end\_ts) \end{array}$$

On reinstatment we would like to restore any reduced flat fees back to their initial width. This in not always possible though for a variety of reason. The reinstatement may not expand the policy range to encompass the initial extent of the fee or there could be gaps in the policy coverage. Because of this, in determining the extent of a fee after a reinstatment we take the coverage period of the policy as an ordered list of intervals separated by gaps and examine their ordered intersections with the initial fee. At a high level, after any reinstatement the restored fee extent should equal the first of the above ordered intersections. The high level intent is realized below with an iterative calculation, that, to a first approximation, trys to expand the current fee's extent as long as the reinstatement starts at the lastest fee version's end timestamp.

$$\begin{array}{l} reinstateOp(m) \triangleq \\ \quad \land pc < maxSteps \\ \quad \land m.type \in \{\text{“Reinstate”}\} \\ \quad \land cancelQ \neq \{\} \end{array}$$

```
    ∧ LET prevMod ≜ leastOf(cancelQ)
      IN  prevMod.start_ts ≤ m.start_ts ∧ prevMod.end_ts = m.end_ts ∧ m.start_ts < m.end_ts
    ∧ LET nextStartTs ≜ IF policy.start_ts = policy.end_ts
                        THEN m.start_ts
                        ELSE policy.start_ts
          nextPolicy ≜ [policy EXCEPT !.start_ts = nextStartTs, !.end_ts = m.end_ts]
          nextChs ≜ IF policy.start_ts = policy.end_ts
                    THEN {[start_ts ↦ m.start_ts, end_ts ↦ m.end_ts]}
                    ELSE UNION {
                                chs,
                                {[start_ts ↦ m.start_ts, end_ts ↦ m.end_ts]}
                               }
          latestV ≜ fees[Len(fees)]
          firstV ≜ fees[1]
          nextD ≜ extendOnReinstate(latestV, firstV, m)
      IN  ∧ policy' = nextPolicy
          ∧ chs' = nextChs
          ∧ cancelQ' = cancelQ \ {leastOf(cancelQ)}
          ∧ pc' = pc + 1
          ∧ IF nextD.start_ts ≠ latestV.start_ts ∨ nextD.end_ts ≠ latestV.end_ts
            THEN fees' = Append(fees, [firstV EXCEPT !.pc = pc,
                                                     !.start_ts = nextD.start_ts,
                                                     !.end_ts = nextD.end_ts])
            ELSE UNCHANGED ⟨fees⟩
```

On cancellation if a flat fee is shrunk to zero width then its amount shrinks to zero. Otherwise the amount will be a constant value and the range of the fee will be in the policy range.

```
cancelOp(m) ≜
    ∧ pc < maxSteps
    ∧ m.type ∈ {"Cancel"}
    ∧ m.start_ts < m.end_ts
    ∧ m.start_ts ≥ policy.start_ts
    ∧ m.end_ts = policy.end_ts
    ∧ LET latestV ≜ fees[Len(fees)]
          nextV ≜ [safeSubtract(latestV, m) EXCEPT !.pc = pc]
          nextPolicy ≜ [policy EXCEPT !.end_ts = m.start_ts]
          nextChs ≜ subtractAll(chs, m)
      IN  ∧ policy' = nextPolicy
          ∧ chs' = IF nextChs = {}
                   THEN {[start_ts ↦ policy.start_ts, end_ts ↦ policy.start_ts]}
                   ELSE nextChs
          ∧ cancelQ' = cancelQ ∪ {m}
          ∧ pc' = pc + 1
          ∧ IF nextV.start_ts ≠ latestV.start_ts ∨ nextV.end_ts ≠ latestV.end_ts
            THEN fees' = Append(fees, nextV)
```

```
                ELSE UNCHANGED ⟨fees⟩

done ≜
    ∧ pc = maxSteps
    ∧ UNCHANGED ⟨policy, chs, fees, cancelQ, pc⟩
```

We start of with a generalized policy and its single policy characteristic and a single fee which may reside anywhere inside the policy interval. Without loss of generality, we can deduce the behavior of all fees from this starting point.

```
Init ≜ ∃ t1, t2 ∈ 0 .. maxTime :
            ∧ t1 < t2
            ∧ policy = [start_ts ↦ 0, end_ts ↦ maxTime]
            ∧ chs = {[start_ts ↦ 0, end_ts ↦ maxTime]}
            ∧ fees = ⟨[pc ↦ 0, amount ↦ 4, start_ts ↦ t1, end_ts ↦ t2]⟩
            ∧ cancelQ = {}
            ∧ pc = 1

Next ≜
    ∨ ∃ m ∈ Modification : ∨ cancelOp(m)
                          ∨ reinstateOp(m)
    ∨ done

Spec ≜ Init ∧ □[Next]⟨policy, chs, fees, cancelQ, pc⟩
```

```
\* Modification History
\* Last modified Thu Jul 15 16:11:29 PDT 2021 by ASUS
\* Created Wed Jul 14 19:53:26 PDT 2021 by ASUS
```

2.3.2 Holdbacks in Detail

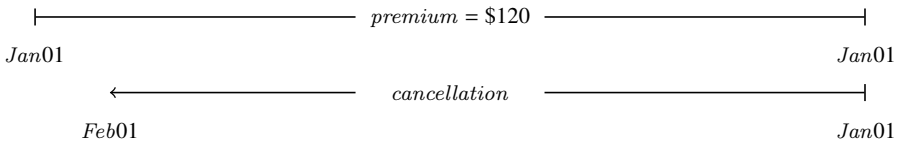

Fig. 2.4 A policy and a cancellation

To explain holdbacks consider a policy running from Jan 1st of one year to Jan 1st of the next year, with a premium of 120 dollars. If a customer decided to cancel this policy starting at Feb 1st, then there are two adjustments that would be made to this policy. First the premium would be prorated to account for the single month of

coverage. Assuming monthly proration 10 dollars would be charged for the month of coverage. Next there may be penalty charges due to the customer canceling the policy early. We call these penalty charges holdbacks, and a typical charge scenario, for our example, might be:

$$\begin{aligned} totalCharges_{Jan01->Feb01} &= prorated(premium) + holdback \\ &= (\frac{1}{12})premium + 0.1(\frac{11}{12})(premium) \end{aligned}$$

where, here, the customer is charged a ten percent penalty fee on the part of the policy that was canceled.
Below is Socotra mathematical specification for maintaining holdbacks. The important points are captured in the comments.

MODULE $ProrationCtx2$

EXTENDS $Integers$, $Reals$

A generic interval type to be used as a short form for any of the formal framework objects below.

$Interval \triangleq [$
    $start\_ts : Int,$
    $end\_ts : Int$
]

Below we model serveral objects in the framework, each of which is an interval. All characteristics are represented as a rate, premium, carrying object. The pc, program counter, element gives the element a uuid which the spec uses to track objects.

$Characteristic \triangleq [$
    $start\_ts : Int,$
    $end\_ts : Int,$
    $premium : Int,$
    $monthPremium : 1 \,..\, 2,$
    $pc : Int$ program counter
]

A rough analog of the data that we might keep in the $policy\_modification$ table. The spec place this data in the $cancelQ$.

$Modification \triangleq [$
    $start\_ts : Int,$
    $end\_ts : Int,$
    $pc : Int$
]

Holdbacks are penalty charges that can be assessed when the coverage interval of a policy is reducted. Holdbacks, currently, can only happen for cancellations.

$Holdback \triangleq [$
    $start\_ts : Int,$
    $end\_ts \;\; : Int,$

$amount : Int,$
$pc : Int$
]

* *Modification* History
* Last modified *Mon Oct* 11 14:33:53 *PDT* 2021 by *marco*
* Last modified Sun *Feb* 28 21:28:02 *PST* 2021 by *ASUS*
* Last modified *Mon Feb* 15 10:50:16 *PST* 2021 by *marcderosa*
* *Created Mon Feb* 15 10:49:27 *PST* 2021 by *marcderosa*

MODULE *ProrationMch2*

EXTENDS *ProrationCtx2*, *Sequences*, *IntervalOps*, *FiniteSets*, *TLC*

VARIABLES *policy*, *policyChs*, *pc*, *cancelQ*, *holdbacks*

CONSTANTS *maxPc*, *maxT*, *pluginActive*

$oneOf(S) \triangleq$ CHOOSE $x \in S$: TRUE
$maxOf(S) \triangleq$ CHOOSE $x \in S : \forall\, y \in S : x \geq y$
$minOf(S) \triangleq$ CHOOSE $x \in S : \forall\, y \in S : x \leq y$

$maxCh(chs) \triangleq$ CHOOSE $ch \in chs : (\forall\, ds \in chs : ch.start\_ts \geq ds.start\_ts)$
$addPremium(n,\ chB) \triangleq n + chB.premium$

$isNotZeroWidth(ch) \triangleq ch.start\_ts < ch.end\_ts$

RECURSIVE $fold(\_,\ \_,\ \_)$
$fold(Op(\_,\ \_),\ init,\ Txs) \triangleq$ IF $Txs = \{\}$
THEN $init$
ELSE LET $tx \triangleq oneOf(Txs)$
IN $fold(Op,\ Op(init,\ tx),\ Txs \setminus \{tx\})$

$sumPremium(chs) \triangleq fold(addPremium,\ 0,\ chs)$

$filter(fn(\_),\ xs) \triangleq \{x \in xs : fn(x)\}$

$InvTypeQ \triangleq \forall\, n \in 1\,..\,Len(cancelQ) : cancelQ[n] \in Modification$
$InvTypeCh \triangleq \forall\, ch \in policyChs : ch \in Characteristic$
$InvTypeHoldback \triangleq \forall\, hb \in holdbacks : hb \in Holdback$

When the policy is fully canceled there is always 1 policy characteristic
$InvPolicy \triangleq$
$\wedge\ policy \in Interval$
$\wedge\ policy.start\_ts \leq policy.end\_ts$
$\wedge\ (policy.start\_ts = policy.end\_ts \wedge pc > 0) \implies Cardinality(policyChs) = 1$
$\wedge\ (policy.start\_ts = policy.end\_ts \wedge pc > 0) \implies ($

```
        LET ch ≜ CHOOSE ch ∈ policyChs : TRUE
        IN  ch.start_ts = ch.end_ts ∧ ch.start_ts = policy.start_ts
        )
```

The start and ends of the policy exactly bracket the range of the characteristics. And none of the characteristics, which are valid for the policy, may overlap.

$$
\begin{aligned}
&InvPolicyChs \triangleq \\
&\quad \land\ pc > 0 \implies Cardinality(policyChs) > 0 \\
&\quad \land\ pc > 0 \implies policy.start\_ts = minOf(\{ch.start\_ts : ch \in policyChs\}) \\
&\quad \land\ pc > 0 \implies policy.end\_ts = maxOf(\{ch.end\_ts : ch \in policyChs\}) \\
&\quad \land\ (\exists\, ch \in policyChs : ch.start\_ts = ch.end\_ts) \implies \\
&\qquad\quad policy.start\_ts = policy.end\_ts \\
&\quad \land\ \forall\, ch1,\ ch2 \in policyChs : (ch1 \neq ch2) \implies notOverlaps(ch1,\ ch2)
\end{aligned}
$$

Holdbacks are always bounded by the interval of the policy. When there are no outstanding cancellations, then there can be no active *holdbacks*. This last property comes about as only cancellations create *holdbacks* and reinstatements always reverse the *holdbacks* created by their matched cancellation.

$$
\begin{aligned}
&InvHoldbacks \triangleq \\
&\quad \land\ Cardinality(holdbacks) = Len(cancelQ) \\
&\quad \land\ \forall\, hb \in holdbacks :\ \land\ hb.start\_ts = policy.start\_ts \\
&\qquad\qquad\qquad\qquad\ \land\ hb.end\_ts \leq policy.end\_ts
\end{aligned}
$$

When the policy start date equals the policy end date, that is a sign that there are one or more unreversed cancellations in the policies modification set. In most cases we expect that the most recent, unreversed cancellation will have a start date equal to the policies end data, but this is not always the case. Since policies can have gaps in coverage, the actual policy end date can sometimes be less than the most recent cancellations start date. Then if there are unrevered cancellations we may have valid *holdbacks* and when all cancellations have been reversed there should be no valid *holdbacks*.

$$
\begin{aligned}
&InvCancelQ \triangleq \\
&\quad \land\ (pc \neq 0 \land policy.start\_ts = policy.end\_ts) \implies Len(cancelQ) > 0 \\
&\quad \land\ Len(cancelQ) > 0 \implies \text{LET } last \triangleq cancelQ[Len(cancelQ)] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{IN } last.start\_ts \geq policy.end\_ts \\
&\quad \land\ \forall\, n \in 1\,..\,Len(cancelQ) : (\exists\, hb \in holdbacks : hb.pc = cancelQ[n].pc)
\end{aligned}
$$

---

make a default, priced characteristic

```
mkDfltCh(from, to, progCtr) ≜ [
        start_ts ↦ from,
        end_ts ↦ to,
        premium ↦ to − from,
        monthPremium ↦ 1,
        pc ↦ progCtr
    ]

prorate(chs, progCtr) ≜ {
    [ch EXCEPT !.premium = ch.monthPremium * (ch.end_ts − ch.start_ts),
```

```
                    !.pc = progCtr + 1] : ch ∈ chs}

reprice(chs, progCtr) ≜ {[ch EXCEPT !.premium = ch.end_ts − ch.start_ts,
                                    !.pc = progCtr + 1] : ch ∈ chs}

reducedChsFrwork(chs, int, progCtr) ≜
    LET remainSet ≜ subtractAll(chs, int)
        prorateSet ≜ {r ∈ remainSet : r ∉ chs}
    IN  IF remainSet = {}
        THEN {[
                  start_ts ↦ int.start_ts,
                  end_ts ↦ int.start_ts,
                  premium ↦ 0,
                  pc ↦ progCtr + 1
              ]}
        ELSE UNION {
                  (remainSet \ prorateSet),
                  prorate(prorateSet, progCtr)
              }
```

In our abstract model the proration has the effect of adding a retained amount to the last characteristic in the reduced set. Since we generally want to add a retained amount, when we split on a characteristic boundary, we have to add the retained amount to a characteristic ouside of the modification interval. In general, this function would potentially change premium, tax, or fee amounts.

```
reducedChsPlugin(chs, int, progCtr) ≜
    LET remainSet ≜ subtractAll(chs, int)
        prorateSet ≜ {r ∈ remainSet : r ∉ chs}
        mx ≜ IF prorateSet = {}
             THEN maxCh(remainSet)
             ELSE maxCh(prorateSet)
    IN  IF remainSet = {}
        THEN {[
                  start_ts ↦ int.start_ts,
                  end_ts ↦ int.start_ts,
                  premium ↦ 0,
                  pc ↦ progCtr + 1
              ]}
        ELSE UNION {
                  (remainSet \ {mx}),
                  prorate({mx}, progCtr)
              }

reducedChs(chs, int, progCtr) ≜
    IF pluginActive = 0
    THEN reducedChsFrwork(chs, int, progCtr)
    ELSE reducedChsPlugin(chs, int, progCtr)
```

Below ts is the start date of the endorsement and te is the resulting end date of the endorsed policy. Although te can be greater than the end of the policy, in this spec we only model the case where $te \leq$ the policy end dt. Note, that endorsments can not change the policy's holdback collection.

$$
\begin{aligned}
&endorsePolicy(ts,\ te) \triangleq \\
&\quad \land\ pc < maxPc \\
&\quad \land\ policy.start\_ts < policy.end\_ts \\
&\quad \land\ ts \geq policy.start\_ts \land ts < te \land te \leq policy.end\_ts \\
&\quad \land\ Len(cancelQ) = 0 \\
&\quad \land\ \text{LET } endorseInt \triangleq [start\_ts \mapsto ts,\ end\_ts \mapsto te] \\
&\qquad\quad repriceSet \triangleq reprice(intersectAll(policyChs,\ endorseInt),\ pc) \\
&\qquad\quad splitSet \triangleq \{ch \in policyChs : within(ts,\ ch)\} \\
&\qquad\quad prorateSet \triangleq prorate(\{ \\
&\qquad\qquad dh \in subtractAll(splitSet,\ endorseInt) : dh.start\_ts < ts\},\ pc) \\
&\qquad\quad unchangedSet \triangleq \{ch \in policyChs : ch.end\_ts \leq ts\} \\
&\qquad\quad nextPolicyChs \triangleq \text{UNION } \{repriceSet,\ prorateSet,\ unchangedSet\} \\
&\qquad\quad nextMaxT \triangleq maxOf(\{ch.end\_ts : ch \in nextPolicyChs\}) \\
&\qquad \text{IN} \quad \land\ Assert(nextMaxT \leq te,\ \text{“unexpected max”}) \\
&\qquad\qquad \land\ Assert(\\
&\qquad\qquad\qquad policy.end\_ts = te \implies sumPremium(policyChs) = sumPremium(nextPolicyChs), \\
&\qquad\qquad\qquad \text{“unmatched premiums”}) \\
&\qquad\qquad \land\ policy' = [policy \text{ EXCEPT } !.end\_ts = nextMaxT] \\
&\qquad\qquad \land\ policyChs' = nextPolicyChs \\
&\qquad\qquad \land\ pc' = pc + 1 \\
&\qquad\qquad \land\ \text{UNCHANGED } \langle cancelQ,\ holdbacks\rangle
\end{aligned}
$$

An abstract cancellation. When cancellations occur new *holdbacks* will be added to the policy and, if there are any existing *holdbacks*, those that will fall outside of the cancelled policy interval are recreated so that their interval remains within the policy interval. Since policies may have gaps in coverage the new policy end date will be less than or equal to the cancellation date, and this end date is determined from the set of valid policy characteristics.

$$
\begin{aligned}
&cancelPolicy(t) \triangleq \\
&\quad \land\ pc < maxPc \\
&\quad \land\ policy.end\_ts > policy.start\_ts \\
&\quad \land\ t \geq policy.start\_ts \\
&\quad \land\ t < policy.end\_ts \\
&\quad \land\ \text{LET } cancelInt \triangleq [start\_ts \mapsto t,\ end\_ts \mapsto policy.end\_ts] \\
&\qquad\quad nextPolicyChs \triangleq reducedChs(policyChs,\ cancelInt,\ pc) \\
&\qquad\quad delHoldbacks \triangleq overlapsAll(holdbacks,\ cancelInt) \\
&\qquad\quad nextMaxT \triangleq maxOf(\{ch.end\_ts : ch \in nextPolicyChs\}) \\
&\qquad \text{IN} \quad \land\ Assert(nextMaxT \leq t,\ \langle nextMaxT,\ t\rangle) \\
&\qquad\qquad \land\ policy' = [policy \text{ EXCEPT } !.end\_ts = nextMaxT] \\
&\qquad\qquad \land\ policyChs' = nextPolicyChs \\
&\qquad\qquad \land\ cancelQ' = Append(\\
&\qquad\qquad\qquad cancelQ,
\end{aligned}
$$

```
                [start_ts ↦ t, end_ts ↦ policy.end_ts, pc ↦ pc + 1])
        ∧ holdbacks' = UNION {
                (holdbacks \ delHoldbacks),
                {[
                    start_ts ↦ policy.start_ts,
                    end_ts   ↦ nextMaxT,
                    amount ↦ policy.end_ts − t,
                    pc ↦ pc + 1
                ]},
                {[dh EXCEPT !.start_ts = policy.start_ts,
                            !.end_ts = nextMaxT] : dh ∈ delHoldbacks}}
        ∧ pc' = pc + 1
```

An abstract creation or renewal is extended by adding a characteristic. Creations and renewals never create new $holdbacks$.

```
extendPolicy(t) ≜
    ∧ pc < maxPc
    ∧ t > policy.end_ts
    ∧ Len(cancelQ) = 0
    ∧ LET addCh ≜ mkDfltCh(policy.end_ts, t, pc + 1)
      IN  ∧ policy' = [policy EXCEPT !.end_ts = t]
          ∧ policyChs' = policyChs ∪ {addCh}
          ∧ pc' = pc + 1
          ∧ UNCHANGED ⟨cancelQ, holdbacks⟩
```

An abstract reinstatment. When we reinstate we check if there are any $holdbacks$ associated with the reinstatements cancellation. If there are then we can invalidate those previous $holdbacks$, as a reinstatement must reverse the effects of its cancellation. When a reinstatement changes the start date of a policy, we may also have to rewrite valid holdback to maintain holdback intervals within the policy interval.

```
reinstatePolicy(t) ≜
    ∧ pc < maxPc
    ∧ Len(cancelQ) ≠ 0
    ∧ LET last ≜ cancelQ[Len(cancelQ)]
          addCh ≜ mkDfltCh(t, last.end_ts, pc + 1)
          delHb ≜ {hb ∈ holdbacks : hb.pc = last.pc}
          policyStart ≜ IF policy.start_ts = policy.end_ts THEN t ELSE policy.start_ts
      IN  ∧ t ≥ last.start_ts ∧ t < last.end_ts
          ∧ policy' = [policy EXCEPT !.start_ts = policyStart, !.end_ts = last.end_ts]
          ∧ policyChs' = filter(isNotZeroWidth, (UNION {policyChs, {addCh}}))
          ∧ holdbacks' = {
                [hb EXCEPT !.start_ts = max(hb.start_ts, policyStart),
                           !.end_ts = max(hb.end_ts, policyStart)] : hb ∈ (holdbacks \ delHb)
                }
          ∧ cancelQ' = [n ∈ 1 .. (Len(cancelQ) − 1) ↦ cancelQ[n]]
          ∧ pc' = pc + 1
```

$term \triangleq$
$\quad \land\ pc \geq maxPc$
$\quad \land$ UNCHANGED $\langle policy,\ policyChs,\ holdbacks,\ pc,\ cancelQ \rangle$

$Init \triangleq$
$\quad \land\ policy = [start\_ts \mapsto 0,\ end\_ts \mapsto 0]$
$\quad \land\ policyChs = \{\}$
$\quad \land\ holdbacks = \{\}$
$\quad \land\ cancelQ = \langle\rangle$
$\quad \land\ pc = 0$

$Next \triangleq$
$\quad \lor \exists\, t \in 0\ldots maxT :$
$\qquad \lor\ extendPolicy(t)$
$\qquad \lor\ cancelPolicy(t)$
$\qquad \lor\ reinstatePolicy(t)$
$\quad \lor \exists\, ts,\ te \in 0\ldots maxT : endorsePolicy(ts,\ te)$
$\quad \lor\ term$

$Spec \triangleq\ Init \land \Box[Next]_{\langle policy,\ policyChs,\ holdbacks,\ pc,\ cancelQ \rangle}$

\ ∗ *Modification* History
\ ∗ Last modified *Fri Apr* 02 12:06:16 *PDT* 2021 by *ASUS*
\ ∗ Last modified *Mon Feb* 15 11:44:51 *PST* 2021 by *marcderosa*
\ ∗ Created *Mon Feb* 15 10:49:49 *PST* 2021 by *marcderosa*

2.4 Observables

2.4.1 Rating Premiums, Taxes, Commissions

Premiums, taxes and commissions are calculated at the peril level via the framework function, Rater.getPricedCharacteristics, which has the simplified signature:

$getPricedCharacteristics : RaterPriceRequest \rightarrow Map\ Locator\ PerilCharacteristics$

The RaterPriceRequest contains a collection of unpriced (policy characteristic, exposure characteristic, peril characteristic) triplets which have the appropriate field values which one intends to price.

$triple : (PolicyCharacteristics,\ ExposureCharacteristics,\ PerilCharacteristics)$

The triplet is formed such that the interval of the exposure characteristic contains the peril characteristics and the interval of the policy characteristic contains the peril characteristic. This containment property, which has been discussed, dictates how

peril characteristics split their parent characteristics and how parent characteristics split peril characteristics. One can see that the containment property is driven by the needs of the rater to have homogeneous coverage across a triplet of characteristics in order to price at the peril level. The containment property also tells us why we are generally unable to price one peril based on properties of another peril. Perils do not compare to one another as most generally a peril characteristic in one peril will not be contained by any peril characteristics in another peril.
The RaterPriceRequest also contains a policy object. The policy object should satisfy the condition that for any characteristics triplet that exists in the RaterPriceRequest each of those characteristics must also be discoverable by iterating through the policy, exposure, peril tree. Also any policy_modification that can be discovered on any characteristics triplet must be discoverable in the policy.getModifications() collection.
With the request object formed, $req : (policy,\ triple)$, the base $rater$ function is called on each triple to obtain a premium, commission, and tax

$$premium = \Delta t \cdot rater(policy, triple_{peril}).premium$$

Similarly for commissions and taxes

$$commission = \Delta t \cdot rater(policy, triple_{peril}).commission$$
$$tax = rater(policy, triple_{peril}).tax$$

Lastly, the $getPricedCharacteristics$ function assembles and returns new priced characteristics indexed by locator.

2.4.2 Rating Fees

Fees are calculated at the policy level via the framework function, LiquidRenderClient.renderPolicyCalculations, which has the simplified signature:

$$renderPolicyCalculations : Context \rightarrow List\ Fee$$

The context contains a policy and the in process policy modification. The policy must contain fully calculated premiums, commission, and taxes on all its characteristics. And the policy modification must have all gross premium and gross tax amounts calculated. Then, a list of fees are calculated by calling the rater function as:

$$renderPolicyCalculations = rater(policy', policyModification')$$

2.5 Modifications - The State Model

So far I have described the main modification functions $extend$, $reduce$, $override$, or to use the actual system functions $create$, $renew$, $reinstate$, $cancel$, $endorse$, as monolithic functions, but that is not quite the case in the framework. In fact, the modification functions are behaviors, a collection of functions which bring the policy in steps to a final target state. These steps, in order, are generally called $draft$, $quote$, $accept$, and $issue$. I think the best way to to succinctly and accurately described this behavioral flow is with the standard notation of hierarchical state diagrams. Figure 2.5 depicts the modification state machine for all current framework modifications. The vertical dotted line indicates that each $Modification$ in the set is operating in parallel with other modifications in the set. The $*$ appended to each $Modification$ type indicates that there can be zero or many modifications of any type. In the specifications ahead we often abstract further on this idea of many modifications by letting a policy's initial modification set consist of the universe of all $Draft$ modifications. In our specifications this allows us to reduce the state space in our developments without loss of generality.
As a supplement to the modification state machine. I am also going to add one more state machine showing the interaction between the modification state machine and the policy state machine, figure 2.6. What I want to stress here is that it is essential for clear thinking to separate modification state from policy state. Often as developers we get sloppy and describe policies as being in the $Accepted$ or some other state when actually we talking about a particular modification. The other point to stress is that state changes of significance for policies only happen on $onIssue$ events. For an issuer and a purchaser of an insurance insurance policy, the $onIssue$ event is the only one that changes real world liabilities.

2.6 Characteristics - A Representative Interval

Characteristics are containers that record a set of attribute values (ex: number_of_ drivers = 2, make_of_car = Honda, resale_value=10000) which a policy contract is based on, and additionally a time interval over which those attribute values are valid. Abstractly, it's a combination of a map and a time interval.
Below we just consider the time interval part of a characteristics and detail how those intervals split as the standard modification operations are applied to intervals

MODULE $CharacteristicsSplitCtx0$

EXTENDS $Integers$
CONSTANTS $MinTime$, $MaxTime$

$Characteristics \triangleq$ [
    $start\_ts : Int$,
    $end\_ts : Int$
]

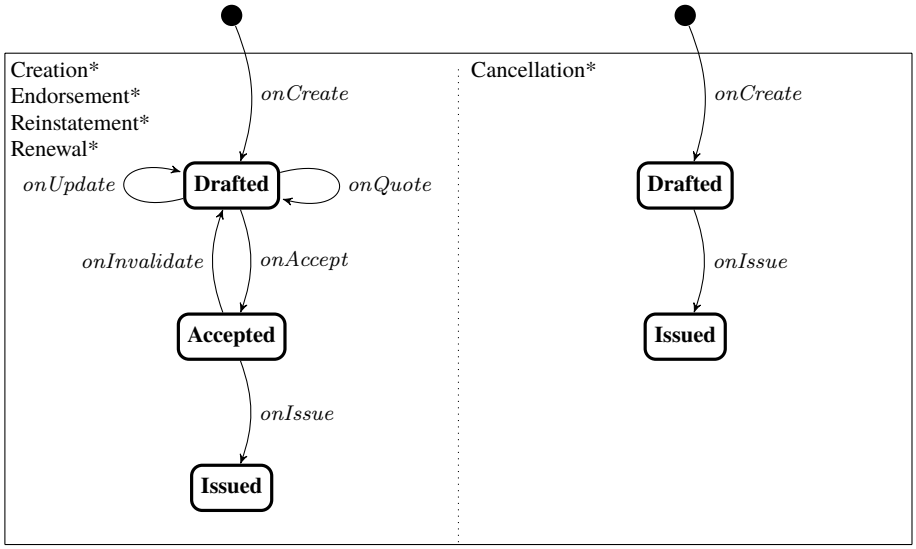

Fig. 2.5 The Socotra *Modification* state model. Any number of modifications can exist concurrently in the modifications set. Each modification transitions between its states, isolated from the effects of other modifications.

$$
\begin{aligned}
&VCharacteristics \triangleq \{ch \in Characteristics : \\
&\quad ch.start\_ts \leq ch.end\_ts \\
&\quad \wedge\, ch.start\_ts \geq MinTime
\end{aligned}
$$

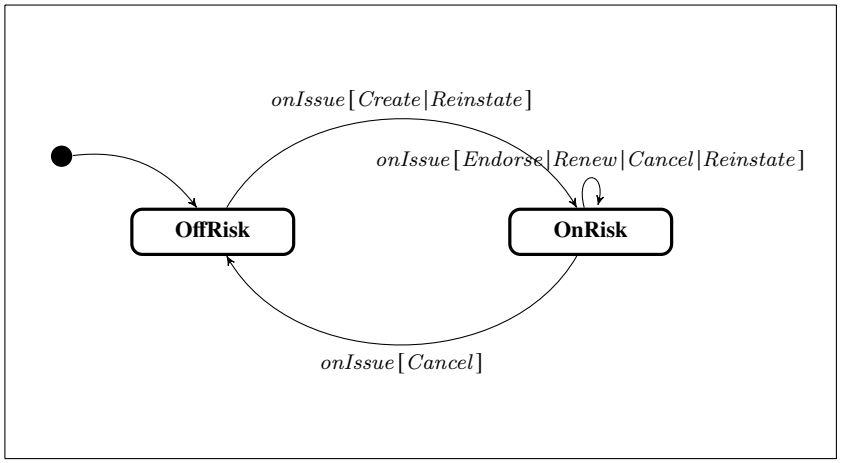

Fig. 2.6 The Socotra *Policy* state model. A *Policy* starts out as *OffRisk* and its effective state changes whenever an *onIssue* event fires.

$\land\ ch.end\_ts \leq MaxTime\}$

* Modification History
* Last modified Sat Oct 17 21:56:37 PDT 2020 by $ASUS$
* $Created\ Fri\ Oct$ 16 13:25:04 PDT 2020 by $ASUS$

MODULE $CharacteristicsSplitMch0$

EXTENDS $CharacteristicsSplitCtx0$, $FiniteSets$

VARIABLES $policyChs$, $exChs$, $perilChs$, $cPolicyChs$, $cExChs$, $cPerilChs$, pc

There are three tables in the persistance layer that hold characteristics, $policy\_characteristics$, $exposure\_characteristics$, and $peril\_characteristics$. All of these tables are examples of a well exstablished sql modeling structure called a temporal table. The purpose of that structure is to describe a current valid state and at the same time to record the entire transaction history, making it easily reconstructable for any given time. Here, I examine characteristic transformations for simple one exposure, one peril policy. the valid state of the policy's characteristics are held in the variables $policyChs$, $exChs$, and $perilChs$. Then, I mimic the history functionality of a temporal table by storing history in the variables $cPolicyChs$, $cExChs$, $cPerilChs$. The framework itself only needs to access characteristic history during a cancellation reinstatement. So these variables hold characteristic state just before the application of the last cancellation.

$max(a,\ b) \triangleq$ IF $a \geq b$ THEN a ELSE b
$min(a,\ b) \triangleq$ IF $a \leq b$ THEN a ELSE b

Interval Predicates

Since characteristics are intervals, making decisions on them is facilitated by a fundamental set of predicate combinators over intervals. The fundamental interval combinators sufficient for all characteristic decisions at $Socotra$ are listedbelow. Most of them are implemented in the framework Duration class (or when not, are implemented in an ad-hoc manner).

$notOverlaps(chA,\ chB) \triangleq\ chB.end\_ts \leq chA.start\_ts$
$\quad \lor\ chB.start\_ts \geq chA.end\_ts$

$overlaps(chA,\ chB) \triangleq\ \neg notOverlaps(chA,\ chB)$

$contains(chA,\ chB) \triangleq\ chA.start\_ts \leq chB.start\_ts \land chB.end\_ts \leq chA.end\_ts$

$equals(chA,\ chB) \triangleq$
$\quad \land\ chA.start\_ts = chB.start\_ts$
$\quad \land\ chA.end\_ts = chB.end\_ts$

$width(ch) \triangleq\ ch.end\_ts - ch.start\_ts$

$within(tm,\ ch) \triangleq\ ch.start\_ts < tm \land tm < ch.end\_ts$

Interval Functions

There are two inverval functions which are the fundamental combinators on which all characteristic functions are based. These are: (1) subtract: Interval $\rightarrow$ Interval $\rightarrow$ Set Interval, returns the parts of the first interval that do not overlap with the second interval. And will be a set of cardinality between 0 and 2. (2) union: Interval $\rightarrow$ $Inteval$ $\rightarrow$ Set Interval, returns the part of the first and second intervals that overlap. And will be a set of cardinality between 0 and 1.

$subtract(chA,\ chB) \triangleq$

```
    CASE notOverlaps(chA, chB) → {chA}
    □OTHER →
      IF within(chB.start_ts, chA) THEN
          IF within(chB.end_ts, chA) THEN {
              [start_ts ↦ chA.start_ts, end_ts ↦ chB.start_ts],
              [start_ts ↦ chB.end_ts, end_ts ↦ chA.end_ts]
              }
           ELSE
              {[start_ts ↦ chA.start_ts, end_ts ↦ chB.start_ts]}
       ELSE
          IF within(chB.end_ts, chA) THEN
              {[start_ts ↦ chB.end_ts, end_ts ↦ chA.end_ts]}
           ELSE
              {}

union(chA, chB) ≜
    IF notOverlaps(chA, chB)
     THEN {}
     ELSE {[
              start_ts ↦ max(chA.start_ts, chB.start_ts),
              end_ts ↦ min(chA.end_ts, chB.end_ts)
           ]}
```

Other functions

```
maxEndTs(chs) ≜
    LET   endTss ≜ {ch.end_ts : ch ∈ chs}
    IN  CHOOSE tm ∈ endTss : ∀ el ∈ endTss : tm ≥ el

minStartTs(chs) ≜
    LET startTss ≜ {ch.start_ts : ch ∈ chs}
    IN  CHOOSE tm ∈ startTss : ∀ el ∈ startTss : tm ≤ el

one(es) ≜ CHOOSE e ∈ es : TRUE
```

In the socotra product a policy always has at least one policy characteristic. One might think that when a policy is fully cancelled it would have 0 policy characteristics but this is not the case. A fully cancelled policy has a single characteristic, fully populated in all respects but with zero width.

```
offPolicy ≜
    ∧ Cardinality(policyChs) = 1
    ∧ LET ch ≜ one(policyChs)
      IN  width(ch) = 0

onPolicy ≜ ¬offPolicy
```

For a policy's characteristic data to be correct. All manipulations should retain the properties below, with some caveats as noted below.

Containment Property: Each peril characteristic must be contained by one characteristic in its parent exposure. Each peril characteristic must be contained by one characteristic in its policy. There are no containment relationships, however, between policy and exposure characteristics. The effect of this property is that splits of peril characteristics migrate up to split both exposure and policy characteristics, while splits of policy or exposure characteristics only migrate down to peril characteristics.

$$
\begin{aligned}
&InvContainment \triangleq \\
&\quad \wedge \forall\, ch \in perilChs : (\mathsf{e}\, pch \in policyChs : contains(pch,\ ch)) \\
&\quad \wedge \forall\, ch \in perilChs : (\mathsf{e}\, ech \in exChs : contains(ech,\ ch))
\end{aligned}
$$

Existance Property: a policy, exposure, or peril always has at least one characteristic. In the concrete implementation this property sometimes will not hold after a $reduction()$ operation like cancellation. Specifically if the start date of a cancellation operation is less than the minimum start date of a set of characteristics then all the characteristics will be removed.

$$
\begin{aligned}
&InvNotEmpty \triangleq Cardinality(policyChs) > 0 \\
&\quad \wedge Cardinality(exChs) > 0 \\
&\quad \wedge Cardinality(perilChs) > 0
\end{aligned}
$$

Off $Policy$ Property: When a policy has been fully cancelled each node in the characteristics tree contains 1 characteristic of zero length. In the concrete implementation there is some deviance here as explained in the $Existance$ Property. Of particular note, exposure characteristics do not behave like policy or peril characteristics which grow and shrink with operations. Exposure characteristics only grow in size. The minimum start date in a set of exposure characteristics is always equal to the start date of the exposure when created. The maximum end date in a set of exposure characteristics is always equal to the maximum policy end date that the set of exposures has observed during its lifetime.

$$
\begin{aligned}
&InvOffPolicy \triangleq offPolicy \implies \\
&\qquad (\\
&\qquad\quad \wedge Cardinality(exChs) = 1 \\
&\qquad\quad \wedge Cardinality(perilChs) = 1 \\
&\qquad\quad \wedge \textsc{let}\ pch \triangleq one(policyChs) \\
&\qquad\qquad\qquad ech \triangleq one(exChs) \\
&\qquad\qquad\qquad prlCh \triangleq one(perilChs) \\
&\qquad\qquad \textsc{in}\ \ width(ech) = 0 \\
&\qquad\qquad\quad \wedge width(prlCh) = 0 \\
&\qquad\qquad\quad \wedge ech.start\_ts\ \ = pch.start\_ts \\
&\qquad\qquad\quad \wedge ech.end\_ts = pch.end\_ts \\
&\qquad\qquad\quad \wedge prlCh.start\_ts = pch.start\_ts \\
&\qquad\qquad\quad \wedge prlCh.end\_ts = pch.end\_ts \\
&\qquad)
\end{aligned}
$$

On $Policy$ Property: When a policy is active, it has no zero length characteristics

$$
\begin{aligned}
&InvOnPolicyNoZeroLengthChs \triangleq onPolicy \implies (\\
&\quad \wedge \forall\, prlCh \in perilChs : prlCh.start\_ts < prlCh.end\_ts \\
&\quad \wedge \forall\, exCh \in exChs : exCh.start\_ts < exCh.end\_ts \\
&\quad \wedge \forall\, ch \in policyChs : ch.start\_ts\ \ < ch.end\_ts \\
&\quad)
\end{aligned}
$$

Bounding Property: For all the exposures characteristics at least 1 must start the policy interval; for all the peril characteristics at least one must start the policy interval. The same is true for finishing the policy interval.

$$
\begin{array}{l}
InvBoundsStart \triangleq \text{LET } start \triangleq minStartTs(policyChs) \\
\quad \text{IN} \quad \wedge \, \mathsf{e}\, ech \in exChs : ech.start\_ts = start \\
\qquad \wedge \, \mathsf{e}\, pch \in perilChs : pch.start\_ts = start \\
InvBoundsEnd \triangleq \text{LET } end \triangleq maxEndTs(policyChs) \\
\quad \text{IN} \quad \wedge \, \mathsf{e}\, ech \in exChs : ech.end\_ts = end \\
\qquad \wedge \, \mathsf{e}\, pch \in perilChs : pch.end\_ts = end
\end{array}
$$

The various modification operations in the framework affect characteristics in 3 abstract ways. $Api.create()$, $Api.reinstate()$, and $Api.renew()$ modifications are abstractly $extend()$ operations. $Api.cancel()$ is abstractly a $reduce()$ operation. And $Api.endorse()$, $Api.update()$ are abstractly $update()$ operations. Below I discuss the character of these abstract operations in more detail.

From the point of view of characteristics a reinstatement can be thought of as application of the set union operator. For example, given an existing set of characteristics, $\{i1,\ i2,\ i3\}$ and a set that are being reinstated, $\{i4,\ i5\}$. The result is $\{i1,\ i2,\ i3\} + \{i4,\ i5\} = \{i1,\ i2,\ i3,\ i4,\ i5\}$

$$
\begin{array}{l}
reinstate(mod) \triangleq \\
\quad \wedge\ pc < 5 \\
\quad \wedge\ cPolicyChs \neq \{\} \\
\quad \wedge\ mod.start\_ts < mod.end\_ts \\
\quad \wedge\ mod.start\_ts \geq minStartTs(cPolicyChs) \\
\quad \wedge\ mod.end\_ts = maxEndTs(cPolicyChs) \\
\quad \wedge \text{ IF } offPolicy \\
\qquad \text{THEN} \quad \wedge\ policyChs' = \text{UNION } \{union(plcyCh,\ mod) : plcyCh \in cPolicyChs\} \\
\qquad\qquad \wedge\ exChs' = \text{UNION } \{union(exCh,\ mod) : exCh \in cExChs\} \\
\qquad\qquad \wedge\ perilChs' = \text{UNION } \{union(prlCh,\ mod) : prlCh \in cPerilChs\} \\
\qquad \text{ELSE} \quad \wedge\ policyChs' = \text{UNION } \{ \\
\qquad\qquad\qquad policyChs, \\
\qquad\qquad\qquad \text{UNION } \{union(plcyCh,\ mod) : plcyCh \in cPolicyChs\}\} \\
\qquad\qquad \wedge\ exChs' = \text{UNION } \{ \\
\qquad\qquad\qquad exChs, \\
\qquad\qquad\qquad \text{UNION } \{union(exCh,\ mod) : exCh \in cExChs\}\} \\
\qquad\qquad \wedge\ perilChs' = \text{UNION } \{ \\
\qquad\qquad\qquad perilChs, \\
\qquad\qquad\qquad \text{UNION } \{union(prlCh,\ mod) : prlCh \in cPerilChs\}\} \\
\quad \wedge\ cPolicyChs' = \{\} \\
\quad \wedge\ cExChs' = \{\} \\
\quad \wedge\ cPerilChs' = \{\} \\
\quad \wedge\ pc' = pc + 1
\end{array}
$$

From the point of view of characteristics a cancellation is the application of the subtract operator, above, over the set of existing characteristics. For illustration, consider the intervals

⊢ —— A ——— ⊢ ——— B ——— ⊢ ——— C ——— | an existing set of characteristics
⊢ ——————— D ——————— | a cancellation interval. Subtracting D from each of A, B, and C leaves just a short prefix of A, which we write mathematically as $\{subtract(x,\ D)\ |\ x\ in \{A,\ B,\ C\}\}$

$$
\begin{array}{l}
cancel(mod) \triangleq \\
\quad \text{LET } startTs \triangleq minStartTs(policyChs) \\
\quad \text{IN} \\
\quad \land\ pc < 5 \\
\quad \land\ cPolicyChs = \{\} \\
\quad \land\ mod.start\_ts < mod.end\_ts \\
\quad \land\ mod.start\_ts \geq startTs \\
\quad \land\ mod.end\_ts = maxEndTs(policyChs) \\
\quad \land\ pc' = pc + 1 \\
\quad \land\ \text{IF } mod.start\_ts = startTs \\
\qquad \text{THEN} \land\ policyChs' = \{[start\_ts \mapsto startTs,\ end\_ts \mapsto startTs]\} \\
\qquad\qquad \land\ exChs' = \{[start\_ts \mapsto startTs,\ end\_ts \mapsto startTs]\} \\
\qquad\qquad \land\ perilChs' = \{[start\_ts \mapsto startTs,\ end\_ts \mapsto startTs]\} \\
\qquad\qquad \land\ cPolicyChs' = policyChs \\
\qquad\qquad \land\ cExChs' = exChs \\
\qquad\qquad \land\ cPerilChs' = perilChs \\
\qquad \text{ELSE} \land\ policyChs' = \text{UNION } \{subtract(plcyCh,\ mod) : plcyCh \in policyChs\} \\
\qquad\qquad \land\ exChs' = \text{UNION } \{subtract(exCh,\ mod) : exCh \in exChs\} \\
\qquad\qquad \land\ perilChs' = \text{UNION } \{subtract(prlCh,\ mod) \quad : prlCh \in perilChs\} \\
\qquad\qquad \land\ cPolicyChs' = \text{UNION } \{union(plcyCh,\ mod) : plcyCh \in policyChs\} \\
\qquad\qquad \land\ cExChs' = \text{UNION } \{union(exCh,\ mod) : exCh \in exChs\} \\
\qquad\qquad \land\ cPerilChs' = \text{UNION } \{union(prlCh,\ mod) : prlCh \in perilChs\}
\end{array}
$$

From the point of view of characteristics an endorsement is the application of the subtract operator over the set of existing characteristics (set −)unioned with the the application of the (interval −)union operator, above, over the set of existing characteristics. The reader can verify this by drawing two sets of intervals on a piece of paper, following illustration for the cancel function, and visually doing the subtractions and unions. We can write the endorsement splitting process mathematically as
$\{subtract(x,\ D)\ |\ x\ in \{A,\ B,\ C\}\} + \{union(x,\ D)\ |\ x\ in \{A,\ B,\ C\}\}$

$$
\begin{array}{l}
endorsePolicy(mod) \triangleq \\
\quad \land\ pc < 5 \\
\quad \land\ cPolicyChs = \{\} \\
\quad \land\ mod.start\_ts < mod.end\_ts \\
\quad \land\ mod.start\_ts \geq minStartTs(policyChs) \\
\quad \land\ mod.end\_ts \leq maxEndTs(policyChs) \\
\quad \land\ pc' = pc + 1 \\
\quad \land\ policyChs' = \text{UNION } \{ \\
\qquad \text{UNION } \{subtract(policyCh,\ mod) : policyCh \in policyChs\}, \\
\qquad \text{UNION } \{union(policyCh,\ mod) : policyCh \in policyChs\}\} \\
\quad \land\ perilChs' = \text{UNION } \{
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad \textsc{union}\ \{subtract(prlCh,\ mod) : prlCh \in perilChs\}, \\
\qquad\qquad \textsc{union}\ \{union(prlCh,\ mod) : prlCh \in perilChs\}\} \\
\quad \wedge\ \textsc{unchanged}\ \langle exChs,\ cPolicyChs,\ cExChs,\ cPerilChs\rangle
\end{array}
$$

$$
\begin{array}{l}
endorseEx(mod) \triangleq \\
\quad \wedge\ pc < 5 \\
\quad \wedge\ cPolicyChs = \{\} \\
\quad \wedge\ mod.start\_ts < mod.end\_ts \\
\quad \wedge\ mod.start\_ts \geq minStartTs(exChs) \\
\quad \wedge\ mod.end\_ts \leq maxEndTs(exChs) \\
\quad \wedge\ pc' = pc + 1 \\
\quad \wedge\ exChs' = \textsc{union}\ \{ \\
\qquad \textsc{union}\ \{subtract(exCh,\ mod) : exCh \in exChs\}, \\
\qquad \textsc{union}\ \{union(exCh,\ mod) : exCh \in exChs\}\} \\
\quad \wedge\ perilChs' = \textsc{union}\ \{ \\
\qquad\qquad \textsc{union}\ \{subtract(prlCh,\ mod) : prlCh \in perilChs\}, \\
\qquad\qquad \textsc{union}\ \{union(prlCh,\ mod) : prlCh \in perilChs\}\} \\
\quad \wedge\ \textsc{unchanged}\ \langle policyChs,\ cPolicyChs,\ cExChs,\ cPerilChs\rangle
\end{array}
$$

$$
\begin{array}{l}
endorsePeril(mod) \triangleq \\
\quad \wedge\ pc < 5 \\
\quad \wedge\ cPolicyChs = \{\} \\
\quad \wedge\ mod.start\_ts < mod.end\_ts \\
\quad \wedge\ mod.start\_ts \geq minStartTs(perilChs) \\
\quad \wedge\ mod.end\_ts \leq maxEndTs(perilChs) \\
\quad \wedge\ pc' = pc + 1 \\
\quad \wedge\ perilChs' = \textsc{union}\ \{ \\
\qquad\qquad\qquad \textsc{union}\ \{subtract(prlCh,\ mod) : prlCh \in perilChs\}, \\
\qquad\qquad\qquad \textsc{union}\ \{union(prlCh,\ mod) : prlCh \in perilChs\}\} \\
\quad \wedge\ \textsc{unchanged}\ \langle policyChs,\ exChs,\ cPolicyChs,\ cExChs,\ cPerilChs\rangle
\end{array}
$$

Lastly, splitting characteristics for renewal is identical to the splitting process for reinstatements. It is just less general as for renewals there is always only a single added characteristics which is added in to the existing valid set.

$$
\begin{array}{l}
renew(mod) \triangleq \\
\quad \wedge\ pc < 5 \\
\quad \wedge\ cPolicyChs = \{\} \\
\quad \wedge\ mod.start\_ts < mod.end\_ts \\
\quad \wedge\ mod.start\_ts = maxEndTs(policyChs) \\
\quad \wedge\ policyChs' = policyChs \cup \{mod\} \\
\quad \wedge\ exChs' = exChs \cup \{mod\} \\
\quad \wedge\ perilChs' = perilChs \cup \{mod\} \\
\quad \wedge\ pc' = pc + 1 \\
\quad \wedge\ \textsc{unchanged}\ \langle cPolicyChs,\ cExChs,\ cPerilChs\rangle
\end{array}
$$

$$
\begin{array}{l}
term \triangleq \\
\quad \wedge\ pc = 5
\end{array}
$$

$\land$ UNCHANGED $\langle policyChs,\ exChs,\ perilChs,\ cPolicyChs,\ cExChs,\ cPerilChs,\ pc\rangle$

$Init \triangleq$ LET $ch \triangleq [start\_ts \mapsto 0,\ end\_ts \mapsto 3]$
IN $\land\ policyChs = \{ch\}$
$\land\ exChs = \{ch\}$
$\land\ perilChs = \{ch\}$
$\land\ cPolicyChs = \{\} \land cExChs = \{\} \land cPerilChs = \{\}$
$\land\ pc = 0$

$Next \triangleq$
$\lor\ term$
$\lor$ e $ch \in VCharacteristics$:
$\lor\ cancel(ch)$
$\lor\ reinstate(ch)$
$\lor\ renew(ch)$
$\lor\ endorsePeril(ch)$
$\lor\ endorseEx(ch)$
$\lor\ endorsePolicy(ch)$

$Spec \triangleq Init \land \Box[Next]_{\langle policyChs,\ exChs,\ perilChs,\ cPolicyChs,\ cExChs,\ cPerilChs,\ pc\rangle}$

---

* Modification History
* Last modified *Tue Dec* 15 15:14:20 *PST* 2020 by *ASUS*
* *Created Fri Oct* 16 13:25:40 *PDT* 2020 by *ASUS*

Chapter 3
Cancellation and Reinstatement

Seek not to have things happen as you choose them, but rather choose them to happen as they do.

Abstract During the course of a policy's lifetime it may be necessary to cancel a policy because invoices have not been paid or for other business reasons. These same policies may then need to be restored once the business reasons that caused the cancellation have been resolved. The frameworks cancellation and reinstatement functionality effects these policy changes.

3.1 Structure of Cancellations and Reinstatements

In our discussion of policies so far we dealt with the modifications

$$\{Creation, Endorsement, Renewal\}.$$

To this set we now add the modifications

$$\{Cancellation, Reinstatement\}.$$

These new modifications are similar to modifications we already know, with slight differences. Reinstatements are different from previous modifications in that they have two unique fields, auto_reinstate and deadline_timestamp, which we will describe shortly. Cancellations differ from previous modifications in that their state lifecycle only has the two events onCreate and onIssue. Below is an abstraction which describes the structural elements of cancellation and reinstatement data items.

MODULE $CancellationCtx1$

EXTENDS $Integers$, $Policy$

In this specification I only model cancellations and renewals processes, but I can not ignore the effects of other concurrent endorsement or renewals that may happen during those processes. I model possible endorsements and renewals with the constant $EndorsementOrRenewal$ which represents such a modification, in the accepted state.

CONSTANTS $EndorsementOrRenewal$

A reinstatement is type of modification. A reinstatement starts at some time and reactivates a policy upto the end timestamp of the cancellation it reinstates. A reinstatement has two specialized attributes. One, specified in configuration, is the $auto\_reinstate$: boolean variable. If this variable is true for a reinstatement a payment recieved on a finalized reinstatement should cause the reinstatement to be issued. The second attribute is a $deadline\_timestamp$ which is the last time at which the reinstatement can be issued. The deadline timestamp can be specified in configuration or it can be specified in $Api.reinstatement\_draft()$ and $Api.reinstatement\_update()$ request. If the value is never specified it defaults to infinity.

$Reinstatements \triangleq$ [
$type$: {"Reinstatement"},
$state$: $ModificationState$,
$start\_timestamp$: $TimeRange$,
$end\_timestamp$: $TimeRange$,
$auto\_reinstate$: BOOLEAN ,
$deadline\_ts$: $TimeRange$,
$product\_revision$: $0 \,..\, maxRevision$
]

A cancellation is also a special type of modification. In the concrete implementation cancellations have an additional attribute, reason, commented out below. The reason is one of many user configuration reason strings which the customer uses to descriminate the purpose of a cancellation. Additionally the system will alway define a define a cancellation reason of "lapse" if such a reason does not exist. The lapse reason is the reason given by the system to cancellations which result from invoices remaining unpaid past their grace period. Lapse cancellations are created by the scheduled lapse routines that run in the background of the API application

$Cancellations \triangleq$ [
$type$: {"Cancellation"},
$state$: $ModificationState$,
$start\_timestamp$: $TimeRange$,
$end\_timestamp$: $TimeRange$,
reason: String
$product\_revision$: $0 \,..\, maxRevision$
]

Cancellations and reinstatements, like endorsements and renewals, are recorded as they are created in the history of policy modifications. Some of these modifications will take effect, by being issued, others will never be issued. In all cases, though, modifications retain an identity to which documents and invoices can be attached. The characteristic of the modifications, which gives them a group commonality, is that they can operate to change liability of the parties to the contract

* Modification History
* Last modified $Tue\ Dec$ 15 15:53:25 PST 2020 by $ASUS$
* Last modified $Thu\ Jul$ 16 16:33:53 PDT 2020 by $marco$
* Created Sat Jun 27 21:12:19 PDT 2020 by $ASUS$

3.2 State Machine

With some understanding of the data in our two new modifications, we below, lay out actions on our policy in the usual state centric way.

MODULE $CancellationMch1$

EXTENDS $CancellationCtx1$, $Policy$, $Sequences$

CONSTANTS $policyTi$, $policyTf$, $reinstatementExpireDays$, $Documents$

VARIABLES $policy$, $step$, $mods$, $term$

Here, I model the total state by separating it into the variables policy and $mods$. The $mods$ variable is a sequential record of the issued modifications associated with the policy. Any accepted modifications associated with the policy are placed in the policy's $pending\_modification$ field. The choice to have a $pending\_modification$ field on the policy variable is driven by the need to express the invariant that only one modification can be in the Accepted state at any time.

A cancellation modification follows the state sequence: $Init -- (create) -> Created -- (issue) -> Issued \ \ldots$ A reinstatement modification follows one of the state sequences:
$Init -- (create) -> Created -- (accept) ->$ Accepted $-- (issue) -> Issued \ \ldots$
$Init -- (create) -> Created -- (accept) ->$ Accepted $-- (invalidate) \rightarrow$ Draft $\ldots$
$Init -- (create) -> Created -- (accept) ->$ Accepted $-- (expire) ->$ Expired

Conceptually, this development takes the view that only accept and issue events are essential to understanding state evolution, this view point reduces the state space without loss of generality.

Functions

$$prevMod \triangleq \text{LET } last \triangleq Len(mods) \text{ IN IF } last = 0 \text{ THEN } NoModification \text{ ELSE } mods[last]$$

$$pendingReinstatementExpired(p) \triangleq \text{LET } pending \triangleq p.pending\_modification \text{ IN } pending \in Reinstatements \land pending.deadline\_ts < step$$

Just a semantic addition for the reader. When this function appears in a guard, the reader should interpret it a a requirement to check for and invalidate any accepted modifications

$$invalidate(mod) \triangleq \text{TRUE}$$

Invariants

Below we define a few common invariants of the system.

Check a few variable fields to ensure that types are correct

$$InvType \triangleq \text{LET } pending \triangleq policy.pending\_modification \text{ IN } \land pending \in \text{UNION } \{Reinstatements, Cancellations, \{NoModification, EndorsementOrRenewal\}\} \land policy.state \in PolicyState$$

Policy Invariants are:
(1) a policy should have zero or one pending modifications.
(2) pending modifications must be in the $Finalized$ (Accepted) state.
(3) if a policy is $OffRisk$ then (a) its $start\_dt$ must equal its end_dt (b) the most recently issued modification must be a cancellations (c) there must be either no pending modifications or one of type $Reinstatment$. (4) all pending reinstatements must have a $deadline\_dt >$ now.

$$
\begin{array}{l}
InvPolicy \triangleq \\
\quad \text{LET } pending \triangleq policy.pending\_modification \\
\quad \text{IN } \land policy.original\_start\_ts = policyTi \\
\qquad \land pending \notin \{NoModification,\ EndorsementOrRenewal\} \implies \\
\qquad\qquad\qquad\qquad pending.state = \text{“Finalized”} \\
\qquad \land policy.state = \text{“OffRisk”} \implies (\\
\qquad\quad policy.original\_start\_ts = policy.effective\_end\_ts) \\
\qquad \land policy.state = \text{“OffRisk”} \implies prevMod \in Cancellations \\
\qquad \land policy.state = \text{“OffRisk”} \implies pending \in Reinstatements \cup \{NoModification\} \\
\qquad \land (\land pending \in Reinstatements \\
\qquad\quad \land pending.auto\_reinstate) \implies pending.deadline\_ts + 1 \geq step
\end{array}
$$

Modification Invariants are: (1) the first policy modification has a $start\_dt$ equals to the policy's original policy start dt
(2) all modifications have $start\_dt < end\_dt$
(3) once a cancellation is issued that modification can only be followed by a cancellation or reinstatement. All issued cancellations must have a matching, following reinstatement before any endorsements or renewals can be issued (4) a cancellation and its matching reinstatement must have equal end_dts

$$
\begin{array}{l}
InvMods \triangleq \\
\quad \text{LET } pending \triangleq policy.pending\_modification \\
\quad \text{IN } \land prevMod \neq NoModification \implies \\
\qquad\qquad prevMod.start\_timestamp \geq policy.original\_start\_ts \\
\qquad \land \forall\, idx \in 1\,..\,Len(mods) : (mods[idx].start\_timestamp < mods[idx].end\_timestamp) \\
\qquad \land \forall\, idx \in 2\,..\,Len(mods) : mods[idx] \in Reinstatements \implies \\
\qquad\qquad (\land mods[idx-1] \in Cancellations \\
\qquad\qquad\ \land mods[idx-1].end\_timestamp = mods[idx].end\_timestamp)
\end{array}
$$

As long as a policy is $OnRisk$, not fully canceled, endorsements or renewals for the policy may transition into an accepted state, creating a pending modification

$$
\begin{array}{l}
acceptGeneric \triangleq \\
\quad \land step < policyMaxTs \\
\quad \land policy.state = \text{“OnRisk”} \land policy.pending\_modification = NoModification \\
\quad \land policy' = [policy \text{ EXCEPT } !.pending\_modification = EndorsementOrRenewal] \\
\quad \land step' = step + 1 \\
\quad \land \text{UNCHANGED } \langle mods,\ term \rangle
\end{array}
$$

A cancellation can be created for any policy that is $OnRisk$. The start date specified by the cancellation should be within the coverage time span of the policy. Note that for the specification, cancellation create events do not result in additions to the policy's modifications set. This addition does happen in the concrete implementation, however, we elide the effect here as maintaining the state in the policy record along with a list of issued modifications is sufficient to validate system properties.

$$
\begin{array}{l}
createCancellation(mod) \triangleq \\
\quad \land\ step < policyMaxTs \\
\quad \land\ policy.state = \text{“OnRisk”} \\
\quad \land\ mod.type = \text{“Cancellation”} \land mod.state = \text{“Created”} \\
\quad \land\ mod.start\_timestamp < policy.effective\_end\_ts \\
\quad \land\ mod.start\_timestamp \geq policy.original\_start\_ts \\
\quad \land\ \neg pendingReinstatementExpired(policy) \qquad \text{assumed} \\
\quad \land\ step' = step + 1 \\
\quad \land\ \textsc{unchanged}\ \langle policy,\ mods,\ term \rangle
\end{array}
$$

A cancellation can be issued whenever its policy is $OnRisk$, and the cancellation date is within the time span of the policy. If any endorsement or renewals are pending then the cancellation invalidates those endorsements / renewals. In the concrete implementation the cancellation can also block when there are existing accepted modifications. For maximal generality, in this development I always assume the cancellation will invalidate.

$$
\begin{array}{l}
issueCancellation(mod) \triangleq \\
\quad \textsc{let}\ pending \triangleq policy.pending\_modification \\
\quad \textsc{in}\ \ \land\ step < policyMaxTs \\
\qquad \land\ policy.state = \text{“OnRisk”} \\
\qquad \land\ mod.type = \text{“Cancellation”} \land mod.state = \text{“Created”} \\
\qquad \land\ mod.end\_timestamp = policy.effective\_end\_ts \qquad \text{assumed} \\
\qquad \land\ policy.original\_start\_ts \leq mod.start\_timestamp \\
\qquad \land\ policy.effective\_end\_ts > mod.start\_timestamp \\
\qquad \text{actions} \\
\qquad \land\ invalidate(pending) \\
\qquad \land\ \textsc{case}\ policy.original\_start\_ts = mod.start\_timestamp \rightarrow \\
\qquad\quad \land\ \ policy' = [policy\ \textsc{except} \\
\qquad\qquad\qquad\qquad !.state = \text{“OffRisk”}, \\
\qquad\qquad\qquad\qquad !.pending\_modification = NoModification, \\
\qquad\qquad\qquad\qquad !.effective\_end\_ts = mod.start\_timestamp] \\
\qquad\quad \square\textsc{other} \rightarrow \\
\qquad\quad\ \ \land\ policy' = [policy\ \textsc{except} \\
\qquad\qquad\qquad\qquad !.pending\_modification = NoModification, \\
\qquad\qquad\qquad\qquad !.effective\_end\_ts = mod.start\_timestamp] \\
\qquad \land\ mods' = Append(mods,\ [mod\ \textsc{except}\ !.state = \text{“Issued”}]) \\
\qquad \land\ step' = step + 1 \\
\qquad \land\ \textsc{unchanged}\ \langle term \rangle
\end{array}
$$

The legacy lapse action is only active if the $cancellations.json$ configuration items are not present in the policy's product configuration. We will assume this is the case for the policy we are considering here, so legacy lapse will never execute.

$legacyLapse(mod) \triangleq$ FALSE

As with legacy lapse, above, legacy reinstatements will be disabled if a $cancellations.json$ file exists in the asset bundle.

$legacyReinstatement(mod) \triangleq$ FALSE

If a policy's most recent modification is a cancellation then then we can begin a draft reinstatement to reinstate it. The draft must specify an effective start date within the timespan of the cancellation which preceded it, and an end date equal to the preceeding cancellation. As with $createCancellation$ we don't model the addition to the modification set, though that would be part of the concrete implementation. Also, in the implementation, drafts are matched 1:1 with a cancellation and that matching is immutable once established.

$$
\begin{aligned}
&draftReinstatement(mod) \triangleq \\
&\quad \text{LET } pending \triangleq policy.pending\_modification \\
&\quad \text{IN} \quad \land\ step < policyMaxTs \\
&\qquad \land\ mod.type = \text{“Reinstatement”} \land mod.deadline\_ts > step \\
&\qquad \land\ mod.start\_timestamp \geq policy.effective\_end\_ts \\
&\qquad \land\ mod.start\_timestamp < mod.end\_timestamp \\
&\qquad \land\ mod.deadline\_ts + 1 \ < step \\
&\qquad \land\ prevMod \in Cancellations \land prevMod.end\_timestamp = mod.end\_timestamp \\
&\qquad \land\ \neg pendingReinstatementExpired(policy) \qquad \text{assumed} \\
&\qquad \land\ step' = step + 1 \\
&\qquad \land\ \text{UNCHANGED } \langle policy,\ mods,\ term \rangle
\end{aligned}
$$

If a reinstatement exists past its $deadline\_ts$ without being issued then it expires. On expiration, the reinstatement is removed as a pending modification; its documents are discarded; and its invoices are settled. Note that it is not possible for a draft reinstatement to transition to the expired state even if $deadline\_dt <$ now. In this case the draft reinstatement will be unable to transition to the accept state and its $deadline\_dt$ will have to be updated via the update API to take part in further transitions.

$$
\begin{aligned}
&expireReinstatement \triangleq \\
&\quad \text{LET } pending \triangleq policy.pending\_modification \\
&\quad \text{IN} \quad \land\ step < policyMaxTs \\
&\qquad \land\ pendingReinstatementExpired(policy) \\
&\qquad \land\ invalidate(pending) \\
&\qquad \land\ policy' = [policy \text{ EXCEPT } !.pending\_modification = NoModification] \\
&\qquad \land\ step' = step + 1 \\
&\qquad \land\ \text{UNCHANGED } \langle mods,\ term \rangle
\end{aligned}
$$

In order to accept a reinstatement the start date of the reinstatement must be within the time span of the cancellation it restores. There must also be (1) no other accepted modification pending and (2) (in this model) the most recently issued modification must be a $Cancellation$. In the concrete implementation, we dont require that a reinstatement follow its cancellation. We are slightly more relaxed on only require that all cancelations must be reinstated before we can quote or accept endorsements or renewals

$$
\begin{aligned}
&acceptReinstatement(mod) \triangleq \\
&\quad \text{LET } pending \triangleq policy.pending\_modification \\
&\quad \text{IN} \quad \land\ step < policyMaxTs \\
&\qquad \land\ pending = NoModification
\end{aligned}
$$

$\land\ mod.type$ = "Reinstatement" $\land\ mod.state$ = "Created" $\land\ mod.deadline\_ts > step$
$\land\ mod.start\_timestamp \geq policy.effective\_end\_ts$
$\land\ mod.start\_timestamp < mod.end\_timestamp$
$\land\ prevMod \in Cancellations \land prevMod.end\_timestamp = mod.end\_timestamp$
$\land\ policy' = [policy$ EXCEPT
$!.pending\_modification = [mod$ EXCEPT $!.state$ = "Finalized"$]]$
$\land\ step' = step + 1$
$\land$ UNCHANGED $\langle mods, term\rangle$

Issuing a reinstatement reactivates the policy restoring the end date and coverage as of the previous cancellation. Note that with cancellations and reinstatements it becomes possible to have gaps in a policy's coverage. These gaps are new to the framework and are of importance for financial calculations and claim coverage. The gaps are recorded in the database in the policy's currently active $policy\_characteristics$ set. These $policy\_characteristics$ previously would have been sequential, meeting, and non overlapping. Now they must be, just, sequential and not overlapping.

$issueReinstatement \triangleq$
LET $pending \triangleq policy.pending\_modification$
IN $\land\ step < policyMaxTs$
$\land\ pending \in Reinstatements$
$\land\ \neg pendingReinstatementExpired(policy)$ assumed
$\land\ policy' = [policy$ EXCEPT $!.state$ = "OnRisk",
$!.pending\_modification = NoModification]$
$\land\ mods' = Append(mods, [pending$ EXCEPT $!.state$ = "Issued"$])$
$\land\ step' = step + 1$
$\land$ UNCHANGED $\langle term\rangle$

When we invalidate a reinstatement we return the modification to the draft state. We discard all documents associated with the reinstatement and then settle any invoices.

$invalidateReinstatement \triangleq$
LET $pending \triangleq policy.pending\_modification$
IN $\land\ step < policyMaxTs$
$\land\ pending \in Reinstatements \land step \leq pending.deadline\_ts$
$\land\ invalidate(pending)$
$\land\ policy' = [policy$ EXCEPT $!.pending\_modification = NoModification]$
$\land\ step' = step + 1$
$\land$ UNCHANGED $\langle mods, term\rangle$

If we receive a payment when we have a finalized reinstatement in the pending modifications set then the policy can be issued if $autoReinstate$ is on. The operation below is just a placeholder to mark the existence of the auto reinstate functionality. It is not included in the model as inclusion would just repeat the $issueReinstatement$ operation above.

$payInvoiceReinstatement \triangleq$
LET $mod \triangleq policy.pending\_modification$
IN $\land\ step < policyMaxTs$
$\land\ mod.auto\_reinstate =$ TRUE
$\land\ issueReinstatement$

$doStep \triangleq$

```
    LET mod ≜ policy.pending_modification
    IN  ∧ step < policyMaxTs
        ∧ IF ( ∧ mod ∉ {NoModification, EndorsementOrRenewal}
               ∧ mod.type = "Reinstatement"
               ∧ step > mod.deadline_ts)
          THEN expireReinstatement
          ELSE ∧ step' = step + 1
               ∧ UNCHANGED ⟨policy, term, mods⟩

doTerm ≜
    ∧ step = policyMaxTs
    ∧ term' = TRUE
    ∧ UNCHANGED ⟨policy, step, mods⟩
```

Testing notes: With configurations that calculate all charges as a percentage of premium (not all), there are many sequences of cancellation and reinstatment operations that should leave the total charges invariant. For example, it should be the case that

Issue; Cancel; Reinstate; = Issue

These sorts of relationship are useful for checking financial calculation routines.

```
Init ≜
    ∧ step = 0
    ∧ policy = [
        original_start_ts ↦ policyTi,
        effective_end_ts ↦ policyTf,
        billing_policy ↦ "BySystemOnFinalize",
        state ↦ "OnRisk",
        revision_start_timestamps ↦ {⟨0, policyTi⟩},
        renewal_start_timestamps ↦ {},
        pending_modification ↦ NoModification]
    ∧ mods = ⟨⟩
    ∧ term = FALSE

Next ≜
    ∨ acceptGeneric
    ∨ ∃ m ∈ Cancellations : createCancellation(m)
    ∨ ∃ m ∈ Cancellations : issueCancellation(m)
    ∨ ∃ m ∈ Cancellations : legacyLapse(m)
    ∨ ∃ m ∈ Reinstatements : draftReinstatement(m)
    ∨ ∃ m ∈ Reinstatements : acceptReinstatement(m)
    ∨ expireReinstatement
    ∨ issueReinstatement
    ∨ invalidateReinstatement
    ∨ doStep
    ∨ doTerm
```

$$\vee\ (term = \text{TRUE} \wedge \text{UNCHANGED}\ \langle policy,\ step,\ mods,\ term\rangle)$$

$$Spec\ \triangleq\ Init \wedge \Box[Next]_{\langle policy,\ step,\ mods,\ term\rangle}$$

* Modification History
* Last modified $Tue\ Oct$ 12 08:20:21 PDT 2021 by $marco$
* Last modified $Tue\ Dec$ 15 12:15:46 PST 2020 by $ASUS$
* $Created$ Sat Jun 27 21:12:50 PDT 2020 by $ASUS$

3.3 Interaction with other Modifications

The development of quotes for Endorsement and Renewals and the development of Cancellation and Reinstatements functionality partially overlapped and so it turns out that the two functionalities interact in a less than perfectly integrated way.
Before we can discuss potential interactions of the two functionalities, I will quickly review the essentials of quotes. The purpose of quotes is to price a potential modification given a policy in some state. And the way that given policy state is described colloquially is as "a policy where the last applied modifications is x", or using the terminology in the code as "a policy with basis x". The framework realization of both of these informal descriptions is a field in both the endorsement and renewal tables called quoted_policy_basis which holds a policy modification locator. The quote code would then, conceptually, have mechanism to allow a quoted endorsement or renewal to progress to the accepted state only when $modification.quote\_policy\_base = lastModification(policy).locator$. Now in fact the actual implementation is not as uniform and explicit as I have described, but the brief explanation here is still a good way to conceptualize and growing complexity may force uniformity and explicitness in the long term anyways.
The challenge for cancellation and reinstatements is then to prevent situations where a quote is made, some arbitrary cancellation reinstatement sequence changes the policy and then the quote becomes accepted and issued. There are three mechanisms that prevent this occurrence that have been added to the cancellation / reinstatement state machine. First whenever a policy is changed by cancellation or reinstatement we invalidate all policy quotes. By definition if the policy has changed and the quote could not anticipate that change then the basis of the quote must be then be invalid. Second, if a policy has any outstanding cancellations then no renewals can be created. This prevents any renewals from effecting the policy until all cancellation reinstatement sequences are completed. Lastly endorsement quotes are not permitted when a reinstatement is in the accepted state. This prevents quotes from being made on policies that are about to change.

Chapter 4
Scheduled Jobs

I call it "The Question." Obviously it should be applied as much to one's own thinking as to others'. It consists of asking in your own mind, on hearing any scientific explanation or theory put forward, "But sir, what experiment could disprove your hypothesis?"

Abstract

4.1 Grace and Lapse

In addition to cancellations that can happen through the API, it's also possible for policies to be canceled automatically by the system. This automated cancellation happens when a policy has outstanding invoices which have not been paid. The system scheduled grace job flags outstanding invoices and the scheduled lapse job, after some time, will eventually cancel the policies associated with those flagged invoices. Below is a simplified model of the process, which details the essential points of understanding for the jobs.

MODULE $AutoGraceLapseInvoiceCtx0$

EXTENDS $Integers$
CONSTANTS $Time$

The invoice table maintains the following variables to track the settlement state of an invoice

$SettlementStatus \triangleq$ {"Outstanding", "Settled"}

$SettlementType \triangleq$ {"ZeroDue", "WrittenOff", "Invalidated", "Paid", "Null"}

A simplified representation of invoices

$$
\begin{aligned}
&Invoice \triangleq [\\
&\quad totalDue : \{10, -10\},\\
&\quad startDt : Time,\\
&\quad dueDt : Time,\\
&\quad settlementStatus : SettlementStatus,\\
&\quad settlementType : SettlementType\\
&]
\end{aligned}
$$

A simplified policy representation

$Policy \triangleq [$
$startDt : Time,$
$endDt : Time$
$]$

A simplified $grace\_period$ record representation

$Grace \triangleq [$
$startDt : Time,$
$endDt : Time$
$]$

A simplified policy modification representation

$Modification \triangleq [$
$startDt : Time,$
$endDt : Time$
$]$

* $Modification$ History
* Last modified $Wed\ Dec$ 23 15:27:56 PST 2020 by $ASUS$
* Created $Wed\ Nov$ 18 17:57:57 PST 2020 by $ASUS$

MODULE $AutoGraceLapseInvoiceMch0$

EXTENDS $AutoGraceLapseInvoiceCtx0$, $FiniteSets$, TLC
CONSTANTS $endTime$, $graceDelta$

VARIABLES $invsIssued$, $policy$, $gracesUnsettled$, $time$

Here we model a simple yearly policy to illustrate the interactions between policy modifications, the scheduled grace/lapse jobs, payments, and payment reversal. Since, with yearly policies, all invoices are generated at the time of modification, we have simplified away the complicating factor that scheduled invoicing can, in the most general case, generate outstanding invoices at any time. Therefore this model is a simplest case analysis.

The possible combinations of settlement status and invoice type

$$\begin{aligned}
&InvInvoices \triangleq \\
&\quad \land \forall\, i \in invsIssued : \\
&\qquad \land\ i.settlementStatus = \text{“Outstanding”} \implies i.settlementType = \text{“Null”} \\
&\qquad \land\ i.settlementStatus = \text{“Settled”} \implies i.settlementType \in \{ \\
&\qquad\quad \text{“ZeroDue”, “WrittenOff”, “Invalidated”, “Paid”} \\
&\qquad \}
\end{aligned}$$

At the time a policy is fully cancelled we insist that all invoices be settled. We can not establish that as an invariant however as with payment reversals there are no restrictions on what invoices can be subsequently flipped into the outstanding state.

$InvPolicy \triangleq policy.startDt \leq policy.endDt$

There must be at most one unsetteled grace record per policy. This property is due to the data definition of grace. A policy is defined as in grace when it has a record in the grace table that is unsettled.

$InvGraceRecs \triangleq$
$\quad \land Cardinality(gracesUnsettled) \leq 1$
$\quad \land \forall g \in gracesUnsettled : g \in Grace$

If any outstanding policy invoices exist, then there must be a unsettled grace record for the policy. If there is an unsettled grace record then there must be an outstanding policy invoice.

$InvInvoicesGraceRecs \triangleq$
$\quad \land \{i \in invsIssued : \land i.settlementStatus = \text{“Outstanding”}$
$\qquad\qquad\qquad\qquad \land time > i.dueDt\} \neq \{\} \implies gracesUnsettled \neq \{\}$
$\quad \land gracesUnsettled \neq \{\} \implies \exists i \in invsIssued :$
$\qquad\qquad\qquad\qquad\qquad \land i.settlementStatus = \text{“Outstanding”}$
$\qquad\qquad\qquad\qquad\qquad \land i.dueDt \leq time$

The scheduled grace job tries to find policies that (1) dont have any associated, unsettled grace records in the *grace_period* table and (2) do have outstanding invoices that are past their due date. If it finds such records it adds them to the *grace_period* table, which roughly can be thought of as a queue of jobs which may or may not get to the top of the queueu and cause a lapse (cancellation)

$autoGraceCanRun(tm) \triangleq$
$\quad \land gracesUnsettled = \{\}$
$\quad \land \exists i \in invsIssued : i.dueDt = tm \land i.settlementStatus = \text{“Outstanding”}$

The scheduled lapse job looks for unsettled records in the *grace_period* period and cancels the associated policy if the end of the grace period $<$ now.

$autoLapseCanRun(tm) \triangleq \exists g \in gracesUnsettled : g.endDt = tm$

$autoJobCanRun(tm) \triangleq$
$\quad \lor autoGraceCanRun(tm)$
$\quad \lor autoLapseCanRun(tm)$

the definition of containment for a point within a closed - open interval.

$contains(interval,\ t) \triangleq interval.startDt \leq t \land t < interval.endDt$

$overdueInvoices(invs) \triangleq \{i \in invs :$
$\qquad \land i.settlementStatus = \text{“Outstanding”}$
$\qquad \land i.dueDt \leq time\}$

$minOf(S) \triangleq \text{CHOOSE } x \in S : \forall y \in S : x \leq y$
$oneOf(S) \triangleq \text{CHOOSE } x \in S : \text{TRUE}$
$max(t1,\ t2) \triangleq \text{IF } t1 > t2 \text{ THEN } t1 \text{ ELSE } t2$

Extension is a generic operation extends a policy range, a renewal or reinstatement.

$extend(m) \triangleq$
$\quad \land \neg autoJobCanRun(time)$
$\quad \land time < endTime$
$\quad \land m.startDt = policy.endDt$
$\quad \land m.startDt < m.endDt$

$\land\ policy' = [policy \text{ EXCEPT } !.endDt = m.endDt]$
$\land\ invsIssued' = invsIssued \cup \{[$
$uid \mapsto time + 1,$
$totalDue \mapsto m.endDt - m.startDt,$
$startDt \mapsto m.startDt,$
$endDt \mapsto m.endDt,$
$dueDt \mapsto max(m.startDt + 1,\ time + 1),$
$settlementStatus \mapsto$ "Outstanding",
$settlementType \mapsto$ "Null"
$]\}$
$\land\ time' = time + 1$
$\land$ UNCHANGED $\langle gracesUnsettled \rangle$

Reduce is a generic operation that reduces a policy range, a cancellation. For the reduction I dont add any negative invoices as negative invoices dont have a grace date and so are not picked up by the automated jobs, which I am modeling here.

$reduce(m) \triangleq$
$\land\ \neg autoJobCanRun(time)$
$\land\ time < endTime$
$\land\ m.startDt \geq policy.startDt$
$\land\ m.endDt = policy.endDt$
$\land\ m.startDt < m.endDt$
$\land\ policy' = [policy \text{ EXCEPT } !.endDt = m.startDt]$
$\land\ time' = time + 1$
$\land$ UNCHANGED $\langle invsIssued,\ gracesUnsettled \rangle$

Change is a generic operations that updates policy coverage, an endorsement.

$change(m) \triangleq$
$\land\ \neg autoJobCanRun(time)$
$\land\ time < endTime$
$\land\ policy.startDt \leq m.startDt \land m.startDt < policy.endDt$
$\land\ m.endDt = policy.endDt$
$\land\ m.startDt < m.endDt$
$\land\ invsIssued' = invsIssued \cup \{[$
$uid \mapsto time + 1,$
$totalDue \mapsto m.endDt - m.startDt,$
$startDt \mapsto m.startDt,$
$endDt \mapsto m.endDt,$
$dueDt \mapsto max(m.startDt + 1,\ time + 1),$
$settlementStatus \mapsto$ "Outstanding",
$settlementType \mapsto$ "Null"
$]\}$
$\land\ time' = time + 1$
$\land$ UNCHANGED $\langle policy,\ gracesUnsettled \rangle$

We have discussed grace processing above. For emphasis here, note that the guard clause below only adds a grace record if there are no unsettled records associated with the policy.

$doAutoGrace(inv) \triangleq$
$\land\ time < endTime$
$\land\ gracesUnsettled = \{\}$
$\land\ inv.settlementStatus =$ "Outstanding" $\land\ inv.dueDt = time$
$\land\ gracesUnsettled' = \{[$
$invUid \mapsto inv.uid,$
$startDt \mapsto inv.dueDt,$
$endDt \mapsto inv.dueDt + graceDelta$
$]\}$
$\land\ time' = time + 1$
$\land$ UNCHANGED $\langle policy, invsIssued \rangle$

Auto lapse will cancel the whole policy and that requires settling all the invoices, not just the particular invoice, associated with the grace record. This prevents multiple loops through grace/lapse and also established the grace invariant. In the implementation the invoice cancellation details are handled in the $CancellationsService$.

In the second branch of the IF statement below we handle the case where a policy is to lapse after the policy has ended. We really cant cancel the policy as cancellation is a reduction in policy extent. So for this case we just settle all outstanding invoices and clear grace. Clearing all the outstanding invoices helps us maintain the processing invariant that $gracesUnsettled = \{\} \implies$ there are no outstanding invoices.

$doAutoLapse(g) \triangleq$
$\land\ time < endTime$
$\land\ g.endDt = time$
$\land$ IF $g.endDt < policy.endDt$
$invalidates =$ invoices written off if grace/lapsed looped multiple times at t
THEN LET $invalidates \triangleq \{i \in invsIssued : \land\ i.settlementStatus =$ "Outstanding"
$\land\ (\lor\ i.uid = g.invUid$
$\lor\ i.dueDt + graceDelta \leq time)\}$
$overdues \triangleq \{i \in invsIssued : \land\ i.settlementStatus =$ "Outstanding"
$\land\ i.dueDt \leq time$
$\land\ i.dueDt + graceDelta > time\}$
IN $\land\ policy' = [policy$ EXCEPT $!.endDt = g.endDt]$
$\land\ invsIssued' = (invsIssued \setminus invalidates)$
$\cup$
$\{[i$ EXCEPT $!.settlementStatus =$ "Settled",
$!.settlementType =$ "WrittenOff"$] : i \in invalidates\}$
$\land\ gracesUnsettled' =$ IF $overdues = \{\}$
THEN $\{\}$
ELSE LET $anOverdue \triangleq oneOf(overdues)$
IN $\{[$
$invUid \mapsto anOverdue.uid,$
$startDt \mapsto anOverdue.dueDt,$
$endDt \mapsto anOverdue.dueDt + graceDelta$
$]\}$
$\land\ time' = time + 1$

```
    ELSE LET invalidates ≜ {i ∈ invsIssued : i.settlementStatus = "Outstanding"}
         IN  ∧ Assert(invalidates ≠ {}, "bad empty invalidates set")
             ∧ invsIssued' = (invsIssued \ invalidates)
                             ∪
                             {[i EXCEPT !.settlementStatus = "Settled",
                                        !.settlementType = "WrittenOff"] : i ∈ invalidates}
             ∧ gracesUnsettled' = {}
             ∧ time' = time + 1
             ∧ UNCHANGED ⟨policy⟩
```

When payments come in, the framework checks to see if the policy is in grace. If it is and the current payment will settle all outstandiing invoices, then the payment routines can also settle the grace record, bringing the policy out of grace. This processing rule tends to be hard to understand for engineers, so remember paying one invoice generally does not get a policy out of grace, all outstanding invoices have to be paid.

```
doPayment(inv) ≜
    ∧ ¬autoJobCanRun(time)
    ∧ time < endTime
    ∧ inv.settlementStatus = "Outstanding"
    ∧ invsIssued' = (invsIssued \ {inv}) ∪ {
          [inv EXCEPT !.settlementStatus = "Settled", !.settlementType = "Paid"]
          }
    ∧ time' = time + 1
    ∧ LET overdueInvs ≜ overdueInvoices(invsIssued \ {inv})
      IN  CASE overdueInvs = {} ∧ gracesUnsettled = {} →
                   UNCHANGED ⟨policy, gracesUnsettled⟩
            □  overdueInvs = {} ∧ gracesUnsettled ≠ {} →
                   ∧ gracesUnsettled' = {}
                   ∧ UNCHANGED policy
            □  overdueInvs ≠ {} →
                   UNCHANGED ⟨policy, gracesUnsettled⟩
```

When we reverse payment we want to (1) reset the due date, and grace date of the affected invoice. This prevents reversals of older invoices from immediately causing the policy to go into grace and immediately after be cancelled. This is a side effect that customers will find hard to understand. Presumably if a customer has to pay some time must be given to them to do so before cancelling their policy. (2) We dont want to reverse payments on policies that are fully cancelled. To reinterate a similiar thought above, it does not make sense in that a customer would have to suddenly owe money on a previous invoice and yet the coverage that invoice is based on is not being offered to them. Customer but also the code will expect this convention so if it is violated the automated lapse code will refuse to cooperate and someone, maybe you, will get a late night call.

```
doRevPayment(inv) ≜
    ∧ ¬autoJobCanRun(time)
    ∧ time < endTime
    ∧ inv.settlementStatus = "Settled"
    ∧ invsIssued' = (invsIssued \ {inv}) ∪ {
          [inv EXCEPT !.settlementStatus = "Outstanding",
```

```
                                !.settlementType = "Null",
                                !.dueDt = max(inv.dueDt, time + 1)]
            }
    ∧ time′ = time + 1
    ∧ UNCHANGED ⟨policy, gracesUnsettled⟩


tick ≜
    ∧ ¬autoJobCanRun(time)
    ∧ time < endTime
    ∧ time′ = time + 1
    ∧ UNCHANGED ⟨invsIssued, policy, gracesUnsettled⟩

term ≜
    ∧ time = endTime
    ∧ UNCHANGED ⟨invsIssued, policy, gracesUnsettled, time⟩

Init ≜
    ∧ ∃ p ∈ Policy :
            ∧ p.startDt < p.endDt
            ∧ policy = p
    ∧ time = 0
    ∧ invsIssued = {[
            uid ↦ 0,
            totalDue ↦ policy.endDt − policy.startDt,
            startDt ↦ policy.startDt,
            endDt ↦ policy.endDt,
            dueDt ↦ policy.startDt + 1,
            settlementStatus ↦ "Outstanding",
            settlementType ↦ "Null"
       ]}
    ∧ gracesUnsettled = {}

Next ≜
    ∨ ∃ m ∈ Modification :
            ∨ change(m)
            ∨ extend(m)
            ∨ reduce(m)
    ∨ ∃ inv ∈ invsIssued :
            ∨ doAutoGrace(inv)
            ∨ doPayment(inv)
            ∨ doRevPayment(inv)
    ∨ ∃ g ∈ gracesUnsettled : doAutoLapse(g)
    ∨ term

Spec ≜ Init ∧ □[Next]⟨invsIssued, policy, gracesUnsettled, time⟩
```

* *Modification* History
* Last modified *Tue Sep* 21 13:23:45 *PDT* 2021 by *ASUS*
* Created *Wed Nov* 18 17:58:19 *PST* 2020 by *ASUS*

4.2 Document Consolidation

During a modification's transition into the Accepted state, documents associated with that modification are generated. The documents are generated from templates written in Liquid code and the templates output either a text or HTML formatted string. If desired the system can take this string output and convert it into a PDF document. The final form of the document is then stored in Amazon S3.
In addition to generating documents some customers would like to consolidate a subset of the existing documents as well, and this has let to the development of the document consolidation plugin and associated consolidation routines. The main consideration in developing the consolidation routines has been to avoid generating the consolidated documents synchronously with the accept transition. Document generation has historically been a large part of the accept transition's service time so there is some trepidation around adding more to what already exists. There is also reason to believe that some customers may have consolidation tasks that are one or two orders of magnitude larger than the usual size of the document generation tasks that currently run semi-synchronously. These two considerations have let to an asynchronous consolidate implementation which is discussed below.

MODULE $DocConsolidateCtx0$

EXTENDS $Integers$

The minimal status codes necessary to track the progress of async jobs.
$ProcessStatus \triangleq$ {"InProcess", "Error", "Success"}

We specify each document as being either a single document or a composite document made up of multiple single documents.
$CollationType \triangleq$ {"Single", "Multiple"}

There are two processes. One that generates a document in some format and a second type of process that takes existing pdf documents and consolidates them into a single pdf document.
$Process \triangleq$ {"DocGen", "DocConsolidate"}

A document request can create either a single document or a consolidated document in large object storage. A single document create request will have an empty $componentDocs$ set. A consolidated document request will supply a list of existing document locators that reference single documents.

$DocRequest \triangleq$ [
    $id : Int$,
    $process : Process$,
    $componentDocs :$ SUBSET Int,
    $processCount : Int$

]

Record, below, represents a document record that has been added to the $policy\_document$ table in the database. The record, here, includes just the fields needed by this document processing specification.

$Record \triangleq [$
$id : Int,$
$collationType : CollationType,$
$processStatus : ProcessStatus$
]

* Modification History
* Last modified $Tue\ Oct$ 12 09:21:02 PDT 2021 by $marco$
* Last modified $Wed\ Oct$ 06 12:57:41 PDT 2021 by $ASUS$
* $Created\ Fri\ Sep$ 17 18:39:27 PDT 2021 by $ASUS$

MODULE $DocConsolidateMch0$

EXTENDS $DocConsolidateCtx0$, $Sequences$, $FiniteSets$, TLC

CONSTANTS $maxPc$

VARIABLES pc, $docNo$, $dbDocs$, dbQ

This specification details a future vision of document generation where single as well as composite documents are generated asyncronously. The pc variable is a 'program counter' representing the progresion of time or execution steps. The $docNo$ variable creates uuid for documents. The $dbDocs$ variable holds data which represent records in the $policy\_document$ table. And the dbQ variable holds data which represent records in the $policy\_document\_queue$ table.

Invariants

The total number of documents, that should be generated, must equal the number of documents in the $policy\_document$ table with $process\_status$ in $\{Success, Error\}$ plus the number of items in the processing queue.

$InvNoRequestLoss \triangleq$
$\land\ Cardinality(dbDocs) = docNo$
$\land\ Cardinality(\{d \in dbDocs : d.processStatus \in \{\text{“Success”}, \text{“Error”}\}\}) + Len(dbQ) = docNo$

When the processing queue is empty, then all records in the $policy\_document$ table must have a $process\_status$ in $\{Success, Error\}$

$InvQueueEmpty \triangleq$
$\land\ Len(dbQ) = 0 \implies \forall\, d \in dbDocs : d.processStatus \in \{\text{“Success”}, \text{“Error”}\}$

The number of records in the processing queue must always equals the number of records in the $policy\_document$ table that have a $process\_status$ equal to $InProcess$

$InvQueueNotEmpty \triangleq$
$\land\ Len(dbQ) = Cardinality(\{d \in dbDocs : d.processStatus = \text{“InProcess”}\})$

Helper functions

$endDocGen \triangleq pc \geq maxPc$

```
componentDocsProcessing(head) ≜
    LET ps ≜ {d ∈ dbDocs : d.id ∈ head.componentDocs ∧ d.processStatus = "InProcess"}
    IN  Cardinality(ps) > 0

componentDocsAllSuccessful(head) ≜
    LET ps ≜ {d ∈ dbDocs : d.id ∈ head.componentDocs ∧ d.processStatus = "Success"}
    IN  Cardinality(head.componentDocs) = Cardinality(ps)

componentDocsHaveErrors(head) ≜
    LET ps ≜ {d ∈ dbDocs : d.id ∈ head.componentDocs ∧ d.processStatus = "Error"}
    IN  Cardinality(ps) > 0
```

Transition functions

Initiate processing to create a single document. A document generation request is added to the processing queue and a document record is added to the *policy_document* table in the database. The record in the *policy_document* table allows clients to observe that some event has created a document, but in an sync world, which we are modeling, there may not yet be an available url for the document.

```
enqueueSingleDoc ≜
    ∧ ¬endDocGen
    ∧ dbDocs' = dbDocs ∪ {[
            id ↦ docNo,
            collationType ↦ "Single",
            processStatus ↦ "InProcess"
        ]}
    ∧ dbQ' = Append(dbQ, [
            id ↦ docNo,
            process ↦ "DocGen",
            componentDocs ↦ {},
            processCount ↦ 0
        ])
    ∧ docNo' = docNo + 1
    ∧ pc' = pc + 1
```

Initiate processing to create a composite document. The composite document request contains a list of component, single documents to combine into the composite. The component documents may or may not have been generated by their own processing requests.

```
enqueueMultipleDoc(ids) ≜
    ∧ ¬endDocGen
    ∧ Cardinality(ids) > 1
    ∧ dbDocs' = dbDocs ∪ {[
            id ↦ docNo,
            collationType ↦ "Multiple",
            processStatus ↦ "InProcess"
        ]}
    ∧ dbQ' = Append(dbQ, [
            id ↦ docNo,
```

```
                process ↦ "DocConsolidate",
                componentDocs ↦ ids,
                processCount ↦ 0
        ])
    ∧ docNo′ = docNo + 1
    ∧ pc′ = pc + 1
```

Process a single document successfully. For this simple processing scenario we just update the record in the *policy_document* table to have a successful process status, and remove the process record from queue.

```
unqueueSingleOk(head) ≜
    ∧ head.processCount ≤ 2
    ∧ LET rec ≜ CHOOSE d ∈ dbDocs : d.id = head.id
      IN  ∧ dbDocs′ = UNION {
                (dbDocs \ {rec}),
                {[rec EXCEPT !.processStatus = "Success"]}
            }
          ∧ dbQ′ = Tail(dbQ)
          ∧ pc′ = pc + 1
          ∧ UNCHANGED ⟨docNo⟩
```

The processing for a composite document is similar to a single document, but there is the added condition that all of the component documents must be finished successfully as a precondition for the composite doc to both be unqueued and to finish successfully.

```
unqueueMultipleOk(head) ≜
    ∧ head.processCount ≤ 2
    ∧ componentDocsAllSuccessful(head)
    ∧ LET rec ≜ CHOOSE d ∈ dbDocs : d.id = head.id
      IN  ∧ dbDocs′ = UNION {
                    (dbDocs \ {rec}),
                    {[rec EXCEPT !.processStatus = "Success"]}
                }
          ∧ dbQ′ = Tail(dbQ)
          ∧ pc′ = pc + 1
          ∧ UNCHANGED ⟨docNo⟩
```

Simulate successful processing of the first item in the process queue

```
unqueueOk ≜
    ∧ Len(dbQ) > 0
    ∧ LET head ≜ Head(dbQ)
      IN  IF head.process = "DocGen"
          THEN unqueueSingleOk(head)
          ELSE unqueueMultipleOk(head)
```

Simulate an unsuccessful processing attempt. If any of the compound documents

component documents have failed to generate then immediate fail the generation of the compound document. If component documents are still in the in-process state then requeue the compound document request without incrementing its process count. Lastly if we try to process the compound docuement and fail then requeue the compound document for a retry with its process count incremented.

$unqueueMultipleNotOk(head) \triangleq$
$\land\ head.processCount \leq 2$
$\land$ CASE $componentDocsHaveErrors(head) \rightarrow$
LET $rec \triangleq$ CHOOSE $d \in dbDocs : d.id = head.id$
IN $\land\ dbDocs' =$ UNION $\{$
$dbDocs \setminus \{rec\},$
$\{[rec$ EXCEPT $!.processStatus =$ "Error"$]\}$
$\}$
$\land\ dbQ' = Tail(dbQ)$
$\land\ pc' = pc + 1$
$\land$ UNCHANGED $\langle docNo \rangle$
$\square componentDocsProcessing(head) \rightarrow$
$\land\ dbQ' = Append(Tail(dbQ), head)$
$\land\ pc' = pc + 1$
$\land$ UNCHANGED $\langle docNo, dbDocs \rangle$
$\square$OTHER $\rightarrow$
$\land\ dbQ' = Append(Tail(dbQ), [head$ EXCEPT $!.processCount = head.processCount + 1])$
$\land\ pc' = pc + 1$
$\land$ UNCHANGED $\langle docNo, dbDocs \rangle$

For a single document if the processing fails for some reason then requeue the document incrementing its process count by one.

$unqueueSingleNotOk(head) \triangleq$
$\land\ head.processCount \leq 2$
$\land\ dbQ' = Append(Tail(dbQ), [head$ EXCEPT $!.processCount = head.processCount + 1])$
$\land\ pc' = pc + 1$
$\land$ UNCHANGED $\langle docNo, dbDocs \rangle$

If we have already tried to process a record twice stop trying. Remove the process record from the queue and set the *policy_document* record status to *Error*

$abortProcessing(head) \triangleq$
LET $rec \triangleq$ CHOOSE $d \in dbDocs : d.id = head.id$
IN $\land\ Assert(head.processCount > 2,$ "only called when we run out of process attempts")
$\land\ dbDocs' =$ UNION $\{$
$dbDocs \setminus \{rec\},$
$\{[rec$ EXCEPT $!.processStatus =$ "Error"$]\}$
$\}$
$\land\ dbQ' = Tail(dbQ)$
$\land\ pc' = pc + 1$
$\land$ UNCHANGED $\langle docNo \rangle$

```
Simulate unsuccessful processing of the first item in the process queue
unqueueNotOk ≜
    ∧ Len(dbQ) > 0
    ∧ LET head ≜ Head(dbQ)
      IN  IF head.processCount > 2
          THEN abortProcessing(head)
          ELSE IF head.process = "DocGen"
               THEN unqueueSingleNotOk(head)
               ELSE unqueueMultipleNotOk(head)

The condition for termination, all records in the queue have been processed
term ≜
    ∧ endDocGen
    ∧ Len(dbQ) = 0
    ∧ UNCHANGED ⟨pc, docNo, dbDocs, dbQ⟩

Init ≜
    ∧ pc = 0
    ∧ docNo = 0
    ∧ dbDocs = {}
    ∧ dbQ = ⟨⟩

Next ≜
    ∨ enqueueSingleDoc
    ∨ ∃ ids ∈ SUBSET {d2.id : d2 ∈ {d1 ∈ dbDocs : d1.collationType = "Single"}} : enqueueMultipleDoc(ids)
    ∨ unqueueOk
    ∨ unqueueNotOk
    ∨ term

Spec ≜ Init ∧ □[Next]⟨pc, docNo, dbDocs, dbQ⟩
```

* Modification History
* Last modified *Tue Oct* 12 09:34:22 *PDT* 2021 by *marco*
* Last modified *Wed Oct* 06 14:37:28 *PDT* 2021 by *ASUS*
* *Created Fri Sep* 17 18:40:13 *PDT* 2021 by *ASUS*

Part II
Systems and Applications

We review Socotras systems. We discuss how they are structured in the large, and what structuring might be an improvement. Some documentation on specific system applications is presented.

Chapter 5
The Systems

To learn to make art you spend time working with the materials. That is it. When the urge arises to judge the quality of your work, it will be tempting to say this is good or this is bad. Take that energy that you might put into assessing the aesthetic and put it back into the materials instead. Persist and grow. Avoid aesthetic judgments, and instead produce work.

Abstract Socotra has a very simple system setup. There are two user interfaces and a few backend systems, the most important of which is the API. The biggest problem with understanding the system setup is that the names of the components don't always imply what the component does. The description, below, will match up components with a function and where possible indicate some reasonable naming.

5.1 Structure of the Overall System

Figure 3.1 shows the overall system. The Config Editor is a policy designer. From within Config Editor product experts can define insurance products as a combination of json configuration files, tables for value lookup, and script files for dynamic policy calculations. Once a product is defined and deployed, then the Policy Manager is used by company representative to create policy holders and to issue and modify policies based on a product.
Load Assets is part of the backend. It accepts zipped product, configuration files from Config Editor and it orchestrates the deployment of those product, configuration files using internal APIs that the Socotra API exposes. The Socotra API is then a very standard API application. It stores data in a few external databases. And it relies on a JavaScript executor called Socol to do dynamic calculations, such as pricing a policy or performing and underwriting decision.

5.2 Structure of Socotra API

The Socotra API application is a simple layered application built on the Spring Framework. The layering is somewhat unconventional; however, which makes the structure not immediately evident. Figure 3.2 shows the current API structure with a generic application structure for comparison. The best way to think of the Socotra

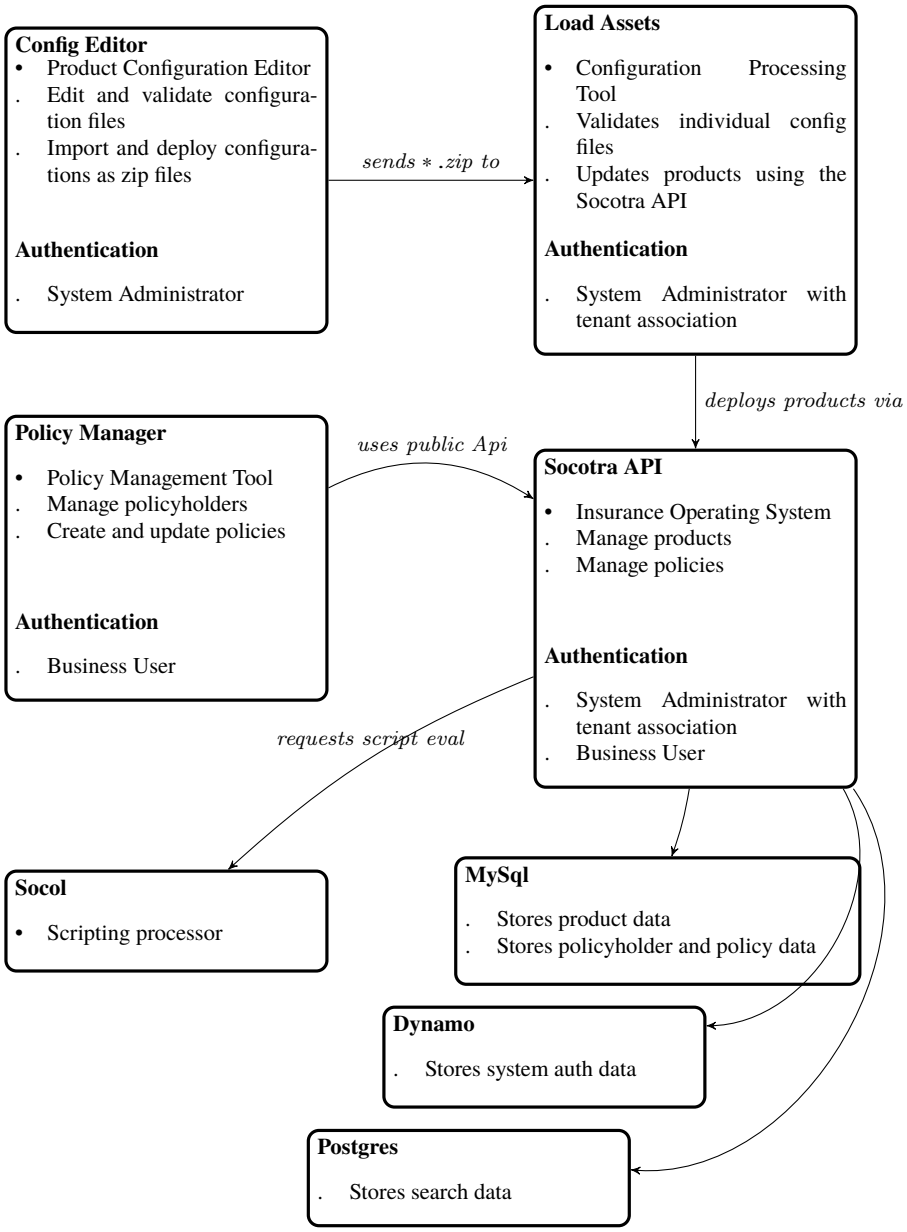

Fig. 5.1 The overall Socotra system

API is as having a very thin controller layer at the front and a very thin layer at the back of request and response objects, that are used everywhere. Then there is a very thick middle layer which contains everything else. The application could certainly

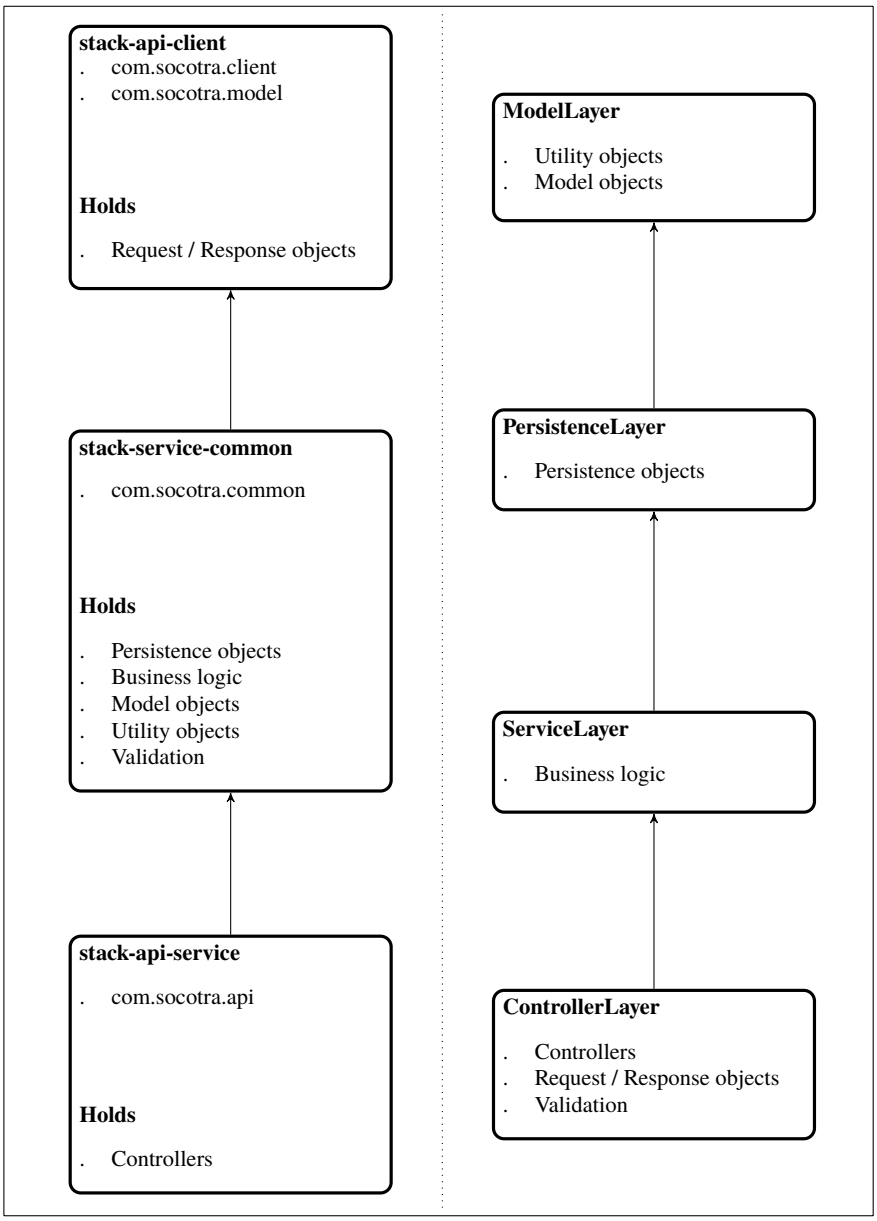

Fig. 5.2 The Socotra API organization on the left. For reference, the industry standard organization for a tiered API on the right.

use more circumspect layering to organize code and developer thinking. It's likely that the application will evolve over time to look more and more like the generic

application structure on the right. In fact, its unlikely that anything more nuanced could be needed.

Chapter 6
The Applications

. . . and now each one went the way upon which he had decided, and they set out into the forest at one point or another, there where they saw it to be thickest.

Abstract Doing work

6.1 Config Editor

6.1.1 Overview

Config Manager allows user to create, modify, and deploy custom, insurance Products.

6.1.2 Local Development (Front End)

Install npm packages: `npm install` (You may have to restart VSCode after this to pick up the installed components)
Generate the default configuration assets as .js files: `npm run create-assets`
Compile the app and serve it via webpack-dev-server: `npm run start-dev`
Chances are you'll need to open the app in a browser that has CORS disabled (b/c the local frontend app will be hitting a deployed backend (as opposed to a backend hosted locally). So this will produce a CORS error). Run this command in a terminal to open a CORS-disabled version of Chrome: `open -n -a /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --args --user-data-dir="/tmp/chrome_dev_test" --disable-web-security`

6.1.3 Local Development (Back End)

- Boot stack-api-service on port 8080 by running it through your IDE, as usual. Stack-api-services, directly, provides login services for config manager
- Boot stack-load-assets on port 5000 by running the command

```
nvm use 8.9.4
rvm use ruby-2.6.3
make run
```

from the stack-load-assets base directory. Stack-load-assets provides all of the asset publishing services for config manager.

- Boot stack-config-manager on port 9001 by running the command

```
make run
```

from the stack-config-manager base directory.
When all three applications are running you should then be able to navigate to http://localhost:9001 in any browser, login, and submit assets all within a local development environment.

6.1.4 Build and deploy

The build of Config Manager is coordinated by the build.sh (python 2) or build3.sh (python 3) in the root of the project. The shell scripts bundle the application and deploy it to repo://source/stack-load-assets/assetload/static/configmanager. Once deployed Config Manager can then be accessed through a running load-assets application, at the path /studio/. So for example, running load-assets locally the access url would be http://localhost:5000/studio/
Build and deploy using Python 3 with:

```
rvm use ruby-2.6.3
nvm use 8.9.4
./build3.sh
```

6.1.5 Directory Layout:

- assets: Contains default configurations artifacts (i.e. default config, default product)
- public: Contains icons and image assets
- sass: Contains sass files for styling. Currently trying to move over to a BEM-style methodology

- src
  - api: Contains code to make requests to backend APIs
  - components: Contains our React components
    - actionpane: code for the actions sidebar
    - actions: Contains the Actions of the Trigger-Action component design
    - modals: Contains modal components
    - pages: Contains pages and layouts
    - popups: Contains popup components (i.e. action menus, dropdown menus). Distinct from modals in that modals are fullscreen and darken the bg. Popups and modals also currently use different underlying libraries for their implementation. They should probably be consolidated at some point.
    - various other components not organized into the above folders
  - context: Contains state management code. We're using React's Context API for state management.
  - models: Contains any complex data models (i.e. the configuration)
  - util: Contains various utility code
- history.tsx: Specifies the history library to be used with react-router
- main.tsx: The main entry point for the app
- routes.tsx: Specifies the app's URL routes. Uses react-router

6.2 Policy Manager

6.2.1 Local Development (Front End)

Install npm packages: `npm install` (You may have to restart VSCode after this to pick up the installed components)
Compile the app: `npm run build`
Serve it locally via webpack-dev-server: `npm start`
Before running locally, ensure the following environment variables are set appropriately:

```
API_URL=<url_of_backend i.e. api.develop.socotra.com>
TENANT_HOSTNAME=<your_tenant_name i.e. johndoe-configeditor.co.develop.socotra.com>
PORT=<this variable is optional. Specify port number to serve the app. i.e. 8080>
```

Note: If you load the app in a browser and log in but experience weird errors, check the console. If there's weird errors (i.e. 'Uncaught Error: Expected to find root ID'), you may need to try loading the app in a CORS-disabled browser. On a Mac, run this command in a terminal to open a CORS-disabled version of Chrome: `open -n -a /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --args --user-data-dir="/tmp/chrome_dev_test" --disable-web-security`
When you run the app, you can log in with the default username and password (Note: these credentials are only available in development. They're disabled in production):

```
username: alice.lee
pw: socotra
```

If you want to log in to the 'Administration' side of the app to manage users and external integrations, you'll need to use your tenant login (not alice.lee).

6.2.2 Local Development (Back End)

If you have a standard backend setup and have setup the docker-dev admin account, via

```
brew install jq jo
cd ~/Code/socotra-stack/docker-dev
make tenant
```

You will be able to start stack-app-static on port 8081 with the commands

```
nvm use 8.9.4
rvm use ruby-2.6.3
make run
```

With stack-app-static running you should then be able to navigate to http://docker-dev-configeditor.co.socotra.com:8081/ locally in a browser (assuming the line '127.0.0.1 docker-dev-configeditor.co.socotra.com' in /etc/hosts) and login with the username/password alice.lee/socotra
To run stack-app-static along with the entire complement of Config Manager, Load Assets, and the API refer to the README.md in the root of the stack-config-manager directory.

6.3 Load Assets

6.3.1 Overview

Load Assets takes configuration files from Configuration Manager, validates them, and then orchestrates though the stack-api-service to configure data stores needed for operation of the software

6.3.2 Development environment setup

Load assets can be run locally on a developer computer against a locally running instance of stack-api-service on port 8080. To setup your environment you can run the command

```
make setup-debian
```

or

```
make setup
```

depending on the type of system you are on. The setup process (1) installs pyenv to manage your virtual environments, (2) installs pipenv and (3) installs all of the dependencies for the stack-load-assets project
After running the setup you should then be able to start your virtual environment by typing:

```
pipenv shell
```

and exit the virtual environment by typing:

```
exit
```

6.3.3 Running the application

The application can be run on port 5000 by entering your virtual environment and then entering the command

```
make run
```

The makefile sets up all of the environment variables needed for the application to run against a locally running stack-api-service

6.3.4 Testing the application

Tests can be run from inside the virtual environment as described below.

- static analysis and type checking is run as part of all the test routines. Both types of checks can also be run stand alone. Static analysis can be run with the command:

```
make static
```

Type checking can be run with the comamnd:

```
make mypy
```

- unit tests can be run with the command:

```
make test
```

- There are currently integration tests for stack-load-assets that run against a locally running stack-api-service. The integration tests can be run with the command:

```
make test_intg
```

6.3.5 Building the application

Though not immediately obvious, Load Assets is built into a composite application where (a) python code orchestrates communication with the socotra api, and (b) where the assetload/static directory contains and serves the build stack-config-studio-static, stack-load-assets-static, and stack-config-manager projects. The assembly of all these elements is managed by the build so just be aware there is more to Load Assets than meets the eye.

6.3.6 Api Overview

There are a documented and non documented API endpoints for the load assets project. A schema for the documented API can be viewed on a development machine at Api Schema A brief discusion of all API endpoints are described below.

6.3.7 Api Detail

6.3.7.1 POST /assets/v1/deploy

Creates or updates a tenant-hostname / tenant combination Request

```
{
    zipFile: Bytes,
    tenantName: String
}
```

- *zipFile* : A zipped directory of socotra configuration files
- *tenantName* : Creates or updates a configuration such that tenant-hostname.hostname = [tenantName].co.socotra.com and tenant.tenant.tenant_name = [tenantName]

Response

```
{
    hostname: String,
    logfile: String,
    success: Boolean
}
```

- *hostname* : the hostname (ex: docker-dev-configeditor.co.socotra.com) at which one can access the uploaded configuration given the uploaded credentials (ex: alice.lee / socotra)

6.3.7.2 /assets/v1/deployTest

Creates or updates a tenant-hostname / tenant combination Request

```
{
    zipFile: Bytes,
    tenantNameSuffix: String
}
```

- *zipFile* : A zipped directory of socotra configuration files
- *tenantName* : Creates or updates a configuration such that tenant-hostname.hostname = [username]-[tenantName].co.socotra.com and tenant.tenant.tenant_name = [username]-[tenantName], where [username] is the admin user contained in the authorization credentials

Response

```
{
    hostname: String,
    logfile: String,
    success: Boolean
}
```

- *hostname* : the hostname (ex: docker-dev-configeditor.co.socotra.com) at which one can access the uploaded configuration given the uploaded credentials (ex: alice.lee / socotra)

6.3.7.3 /assets/v1/export

Request

```
{
    ? includeFailed: Boolean
}
```

Response

```
{
    deployed: Boolean,
    expiresTimestamp: Integer,
    url: String
}
```

6.3.7.4 /assets/v1/login

6.3.7.5 /assets/v1/createAdmin

6.3.7.6 /configuration/deployTest

Request

```
{
    zipFile: Bytes,
    ? tenantNameSuffix: String
}
```

Response

```
{
    hostname: String,
    logfile: String,
    success: Boolean,
    tenantName: String
}
```

6.3.7.7 /configuration/updateExternalServiceIntegrations

6.3.8 End Points For Config Studio

6.3.8.1 POST /configeditor/deploy

This end point is used by Config Studio to submit assests to the Socotra main api service. Requires that the client set an 'authorization' field in the header containing a authorization header
The request should contain a json body of the form

```
{
    assets: Map<String, JSON>
    recreate: Boolean
}
```

ex:

```
{
    assets:{
        config.json: {
            timezone: "America/Los_Angeles",
            currency: "USD",
            improvedRating: true
```

```
            },
        security/roles.json: {
        underwriter1: "Underwriter Level 1",
        underwriter2: "Underwriter Level 2"
        },
    },
    recreate: false
}
```

The response will be of the form

```
{
    hostname: String,
    logfile: String,
    success: Boolean
}
```

6.3.8.2 POST /configeditor/validate

This end point is used by Config Studio to check if asset files are valid. The client does not need to set and 'authorization' header.
The request should contain a json body of the form

```
{
    assets: Map<String, JSON>
}
```

The response will be of the form

```
{
    "config.json":null,
    "payment/payment.json":null,
    "policyholder/policyholder.card.json":null,
    ...
}
```

where null indicates that the corresponding file has passed validation

6.3.8.3 /configeditor/import

6.3.8.4 /configeditor/export

6.3.9 Monitoring End Points

6.3.9.1 GET /health

Just a health check endpoint for monitoring. The response is of the form


```
{
    "duration_millis": 0.011139154434204102,
    "generated_at": "2020-02-07T22:20:58Z",
    "result": "passed",
    "tests": {
        "core_api": {
            "duration_millis": 0.011105060577392578,
            "result": "passed",
            "tested_at": "2020-02-07T22:20:58Z"
        }
    }
}
```


6.3.10 Traffic Analysis

Start mitm proxy using reverse proxy mode, listening on port 8082. With the api service running on port 8080 we will then record any interactions to the sessiondata.dat file.

```
mitmdump -p 8082 --mode reverse:http://localhost:8080/ -w sessiondata.dat
```

6.4 Python Client

6.4.1 Overview

The stack-python-client package provides an interface for communicating with the stack-api-service. There are several utility programs in the bin directory with can be used to setup a client. In addition this package is used by stack-load-assets to help upload zip asset files.

6.4.2 Development environment setup

This library is a dependency of the stack-load-assets application and its development setup procedures are similar to stack-load-assets. To setup your development environment for working on stack-python-client, from the project base directory type:

```
make setup
```

This will install pyenv, pipenv and lastly install the project dependences. After running the setup you should then be able to start your virtual environment by typing:

```
pipenv shell
```

and exit the virtual environment by typing:

```
exit
```

6.4.3 Testing the application

You will be able to run tests in your development environment by activating your virtual environment (see: Development setup) and entering the commands below.

- Static analysis can be run with pylint through the command

```
make static
```

- The unit tests can be run wit the command

```
make test
```

- The integration tests can be run by booting the Java API running on port 8080 and then entering the command

```
make test_intg
```

6.4.4 Using the Python Client library

Do the following to instantiate a Client class that will allow you to communicate with a Socotra API:

```
from socotra import Client
c = Client('http://localhost:8080')
c.authenticate(user_nm, pwd, tenant_loc)
```

Once you have authenticated, you may then call methods on the client object which will in turn make calls into the Socotra API. Below are a few documented methods. See the socotra/**init**.py file for more possibilities.

6.4.4.1 get_tenants

This grabs a json blob of all tenants available in the system currently.

```
tenants = c.get_tenants()
```

6.4.4.2 add_tenant

Adds a tenant
Params:

- name (str): Name of the tenant being added
- type (str): Type of tenant

```
c.add_tenant('Blah', 'tenant.test'))
```

6.4.4.3 add_hostname

Adds a hostname
Params:

- id (str): The ID of the tenant
- hostname (str): The hostname of the tenant

```
c.add_hostname(tenant_data['id'], 'balls')
```

6.4.4.4 ping

Ensures the server is up and ready for requests
Params:

- authorized (bool): Defaults to false

```
c.ping() # will use the endpoint that does not require login
c.ping(authorized=True) # will use the endpoint that requires login
```

6.4.5 Using the socotratenant program

This program allows one to submit a zipfile containing assets to a running version of load-assets. It creates or updates a tenant An example of usage is:

```
export PYTHONPATH=$PYTHONPATH:./:../stack-monitoring
python bin/socotratenant -z ~/Code/Configs/defaultConfig.zip -u docker-dev -p socotra -a ht
```

** -z ** specifies the zip file to upload through load assets ** -u ** specifies the user name which has api access permissions ** -a ** is the endpoint of the Java api ** -l ** is the endpoint of the load assets api ** --test ** flag determines which load asset api is used to upload the zip file. If --test is specified then the assets are uploaded to the endpoint /configuration/deployTest. If --test is not specified then the assets are uploaded to the endpoint /assets/v1/deploy. In both cases the tenant type created in the tenant table is 'tenent.test' ** -n ** is a name related to the hostname / tenant combination that is being created or updated. How the name maps to data in the database differs depending on the --test flag and is as follows:

1. if --test is specified, affected records are those linked to
   - tenant-hostname.hostname = [user name]-[name].co.socotra.com and tenant.tenant_name = [user name]-[name]
   - As an example, the example command, above, creates or updates data associated with tenant-hostname.hostname = docker-dev-docker-dev-configeditor.co.socotra.com tenant.tenant_name = docker-dev-docker-dev-configeditor
2. if --test is not specified, affected records are those linked to
   - tenant-hostname.hostname = [name].co.socotra.com and tenant.tentant_name = [name]
   - As an example, the example command, above, without --test, would create or update data associated with tenant-hostname.hostname = docker-dev-configeditor.co.socotra.com tenant.tenant_name = docker-dev-configeditor

6.4.6 Using the socortraadmin program

This program allows one to update and read information from an installation of the socortra platform. The program takes a combination of positional and named arguments the combination of which defines the data to read or update. One can get information on the commands offered and the format required by the commands by using the --help argument on the command line. Examples of using the --help argument to walk through the positional argument tree is:

```
python bin/socotraadmin --help
python bin/socotraadmin tenant --help
python bin/socotraadmin tenant set_config --help
```

Some example usages of common commands in a local development environment are listed below.

6.4.6.1 Creating a bootstrap account

The command usage is:

```
socotraadmin environment bootstrap_admin [--name NAME] [--username USERNAME]
                            [--password PASSWORD]
                            [--account_type [account.test.tenant.admin | account.internal]]
                            [--api_url] [--jwtsecret]
                            [--admin_username] [--admin_password]
```

This command is usually used to create a special account of type account.internal. These accounts are not associated with a tenant (the tenant is '_'), but once created the account can be used to log onto an instance of Config Manager, create tenants, and users for specific tenants. An example of the usage in a local dev environment is:

```
export PYTHONPATH=$PYTHONPATH:./:../stack-monitoring
python bin/socotraadmin environment bootstrap_admin --name DockerDev --username docker_dev
    --password socotra --account_type account.internal --api_url http://localhost:8080 --jw
```

6.4.6.2 Tenant Maintenance

Adding a user to a tenant

Once a tenant exists a user can be added to the tenant with a command of the form

```
socotraadmin tenant add_user [tenant_name] [user_display_name] [username] [password] [email]
```

where the currently supported account types are

1. account.tenant.employee - the standard user that performs policy operations
2. account.tenant.admin - a user that can deploy configurations but can not perform policy operations
3. account.tenant.read.only.user - a user that can read but not write policy information
4. account.tenant.claims.only.user - a user that can read policy information and write only claim related information

Below is an example of a command that creates a tenant admin account.

```
export PYTHONPATH=$PYTHONPATH:./:../stack-monitoring
python bin/socotraadmin tenant add_user docker-dev-configeditor Admin admin.lee socotra adm
    --api_url http://localhost:8080 --jwtsecret SGAGWfq31D2HRccsq87s33v1
    --admin_username docker-dev --admin_password socotra
```

and a command that creates a readonly user

```
python bin/socotraadmin tenant add_user docker-dev-configeditor ReadOnly readonly.lee socot
    --api_url http://localhost:8080 --jwtsecret SGAGWfq31D2HRccsq87s33v1
    --admin_username docker-dev --admin_password socotra
```

Finding the tenant info associated with a hostname

The command usage is:

```
socotraadmin tenant find_tenant [--api_url] [--jwtsecret]
                                [--admin_username] [--admin_password]
                                hostname
```

This will return the contents of the tenant table in Dynamodb given a hostname. An example of the usage in a local dev environment is:

```
export PYTHONPATH=$PYTHONPATH:./:../stack-monitoring
python bin/socotraadmin tenant find_tenant --api_url http://localhost:8080 --jwtsecret SGAG
    --admin_password socotra docker-dev-configeditor.co.socotra.com
```

Geting the tenant configuration flags

This endpoint returns a list of the tenant feature flags and their values. The command usage is:

```
socotraadmin tenant get_config [--api_url] [--jwtsecret]
                               [--admin_username] [--admin_password]
                               tenant_name
```

An example of the usage in a local dev environment is:

```
export PYTHONPATH=$PYTHONPATH:./:../stack-monitoring
python bin/socotraadmin tenant get_config --api_url http://localhost:8080 --jwtsecret SGAGW
    --admin_password socotra docker-dev-configeditor
```

Setting a tenant configuration flag

This endpoint will set the value of a tenant level configuration flag. The names of available flags can be queried using the get_config option, above. The command usage is:

```
socotraadmin tenant get_config [--api_url] [--jwtsecret]
                               [--admin_username] [--admin_password]
                               [--config_uri] [--config-value]
                               tenant_name
```

An example of the usage in a local dev environment is:

```
export PYTHONPATH=$PYTHONPATH:./:../stack-monitoring
python bin/socotraadmin tenant set_config --api_url http://localhost:8080 --jwtsecret SGAGW
    --admin_password socotra docker-dev-configeditor property.proration.plugin.enabled true
```

6.5 Stack Api Service

6.5.1 Overview

stack-api-service is the highest layer in the Socotra platform. It depends on the statck-service-common project for business and persistence services. It despends on the stack-api-client project for generic and specialized model classes.

6.5.2 Testing

6.5.2.1 Overview

Testing is organized so that a developer can run unit tests and an essential subset of the integration tests locally on a development machine. These unit and essential integration tests are referred to as the essential test subset and satisfy the following criteria:

- Runs must of the essential test subset must run within a time budget of 15 minutes.
- The essential test subset may be run locally by a developer, but the essential subset must run and pass before developers are allowed to merge a pull request into the develop branch.

The full set of project tests, which is a superset of the essential subset, is run continuously by the automated build tools. A failure of the full set indicates a code issue on the develop branch and must be resolved by the team as a their first priority.

6.5.2.2 Running tests

- The unit tests for stack-service-common will run on any machine with a Java installation. The unit tests can be run by the command

```
make test
```

This command insures the local maven repository contains a build of the latest branch specific Java libraries and runs all of the Java unit tests in the socotra-stack project. If one would like to run just the unit tests for the stack-api-service project, that can be done by issuing the command

```
mvn test
```

from the root of the project.

- The essential test subset, as defined above, for stack-service-common will run on any machine with a standard development environment. The essential subset can be run by the command

```
make test_intg
```

If a developer would like to run just the essential test subset for the stack-service common project alone, that can be done by issuing the command

```
mvn verify -P essential-test
```

from the root of the project.

- Currently the full set of project tests will run with the command.

```
mvn verify -P integration-test
```

A developer machine will likely choke on the processing that will ensue after this command so you really don't want to run all the tests. Still, everyone should be familiar with the tests commands that are used in the orgainization.

Appendix A
Extra stuff that I hope might help

Sin bravely.

A.1 Engineering Practices

Over the years of doing engineering, we all come away with different learnings and a view of the task shaped by our experiences. Below are some thoughts I have about engineering which have grow out of my experiences and condensed into topics. Perhaps some of the topics will resonate with you and become part of your experience. The summary is below and details follow:

1. Define the problem. The problem you have isn't the problem you think you have.
2. Build in minimal commits.
3. If you don't understand all observations, then your understanding is wrong.
4. If code is not being used, then it doesn't work.
5. System problems can not be solved with single solutions, or by efforts at a single time.
6. Change one thing at a time.
7. There is no right way. There are only trade offs.
8. Build your personal rule set.
9. If you don't know what to do, then do something anyways.
10. Theory and practice.

A.1.1 Define the Problem.

It should be axiomatic that if one cant define a problem, one cant solve it. It should be equally clear that if one defines a problem incorrectly, then regardless of the quality of the solution one provides, the solution provides no value in the context of the actual need. As obvious as the above statements are, you should not deceive yourself into thinking that the ability to define problems is natural, and even less that a problem, when it reaches your desk, has been correctly defined as stated. One should, in fact, assume just the opposite. In the course of your career, unless you are

finding that 90 percent of the time your problems have to be reexamined and refined, then it's probable you are not thinking carefully enough before you work.
Let's think through a current, discussion topic at Socotra, feature flags, to illustrate defining a problem. I don't happen to know what the future resolution to the feature flag "problem" will be and perhaps that is good. You, the future reader with hindsight and proof of time, will all the better be able to evaluate the usefulness of the thoughts on problem definition that follow.
Lets start with a real life, feature flag problem statement that we can quickly disregard.

Problem The current feature flag mechanism (with flags in database tables) is inelegant.

Perhaps no one has made exactly this feature flag problem statement. But statements like this have been thrown about and, of course as developers, we love to disparage the technologically ugly with variants of this statement. Now, I am not sure what the official engineering definition of inelegant is. If it had an engineering definition, I am not convinced that such a label would be sufficient to establish the existence of a problem. Inelegant is an aesthetic, emotional judgment, and while there is room for aesthetics in engineering, we should be on our guard. Feature flags presumably would fulfill a functional need and not an aesthetic, emotional need. Let's try again.

Problem Lets investigate the Launch Darkly feature flag functionality to see how it could be used in our current systems.

This statement is an example of a solution with an assumed problem. A problem could exist, but certainly the author of the statement above has skipped the formality mentioning it. As an engineer you should sensitize yourself to solutions that eclipse their problems. "We should build new functionality as micro services" or "We should use Amazon Lambda for X, because Lambda is scalable and the future of software development" are other examples. I am sure Launch Darkly is wonderful, as well as micro services and Amazon Lambda, and yet their wonderfulness does not mandate their use. As an engineer, when technical discussion leads directly to technical implementation X or a technology Y, that is a tell that perhaps someone had a vague feeling about a problem and that vague feeling of discontent was easiest for them to causally assign as due to the absence of a technology, for instance "MongoDB". (And as the joke goes, now you have two problems).
Since we achieved insufficient clarity with the above problem definitions let's metaphorically go back to the beginning and talk about feature flags. Feature flags are a soft deployment mechanism, which activate software functionality. Feature flags complement and add additional flexibility to physical deployment mechanisms. In any existing implementation, the flag consists of a name and associated data (Boolean, Enumerated...) stored in persistence somewhere. Is that elegant? We can list some of the common problems that feature flags solve. I took the following problems from the web site of a commercial service that sells a feature flag management product. I believe the list is complete.

Problem 1 A mechanism is needed to rollout software incrementally (for example: 10% of users, 20% of users,. . .) while monitoring metrics for errors or performance problems.

Problem 2 A short lived mechanism is needed to enable subsets of users to see different site functionality, perhaps for the purpose of running experiments.

Problem 3 A mechanism is needed to reconfigure code and it is not practical or desirable to do this via physical deployments.

Briefly reflect on how the above problems apply to Socotra. And with that reflection done, let's define Socotra's problems.

Since feature flags are just a deployment mechanism, let's not immediately bring feature flags into our problem definition exercise, as we might fool ourselves into talking about a solution. Instead lets talk about what needs Socotra has around deployment functionality. Here is one deployment problem I have witnessed at Socotra

Problem A mechanism is needed to rollout software incrementally (for example: organization #1, a few days later organization #2, . . .) while monitoring metrics for errors or performance problems.

For the instance where the above problem came up, Devops could have made incremental physical deployments but since the rollout had more than one moving part it was conceptually simpler to put all the pieces in place and then flip a flag to bring everything live. A database table, _env_feature, supplied the global, boolean feature flag to bring the new functionality live and that mechanism, in practice, turned out to be simple and sufficient. There were no problems. That said, I can envision a more complex Socotra system where functionality might be deployed in independent components in pieces; where one flag to bring all component functionality live at once, would enable conceptually simple deployment; and where a flag in a database table would not be available to all the components. That would pose a deployment problem. We would probably want to be aware of that future problem and address it at the proper time according to our engineering judgment.

In contrast, here is another common deployment problem I am aware of at Socotra

Problem New functionality X has been developed and insurance companies are dying to start using it. Unfortunately insurance companies cant readily use the new functionality because the new functionality requires that new fields or a new file be added to the product configuration. Even though insurance companies can deployed the new product configurations, all their existing policies will be tied to the previous product revision. They have to wait for each existing policy to be renewed, maybe a year, until their policyholders can all enjoy the new functionality. It's inconvenient for insurance companies, their systems, and their policyholders to receive functionality in this one policy by one policy manner.

This is a problem that we anticipate having once product versioning goes live. Today this problem seems more immanent than any other deployment problem Socotra has and as engineers we should have immanent problems clear in our minds. I am going to leave the possible importance of this problem and its solution to the future

but we might observe that practical engineering, perhaps not elegance and certainly not Launch Darkly, will be involved in the solution. We might also observe that, exactly like feature flags, the deployment of configuration zip files is a type of soft deployment. We have an additional, existing soft deployment capabilities which may not have been identified as such when we started the feature flag "problem" discussion.

I will leave the exercise there. There are probably more deployment problems that could be discussed. But that belabors the point to make which is that one should get into the habit of turning over a problem statement in ones mind, formulating it from various angles and reducing it to essentials. In your career as an Engineer it is important that you be successful, and one of the keys skills that will guide you towards doing what should be done, and assist co-workers with business skill sets is the ability to clearly and correctly define what the problem is.

A.1.1.1 Build in minimal commits.

I am often surprised, when I model a system with a computer checked tool, by how difficult it is to get the model correct. Usually the model is highly abstracted and simplified, a toy compared to the real thing. And yet the tools find corner cases that I would never have discovered with human reasoning skills. The reason for the surprise is that as engineers, and more so for non engineers, we have only a passing feeling for the ability of state to multiply. Consider the simple, cancellation model in Chapter 2. How many states are there if the model steps through 8 time intervals? There are almost a million and the model checker takes half and hour to check them all of them.

You are not impressed perhaps. We already know software is complex. This afternoon, I sat through an engineering discussion where an ongoing systems problem was brought up: "Its hard for me to predict the effects of my changes . . . ", because system bad. A few months ago I sat through a related engineering discussion about how difficult it is to estimate delivery dates for new and ongoing projects, because system bad. What we need is? to have better encapsulation, to move the system towards a micro-service architecture, to design with better architecture and patterns . . . or add your own favorite delusion. I am going to suggest an alternate theory for system bad based on my frustrations building simple, abstract mathematical models. I will suggest that you are a very limited human being in the face of what you have created, and you have to learn to work in a way that is appropriate to that reality.

Now, I know you are an experienced engineer, and you have read all the books, and you have come through the toughest, largest projects there are. You already know what you are doing. And still there may be just one more practice that no one has ever told you and in fact its just the opposite of what the people who talk cleaner code, and micro-service architecture, and design patterns propagandize. The practice is: make minimal commits. If you have a bug, or if you have a feature to implement, or if you have a project to finish, find the smallest problem related to your bug, feature or project and define it precisely. Write code to solve just that problem, precisely.

Commit. Repeat until you are done. Anything can be accomplished as a series of minimal commits.
I know you will be tempted to disregard this practice. The minimum. That sounds so negligent. Why you may even desire to refactor. After all that was promoted in those books about clean code, micro-services and patterns and stuff; the guys that wrote those books really know what they are doing, right? Well, I am not telling you to do the minimum. I am not telling you to not make systems better. What I am saying is you are limited. You didn't 100% understand the ramifications of that ambitious bug fix you made today and, by the way, you didn't sleep well last night. That ambitious architecture idea you've been thinking on; two years from now you are going to look back and think it was foolish, because you have grown. I am saying if you are limited, a precise, rigorous practice will help you be successful, and I am afraid other engineers you meet in your career are not going to tell you. Minimal commits. Precisely define the minimal problem, write just that code, and commit. A lot of tasks become better executed, a lot of systems improve, and you will be more successful if you can just repeat that process.

A.1.2 If you don't understand all observations, then your understanding is wrong.

One of the traits of a good theory is that when you apply it to a phenomena you are investigating, then it explains all of the observations you have made, not 90% of your observations, not 95% of you observations, all of your observations. One of the traits of a bad theory is that it explains enough to satisfy you. The classic example of a bad theory is Ptolemy's geocentric model of the universe. It explained almost all heavenly observations and served as the basis for precise astronomical calculations. It was satisfactory enough. One just ignored the last few percent of observations that didn't make any sense. Now, stop for a second and imagine yourself believing the geocentric model of the universe. Are you mostly correct in your understanding of the universe or do you know essentially nothing about the universe?
If you have lived long enough, you can dip into your memories of past, very reasonable, theories which did not quite pan out. When I was growing up, I had an Aunt who had a subscription to National Geographic, a very reputable magazine at the time. Sometime in the 70's I remember National Geographic running a whole magazine detailing the case for an impending famine in the US. Back in the 70's it was also assured knowledge that the world would run out of oil sometime in the early 1990s. In retrospect, National Geographic knew essentially nothing about famine. Assured knowledge knew essentially nothing about the availability of energy resources on the planet. And yet the theories accounted for most to the facts. How can that be? What you have to realize is that mismatches between observations and theories have to drop your confidence in your theory a lot. If you have 10 observations and your theory explains 9 of those observations. As much as you'd like to, you should not have 90% confidence in your theory. You should have more like 50% confidence. If

you can explain 8 observations, you should have more like 25% confidence in your theory. Put another way you need more explanatory power than you think before you know more than nothing.
Now lets talk about theories in relation to computer systems at Socotra. I am going to recount a story from a few months ago. I was in Socotra's daily scrum and an engineer explained that he was working on a bug. He explained that he couldn't quite reproduce the error or detail how the error was coming about. But he was pretty confident that the ultimate cause had to be X and he had committed a solution Y. That is bad practice, and it is all too common among software engineers. Software engineering culture accepts and allows people to make the jump from "my theory explains an observation" to "my theory is correct". This jump is not acceptable or allowed in any other engineering discipline. Unless you can reproduce, unless you can explain, unless you can account for all the observations, then you know nothing. You need to dig deeper, and you have no business committing solution Y, except to gain missing data.
Above we talked about when a bug is properly understood and hopefully corrected. On a related note, and getting back to the title of this post, imagine you are writing code, building a system. Further, imagine that in one of the system executions you notice something curious. Possibly an error, but you cant reproduce it. And you are very busy; it's probably something one off. I am going to suggest that your understanding of your code is like a theory and you have an observation that contradicts the theory. With that one contradiction, you now need more explanatory power than you think to assume something one off happened. In fact, despite your best feelings of assurance, you have to assume, with assurance, that the code is wrong. And do work to generate an explanation.
What I have written above might seem a bit strong. If it seems a bit strong then for your next few bugs fixes, start by listing all you theories. Maybe you will start off with 10 possible theories and you might well add new theories as you work through the bug. Note that you will generally have to toss away a lot of plausible theories before you finally find a root cause of your bug. Note also that the solution that you finally end with, will not generally be in the initial set of theories you started with.
What I have written above might, on the other hand, seem obvious. If that is that case, then the next time you read the news just take note of topics of current concern and the related theories, sufficient for you to feel assured in your opinions. (Covid and Global Warming are on the front page of CNN as I write this.) Then come back to this paragraph in 15 years.

A.1.3 If code is not being used, then it doesn't work.

When building systems its common to have unused code: sub-systems that are finished, waiting to go live or code that is not executed on a regular basis. Any code that one expects to work must prove it works on a regular basis. If the code does not prove it works, your best assumption is that it does not.

A.1.4 System problems can not be solved with single solutions, or by efforts at a single time.

In a recent engineering meeting development quality was evaluated in terms of the number of customer bug reports per month. One tentative suggestion from the graphs was that there was a large number of latent bugs in existing code waiting to be exposed by new customers doing new things. This prompted the question of how can we remove bugs from the existing code and the response was "we have to write more tests". How likely is that to be a solution for uncovering latent bugs in code?
We can answer questions like this by making a quick engineering estimates. How many state items are there in the configuration plus database fields that can influence processing? Lets say the state items are bounded on the low side by 10^2. Lets further assume that all states are binary and that processing is never influenced by the confluence of more than three data items. The system then has a lower bound of 10^6 state combinations. If not properly handling some of those 10^6 states would lead to a bug, then we can say that to sufficiently remove just data bugs, a lower limit on the number of tests necessary would be 10^6. Further we would still have many other classes of bugs to investigate with perhaps equally large state spaces. No team will ever write those tens of millions of tests, and if they could then they would spend months of time just trying to run their test suite, once. The answer "we have to write more tests" does not work in theory. Slightly differently, theory tells you a given solution is wrong.
I am going to give you another answer to the question of how to remove bugs in existing code. But the answer is general to all improvements that one seeks to implement in complex systems. You have to accumulate a set of effective attack methods to use towards your goal and you have to apply those methods systematically over an extended period of time. The extended period of time varies by the problem domain but for software systems a typical value of the extended period of time is on the order of one year.
When I started my first job as a software engineer, I worked at a financial institution in Boston with a faulty reporting system. It was a big problem because customers expected timely (right now) financial reports and there were ten to fifty failures that had to be resolved every morning. Every morning I would try to identify every problem and manually produce the reports by any means necessary. Then I would go downstairs to my desk and fix as many problems as were identified. I would also search all the other reports for any specific classes of problems that could be extrapolated from the days failures and I would fix those too. For some issues I would ask "what would I have to do to prevent an issue like this from ever happening again?" and then I would do that. For issues that I couldn't find a root cause for, I had a lab notebook where I would record every issue and every observation that I had made about the issue. Everything was tracked. Where I needed more information, I would add logging and monitoring. After a year of persistent attack from many angles, one day I came into work and there were no report failures. Over the years, with many failing systems I have personally seen variations of the above story replay many

times. There are no easy one shot solutions. One applies many simple strategies, consistently over time, and in about a year one will end up where one wants to be.

A.1.5 Change one thing at a time.

Most systems, even simple systems are complex enough to exceed the reasoning ability of the best engineers, and we can see the manifestation of this truth regularly at Socotra. We have a moderately complex system and yet its not unknown to see one or more engineers struggle to fix system problems and be uncertain about the ultimate fix. Will the fix be correct in all circumstances? It's also not unknown for specifications of new functionality to be incomplete or in need of revision after discussion of system details. Demonstrably, no one can reason about our system with confidence, and I would suspect that the most confident are in line to fall the hardest. One golden rule for navigating system complexity (along with minimal commits) is to always change one thing at a time. If someone changes module X and you would like to add functionality in module X for your feature, Y. Wait for X to go live in production and then add your feature Y. If problems occur in Module X or a related module before your changes land, then someone else is closer to isolating what changes have created problems. It's always much harder to analyze a problem when multiple changes have happened at the same time especially if the changes interact in unintuitive ways. One change at a time is also why many organizations deploy code continuously. If each commit is well targeted, which it should be, then only one piece of functionality has changed. If after a deployment, monitors show abnormalities, then code can be rolled back and it will be very apparent what single change caused what problem. Surprisingly what seems to be very dangerous, deploying as soon as commits land, turns out to be safer because only one thing is changing at a time.

A.1.6 There is no right way. There are only trade offs.

Try never to say "this is the right way to do it". That statement may be emotionally satisfying but it is not engineering. Engineering is about trade offs. Just like deciding on family priorities, or deciding what business opportunities to pursue. One defines the problem (see above) or, mathematically, one defines a set of inequalities which bound the solution space. One then navigates among needs, values, precise discussion with stakeholders, and hopefully good judgment to pick a point in that solution space. Note I said "a point" not "the optimal point". There is no optimal point as the optimal point depends on your loss function, and the loss function depends on ones priorities. For some efficiency might be highly weighted in the loss function, for others time to market, or something else entirely, might be highly weighted. Optimality depends on what trade offs are important to you. Being an engineer means you can come up

with many possible solutions to any problem and you can explain the trade off for each one.

A.1.7 Build you personal rule set.

When you work on complex systems try to find rules that can guide you in your interaction with that system. All interactions with complex systems that have survived through the ages do precisely this. In human society the rules that guide interaction are called culture. In governance the rules that guide interactions will be conventions like parliamentary procedures, rule of law, and freedom of speech. In an economic system one rule might be voluntary exchange. The rules exist not because they are perfect but because they have been found to work well most of the time. And because human judgment is not sound enough to distinguish the exceptional cases. Similarly when you code or modify systems think about the rules that should guide you in your changes. If you create a bug perhaps you should have a rule for each bug you fix you must write a unit test. That's a common rule that a lot of engineers have. If you have to make a change to a system you might decide that as a rule you will identify the minimal fix, as discussed. And code just that. You might decide that you will never commit a change to source control without actually doing a test on a live system and checking output, even when there is no chance you could have made a mistake.
Building a personal rule set is a task of identifying where you fail and where others succeed. In both cases you identify a better way that brings you closer to the kind of engineer you want to be and then you stick to that way. You rules don't have to be perfect. They just need to lead you to do the right thing most of the time.

A.1.8 If you don't know what to do, then do something anyways.

In the course of you career you will have to solve many different kinds of problems. Some of those problems will be very stressful. They may seem to be impossible to solve or understand. You may even run out of avenues of attack where you don't know what to try next. When this happens, be systematic, don't give up, and do something anyways. Persisting and staying observant often leads to the discovery of a hint that will lead you to your solution.

A.1.9 Theory and Practice.

In the course of my career I have met some very practical engineers who were very serious about doing but who lacked theory to discipline their thoughts. Those engineers were handicapped without theory and they were often unaware of their

handicap. I have also met very theoretical engineers who lacked the applied interest in creating practical technical artifacts. Some of those engineers ended up disappointed and disillusioned, when their lofty visions where not realized in practice.

Don't be a practitioner and don't be a theoretician. Your job as an engineer is to continuously circle back between both theory and practice. Theory tells you which avenues will be possible and which will be impossible. It is your first error correction filter. Practice is where you meet the real world and you learn the limitations of your technology and your abstractions through feedback. This is your second error correction filter. Once you have met the real world then you recycle your learnings back into your theory to make it better. And you continue the cycle over and over.

Glossary

Characteristics A set of attributes, key value pairs, that describe selected options for policy coverage.

Endorsement A change to an existing policy that adds, subtracts, or updates an element of coverage.

Premium Report Refers to the action of reporting a premium to an invoicing system, thereby generating an invoice. The term has nothing to do with generating reports.

Tenant A logical subdivision of data in a physical deployment. Policies within a physical deployment are always associated with one tenant and within a physical deployment there are one or more tenants.