Universidade de São Paulo
Institute of Mathematics and Statistics
Department of Computer Science

A Formalization of a Startup Finance Transaction Model using Alloy

Rodrigo Ehrlich Stevaux

Advisor: Prof. Ana Cristina Vieira de Melo

Dissertation presented to the Institute of Mathematics and Statistics at the University of São Paulo, as part of the requirements for the degree of Master of Science in Computer Science

São Paulo

2023

Abstract

This thesis proposes a formal model for a subset of the startup finance transaction space. The initial version of the provided domain is the result of an industry coalition effort to make the data model standard.

The data definition explains how this domain can be modeled syntactically. We refined this first model with semantics on transactions by using the Alloy formal modeling language and analyzer, aiming for a more expressive and correct model by capturing domain invariants. As a result, our design is machine-checkable for important safety integrity criteria and stated as a formal domain specification.

This research contributes to the field of formal methods by demonstrating how to progress from a semi-formal specification to a formal one, evaluating the results, and providing a case study of a real-world domain to other researchers and practitioners in the field of formal methods.

I dedicate this thesis to my late, loved mother, and to my father.

I thank my advisor, Prof. Ana Cristina Vieira de Melo, not only for her academic guidance and support throughout this work, but for the friendship and care she has shown me since during this endeavor.

I am grateful to my institution for providing me with the opportunity to pursue this work, and from all I have learned from my colleagues and professors.

I thank all the people who helped me arrive at this point, including my family, friends, and colleagues.

Contents

List of Figures

List of Tables

List of Listings

Glossary

acquisition An acquisition is when one company purchases another company. Depending on how many shares are acquired, the acquiring company may gain control of the acquired company.. 9, 16

angel investor An angel investor is an individual who provides capital for a business start-up, usually in exchange for convertible debt or ownership equity. These investors typically support startups in the early stages of growth. 9

asset An asset is something that can eventually generate cashflows. Because not all future cashflows are known with certainty, the value of an asset must be discounted to reflect the risk that those cashflows do not meet expectations.. 9, 35

asset class An asset class is a group of securities that have similar characteristics. Stocks, bonds, and real estate are all asset classes.. 9, 16, 20, 32, 38, 39

capitalization table A capitalization table is a table that lists all the securities issued by a company. The capitalization table lists the number of shares issued, the type of security, the price per share, and the date of issuance. The capitalization table is used to calculate the ownership of each shareholder.. 9, 15–17, 19, 20, 22, 23, 28, 35, 46, 49, 53, 108

cliff period The cliff period is the period of time before which no stock options are vested. After the cliff period, stock options are typically vested monthly.. 9, 43

common stock Stock that holds no special rights beyond a share in profits. Common stock is the most common type of stock.. 9, 16

convertible debt In startup financing, it is typical to encounter convertible debt. Convertible debt is a loan that can be converted into equity at a later date. The conversion is typically triggered by a future financing round. The conversion price is typically set at a discount to the price of the future financing round. Since debt is safer, convertible debt has lower investment risk. Conversion is typically at the option of the holder.. 9, 44

debt Debt is a loan that must be repaid. Companies might raise funds via equity issuances or debt issuances. Debt is issued as security in terms of the amount that was loaned, the interest rate, and the maturity date. Debt is safer than equity, and must be repaid before equity holders can receive any cashflows.. 9, 21, 35, 44

dilution Dilution is the reduction in ownership stake that occurs when new shares are issued. In the special case that in the financing round investors purchase new shares proportionally to their then current stake, no dilution occurs. Founders and employees can also be diluted by the issuance of new shares.. 9

equity Equity represents ownership interest in a company. It can come in the form of stocks or stock options and may be granted to founders, employees, or investors. Equity holders have a claim to the profits of the company.. 9, 21, 22

exercise Stock options are exercised and become stocks. The strike price is the price at which the stock options can be converted to equity. They can only be exercised after they have been vested.. 9, 20, 21, 31, 35, 39

financing round Each stage of financing is also called a financing round. Each financing round is typically led by a lead investor, who sets the terms of the financing round. The terms of the financing round include the valuation of the company, the price per share, and the type of security issued.. 9, 20

initial public offering During an initial public offering, a company sells shares of its stock to the public for the first time. This is also known as *going public* and is a way for companies to raise capital for new investments or to pay off existing debt.. 9, 16

issuance An issuance is the creation of a new security plural. 9

issuer Companies are issuers of securities.. 9, 28, 30, 31

liquidity event A liquidity event is an event that allows a company's stakeholders to cash out or make a profit from their investment. This could be a sale of the company (also known as an exit), an IPO, or a large dividend distribution. 9

preferred stock Preferred stock is stock that holds special rights. As an example of a special right, preferred stock might have a guaranteed dividend payment but less voting rights.. 9, 16, 22, 23, 30, 39, 44

security A security is a financial asset that can be bought and sold. Stocks, options and debt notes are all securities. Every security has an Issuer. A loan from a bank is not a security, because the bank can not generally sell the loan to another bank.. 5, 9, 16, 28, 30–32, 35–37, 39, 41, 44–46, 53

share Shares represent a fraction of ownershp in a company. The number of shares a stock holder owns is the starting point for calculating their ownership stake in the company.. 9

staged financing Staged financing is a financing strategy in which a company raises funds in stages. The first stage is typically called the seed round, with subsequent stages receiving a latin alphabet letter (such as Series A, Series B, etc.). Staged financing allows investors to reduce their risk by investing in stages, and allows the company to raise funds as it grows.. 9, 21

stakeholder A stakeholder is any person, legal or natural, with an economic interest in a company, including all debt, option and stock holders.. 9, 16

startup company A startup company is a new company that is searching for a business model as it grows. Startup companies are typically funded in stages and by specialized venture capital investors such as individual (angel) investors and funds. Startup companies usually aim for high growth and high returns, by choosing projects with higher risk.. 9, 15, 20, 21

stock Stocks are securities that represent ownership in a company. Stock are typically issued as shares (e.g. 100 shares of Apple stock). Shares are fractions of a company's total ownership. plural. 9

stock class A stock class is a group of stocks that have similar characteristics. Common stock and preferred stock are both stock classes.. 9, 16, 31, 34, 39, 44

stock option Stock options are securities that give the right for their holder to purchase stock at a predefined price (the strike price) at a predefined date (the maturity date). The value of a stock option is the different between the market price for the stock and the strike price. Stock options are typically issued to employees as part of their compensation package.. 9

strike price The strike price is the price at which a stock option can be exercised. The strike price is typically set at the market price of the stock at the time of issuance, but may be futher discounted to incentivize employees.. 9, 16

sweat equity Startup founders raise capital by selling shares of their stock to investors. For example, a investor might take 20 of 100 shares, leaving founder with 80 shares that were not issued against cash. This complement of the shares that held by investors is called sweat equity.. 9, 22

transaction A transaction refers to the issuance, change, transfer and cancellation of securities. A transaction is typically initiated by a stakeholder, and must be approved by the company. Most transactions involve a cost, with money changing hands in the opposite direction of the securities.. 9, 16, 21, 28, 29, 31–36, 39, 46, 108

venture capital Venture capital is a form of private equity financing that is provided by venture capital firms to startups and early-stage companies that have been deemed to have high growth potential or which have demonstrated high growth. 9, 15, 16, 20

vesting period A vesting period is a period of time during which an employee must remain employed in order to receive the full value of their stock options. Vesting periods are typically 4 years, with a 1 year cliff period. Only vested stock options can be execised.. 9, 111

vesting schedule A schedule that sets out how many stock options are vested at different dates, usually monthly, but also in according to specific milestones that can accelerate the vesting of options. 9, 20

Part I

Background

Chapter 1

Introduction

Capitalization tables are essential documents required for conducting **venture capital** investments in **startup companies**. A capitalization table depicts the ownership structure of a company, and this structure is subject to change following investments, transfers, and acquisitions.

Errors in **capitalization tables** can prove costly and may lead to potential legal disputes. Traditionally, **capitalization tables** have been maintained in spreadsheets, a method which is prone to error and difficult to audit. Furthermore, spreadsheets do not adhere to a standard format, requiring all parties involved in a transaction to agree on a uniform format before exchanging data.

The inherent issues associated with maintaining **capitalization tables** in spreadsheets have prompted the emergence of several companies that offer **capitalization table** management as a service. These companies provide a web-based interface for managing **capitalization tables**, which aims to streamline the process and reduce errors. However, these services are not without limitations. The underlying data models they use are proprietary and not publicly accessible, and the rules for updating the **capitalization tables** are often not explicitly defined.

1.1 The economic motive to adopt a specification

To give objective measures of cost and risk to the task of validating **capitalization tables**, we can consider the following:

1. 38,644 **venture capital** deals were closed in 2021 (according to accounting firm KPMG [1]). Each deal requires tens of hours of lawyers and accountants.
2. The **venture capital** market invested USD 671 billion from investors in 2021, and exits (including **initial public offerings** and **acquisitions**) amounted to USD 1.378 billion (also according to the same KPMG report).
3. As of June 2023, Apple, Google, NVIDIA, Amazon, and Microsoft alone were worth USD 9 trillion. All companies started as **venture capital** investments. Their combined market up is comparable only to the gross domestic product of the United States of America (USD 23 trillion) and that of China (USD 17.7 trillion).

1.2 Complexities in managing capitalization tables

Capitalization tables are complex to manage for several reasons:

1. Different **asset classes**. **Common stock** entitles profit sharing and voting rights, generally proportional to the number of shares. **preferred stock** generally carries less voting rights but more economic rights, such as having priority over **common stock** in case of, and can in some cases be exercised (converted) to **common stock**. *Stock options* are **securities** that can be converted to shares of a given **stock class** upon the payment of a **strike price**, which, if lower than the market price, will result in profits for the option holder.
2. Distinction between actual and diluted shares.
3. Difference in the price each investor paid for shares, and the respective investment cost must be recorded properly to calculate returns in case the company is acquired or goes public.
4. There is no information on the **transactions** that led to the current capitalization table, which *must* be considered to validate a capitalization table.

The last point is the most critical one. The only way to attest that a **capitalization table** accurately reflects the correct stakes of each **stakeholder** is to validate the **transactions** that led to the current **capitalization table**.

This is a non-trivial task, and is extremely error-prone if done manually, which alone motivates the need for a specification for **capitalization tables**. As we shall see in depth, a candidate for such a specification is in development by the Open Cap Table Coalition [2], based on JSON Schema, which we introduce next.

1.3 Open-source efforts

Recently, the Open Cap Table Coalition [2], formed by industry members, has been working on a standard for **capitalization tables**.

The standard is based on a data model that is publicly available, and that can be used to build software systems that manage **capitalization tables**. However, the data model is not accompanied by a formal specification of the rules for updating the **capitalization tables**. This is an unfortunate consequence of the specific semantics of the chosen technology, JSON Schema, which is syntax-oriented and does not allow for the specification of complex rules.

1.4 Structure

In this thesis, we present a formal specification of the rules for updating **capitalization tables**. We do so by first modeling the data model in Alloy, and then by specifying the rules for updating the **capitalization tables** in Alloy. We then use the Alloy Analyzer to check that the rules are consistent with the data model, and that the rules are free of errors.

Our goal is to show that formal methods, in the form of Alloy, can be feasibly used to specify the rules for updating **capitalization tables**. We hope that this work will lead to a clearer understanding of the rules for updating capitalization tables, and to the development of software systems that implement these rules.

1.5 Related works

Although Simon P. Jones' paper provides a comprehensive formal semantics for the evaluation of derivative contracts, a similar formal semantics for capitalization tables remains absent from existing literature[3]. Financial instruments compose into a big expression that can evaluated to the raw dollars.

Relevant contributions to Alloy modeling will be discussed in the chapter dedicated to the model itself.

The work by INRIA in Catala is also relevant[4]. Catala is a domain specific language for writing legal text, with applications to British immigration law and the french tax code. Writing tax rules must be as difficult as investment rules, so it warrants a full study object for a language. Catala can analyse contracts to show contradictory clauses. It is certainly useful in doing due-dillugence using Catala.

There is research in smart contract verification, and the field is accordingly moving fast. It is not entirely visible as research, since many projects exist as open-source software projects. Nevertheless, we can point to the KEVM, which represents a formal semantics for the Ethereum Virtual Machine (EVM) and is grounded in the K framework, a tool for modeling programming languages [5], and other projects such as Lolisa [6], for the Solidity language, and Scilla, an intermediate-level contract language based in Coq[7].

In contrast, our focus is not only on correctness but conceptual; we aim to establish a robust *domain model* or *conceptual model* that can serve as a foundation for deriving properties to shape software specifications. Our choice of Alloy stems from its user-friendly nature, its ability to provide a high-level representation of the domain, and its utility in model checking. While we initially contemplated the use of dependent types, we ultimately opted for Alloy due to its simplicity and adequacy for our objectives. Moreover, Alloy augments our work with valuable visualization tools, enhancing our modeling and analysis capabilities.

The organization of this thesis is outlined as follows:

- In Chapter 2, we introduce the original data model for capitalization tables as proposed by the Open Cap Table Coalition, along with an examination of the JSON Schema technology that serves to define this data model.
- Chapters 3 and 4 are devoted to elucidating our Alloy model, which represents the data model for capitalization tables. Additionally, this chapter outlines the rules governing the updates to these tables.
- In Chapter 5, we explore the interplay between our Alloy model and traditional prose contracts, specifically focusing on their alignment in the context of code normativity.
- Finally, Chapter 6 provides our concluding thoughts and sheds light on potential avenues for future research.

Chapter 2

Capitalization tables and the need for specifications

2.1 Overview

A **capitalization table** is a table that shows the ownership stakes in a company. We give an example in table 2.1. The table shows the ownership stakes of the founders, angel investors, funds, and employees. The table also shows the number of shares issued, the type of security, the price per share, and the date of issuance. The capitalization table is used to calculate the ownership of each shareholder.

Acme Corp.
Capitalization table — December, 20XX

Asset class	Stakeholder	Shares Actual/diluted	Cost	% Ownership Actual/diluted
Common stock	Founders	700/700	0	78%/70%
Preferred stock	Angel investors	50/50	0	6%/5%
Preferred stock	Funds	150/150	300	16%/15%
Options	Employees	0/100	0	0%/10%
Total		900/1000	350	100%/100%

Table 2.1: A capitalization table

Stakeholders have interests in different **asset classes**, resulting in some shares and % ownership at a given cost. Prices can change as investor value the company differently at different points in time.

Options are granted to employees, who can **exercise** the options and exchange them for actual shares. As an incentive for the employee to stay with the company, virtually all options have a **vesting schedule**, which is a period of time that the employee must remain with the company in order to **exercise** the options. The most typical case is to vest shares after 1 year, and then vest the remaining shares monthly over the next 3 years. Vesting will be the subject of a later section.

2.1.1 Use in startup financing

Capitalization tables are used specially in startup financing. **Startup companies** are financed in stages, which each stage requiring the achievement of certain objectives and milestones but also providing a larger of capital.

Stages in startup financing

Stages are defined in terms of the phase of the company's development and the amount of capital required to achieve the next stage. The stages are[8]:

- Seed/Startup financing: relatively small amount of capital, for companies mostly aiming to prove a business concept. Beyond **Venture capital** firms, individual investors ("angel" investors) are also common sources of seed funding.
- Early-stage financing: moderate amounts of capital, for advancing pilots or productions and searching for commercial viability evidence.
- Mid-stage financing: this stage revolves on supplying a company with working capital so that it can continue to grow.
- Later-stage financing: this stage is for companies that are already on the way to profitability, and serves as a bridge to an initial public offering (IPO) or a sale to a larger company.

Financing rounds are numbered not by their stage but by letters, typically: Seed, Series A, Series B, Series C and so on. Investment sizes range from USD 1 million in seed rounds to USD 100 million or more in later rounds.

In this manner, the computer scientist reader may note that **a capitalization table is the state of a system, and financing operations are transitions of that state**. Figure 2.1 illustrates this idea.

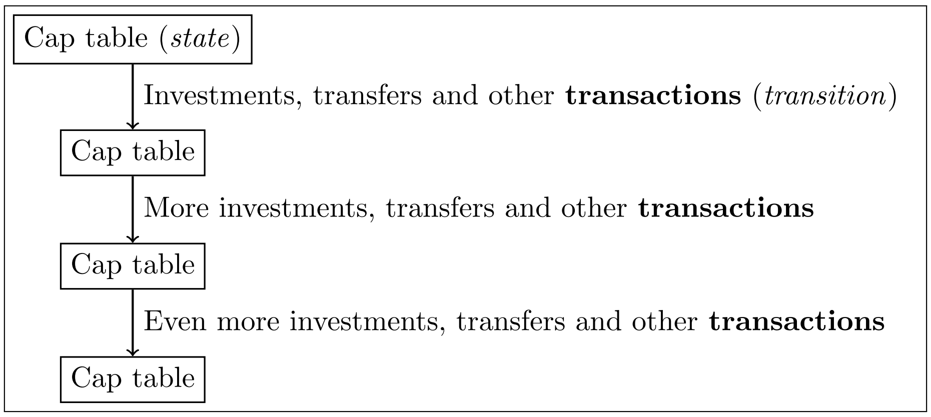
Cap table (*state*)
Investments, transfers and other **transactions** (*transition*)
Cap table
More investments, transfers and other **transactions**
Cap table
Even more investments, transfers and other **transactions**
Cap table

Figure 2.1: Capitalization tables as states of a system

Financing rounds and investment contracts

Each **staged financing** round is defined in contracts that define business rules that are potentially complex to validate. It is common for the investors of each round to have different rights and obligations:

- Seed rounds are frequently financed via a loan to the company that can later on be converted to **equity**. Precise triggers and conversion rates are defined in a contract. **Debt** confers more rights to the investor than **equity**, such as the right to receive interest payments and the right to be paid back before equity holders.
- **Startup companies** typically reserve a portion of shares for employees, which are typically granted in the form of stock options. Stock options are granted according to vesting schedules, and become shares when the employee **exercises** the option.

Conversions, transfers, vesting and other events can only happen within certain conditions, and those conditions must always be validated, on the risk of misrepresenting the company's ownership structure.

2.1.2 Capitalization table over two investment rounds and a final sale transaction

Since we hinted at seeing the **capitalization table** as states in a transition system, we can now see an example of a **capitalization table** over two investment rounds:

Company formation

At company formation, the capitalization only reflects the founders' ownership of the company, by giving them a number of shares with no matching cost. **Equity** that has no matching investment is typically called **Sweat equity**.

Asset class	Stakeholder	Shares Actual/diluted	Cost	% Ownership Actual/diluted
Common stock	Founders	100/100	$ 0	100%/100%
Total		100/100	$ 0	100%/100%

Table 2.2: A capitalization table at company formation

After a seed round, a group of investors joined the **capitalization table**, purchasing 100 shares for $1 each. The founders' ownership is diluted to 50% of the company. The new shares are issued as **preferred stock**.

Asset class	Stakeholder	Shares Actual/diluted	Cost	% Ownership Actual/diluted
Common stock	Founders	100/100	$ 0	50%/50%
Preferred stock	Seed investors	100/100	$ 100	50%/50%
Total		200/200	$ 100	100%/100%

Table 2.3: A capitalization table after a seed round

The company creates an option pool for its employees, reserving 100 shares for that purpose. The founders' ownership is diluted to 33% of the company. The new shares are issued as options.

Asset class	Stakeholder	Shares Actual/diluted	Cost	% Ownership Actual/diluted
Common stock	Founders	100/100	$0	50%/33%
Preferred stock	Seed investors	100/100	$100	50%/33%
Options	Employees	0/100	$0	0%/33%
Total		0/100	$100	100%/100%

Table 2.4: A capitalization table after an option pool is created

And then the company receives a Series A investment, where a new group of investors purchases 100 shares for $2 each. The founders' ownership is diluted to 25% of the company. The new shares are issued as **preferred stock**.

Asset class	Stakeholder	Shares Actual/diluted	Cost	% Ownership Actual/diluted
Common stock	Founders	100/100	$0	33%/25%
Preferred stock	Seed investors	100/100	$100	33%/25%
Preferred stock	Series A investors	100/100	$200	33%/25%
Options	Employees	0/100	$0	0%/25%
Total		300/400	$300	100%/100%

Table 2.5: A capitalization table after a Series A investment

Finally, after the company is sold, the **capitalization table** is updated to reflect the exit. The company is sold for $1000, and the proceeds are distributed to the shareholders. The founders' ownership is diluted to 25% of the company. The new shares are issued as **preferred stock**. The share numbers are recalculated to simplify the math[1]. The stocks from the employees' stock options are exercised and become actual shares.

[1]It is common practice in finance to rebase figures after many transactions to keep the numbers simple enough to facilitate calculations. In our case, all investors where divided into founders and non-founders, and the corresponding participations of 25/75 used as a basis for the stock after the acquisition

Asset class	Stakeholder	Shares Actual/diluted	Cost	% Ownership Actual/diluted
Common stock	Founders	25/25	$0	25%/25%
Common stock	Float	75/75	$100	75%/75%
Total		100/100	$100	100%/100%

Table 2.6: A capitalization table after an exit

Float refers to the publicly held shares of a company. In this case, the float is 100 shares, and the founders' ownership is 25% of the company.

2.2 JSON Schema

JSON Schema is a specification for JSON (JavaScript Object Notation) data, and provides an approach for defining the structure and constraints of JSON data [9]. JSON is a lightweight data interchange format, and is ubiquitously employed in web applications and APIs due to its simplicity and readability, and JSON Schema, allows developers to meticulously define the structure and constraints of JSON data. A comprehensive description of JSON Schema is provided in RFC8259 [10].

The versatility of JSON Schema is evident in its extensive range of validation rules and constraints. These include, but are not limited to, data types, required fields, minimum and maximum values, and regular expressions. Furthermore, JSON Schema supports custom validation rules and extensions, thereby offering developers the flexibility to define their own rules and constraints. This feature significantly enhances the adaptability of JSON Schema to cater to diverse and specific requirements.

JSON Schema is a powerful tool that leverages the simplicity and readability of JSON, making it human-friendly and easy to comprehend. It employs hypermedia principles, allowing schemas to reference other schemas through a Universal Resource Identifier (URI). This feature enhances the modularity and reusability of schemas, thereby promoting efficient schema design and management.

In the context of designing messages for HTTP application programming interfaces (APIs), JSON Schema is exceptionally adequate. It provides robust validation capabilities, ensuring the integrity and consistency of data communicated through APIs. Not only can it validate the presence of all necessary arguments, but it also offers basic format validation. This means it can check if the data conforms to the specified types, patterns, or other constraints, thereby ensuring that the data received or sent via APIs adheres to the expected structure and format. This comprehensive validation capability significantly enhances the reliability and robustness of HTTP APIs, making JSON Schema an indispensable tool in modern web development.

2.2.1 Validating the presence of specific keys in a document

JSON Schemas can be used to check the presence of a set of valid keys of a document, as well as the set of valid values for that key. The values for each key are given a JSON Schema themselves. This means that JSON Schemas cam compose with other JSON Schemas.

The structure and acceptable formats for the fields can be defined in terms of types, enumerations, and constants. Table 2.7 shows the available validations in JSON Schema.

Validation Keyword	**Specification**
type	of the data type (e.g., string, number, object, array, boolean, null)
enum	a predefined list of acceptable values
const	a constant value that the data must match

Table 2.7: Available Validations in JSON Schema (Types and enums)

2.2.2 Composing schemas to form more complex schemas

Schemas can be composed using composition keywords. Any schema might refer to other schemas by a URI, and the referred schema is expected to be found at that URI. The composition keywords are shown in table 2.8.

Operator	Specification
allOf	all the schemas must be valid for the document to be valid.
anyOf	at least one of the schemas must be valid for the document to be valid.
oneOf	exactly one of the schemas must be valid for the document to be valid.
not	the schema must not be valid for the document to be valid.

Table 2.8: Composition Operators in JSON Schema

2.2.3 Validating numeric, string and array values

Numeric validation is restricted to the value of a single field, which can be constrained to be a multiple of a constant, or to be within a range. Table 2.9 shows the available numeric validations in JSON Schema.

Validation Keyword	Specification
multipleOf	that a numeric instance is divisible by this keyword's value
maximum	the maximum numeric value
exclusiveMaximum	a numeric instance to be strictly less than this keyword's value
minimum	the minimum numeric value
exclusiveMinimum	a numeric instance to be strictly greater than this keyword's value

Table 2.9: Available Validations in JSON Schema (Numbers)

Strings can be validated by their length, or by a regular expression. Table 2.10 shows the available string validations in JSON Schema.

Validation Keyword	**Specification**
maxLength	the maximum length of a string
minLength	the minimum length of a string
pattern	a regular expression that a string must match

Table 2.10: Available Validations in JSON Schema (Strings)

Arrays can be validated by specifying a schema for its elements, by the number of elements, and by whether the elements must be unique. Table 2.11 shows the available array validations in JSON Schema.

Validation Keyword	**Specification**
items	constraints for array items
maxItems	the maximum number of items in an array
minItems	the minimum number of items in an array
uniqueItems	that all items in an array must be unique

Table 2.11: Available Validations in JSON Schema (Collections)

Object validation is the most basic of all, and is used to specify what keys are allowed and required in any object. Table 2.12 shows the available object validations in JSON Schema.

Validation Keyword	**Specification**
properties	constraints for object properties
required	required properties in an object
additionalProperties	whether additional properties are allowed

Table 2.12: Available Validations in JSON Schema (Objects)

2.2.4 Achievements and limitations of JSON Schema

JSON Schema is supported by many tools [11]: validators, schema generators and code generators. There are validators for all major languages, and the schema generators can generate schemas from code, data, and models. Code generators can implement basic Web-based user interfaces and generate data based on schemas. It is also closely integrated with the OpenAPI specification, which is the de facto standard for describing HTTP APIs.

JSON Schema can validate data syntactically, which is adequate for a file or data exchange format. However, JSON Schema provides little in terms of expressiveness in sets, relations, and logic. It is not possible to express constraints such as "the sum of the shares of all stakeholders must be equal to the total number of shares issued by the **issuer**". In JSON Schema, we cannot rule out cyclical sequences of **security** trades because the language lacks a closure operator. JSON Schema can not reason about any relational properties of instances.

2.3 The Open Cap Table format

Our analysis is based on the following commit. The reader can run the command to fetch the specific files from GitHub.

```
git clone https://github.com/Open-Cap-Table-Coalition/Open-Cap-Format-OCF
git checkout 20f3ede62d1f5bdbef16ae1edfa98c34fbda2610
```

2.3.1 File format

The Open Cap Table format defines a package, or set, of JSON files, for storing data on transactions and business entities. Adopting the format means being able to export the **capitalization table** data according to the format.

The manifest contains metadata about the other files, which contain data. The data files either contain immutable entities which participate in **transactions** or the transactions themselves, which are the events that change the state of the entities. Table 2.13 shows the files in the format.

File	**Contents**
Manifest	Metadata
Issuers	Static
Stakeholders	Static
Stock classes	Static
Stock legend templates	Static
Stock option plans	Static
Vesting terms	Static
Transactions	Dynamic

Table 2.13: Files in the Open Cap Table format

2.3.2 The existence of an implied conceptual model within the data model

Although the OCF specifies a set of *files* by defining its contents as JSON documents with associated JSON Schema, it is built on top of an implied conceptual model that underlies the data, which is just a representation.

This *conceptual* model implied in the *data* model is of great value for us. We will therefore present its organizing principles, describe its structure, and discuss the key components of the model while identifying the design patterns used to represent them.

2.3.3 Organizing principles of the underlying model

The format is given as a set of JSON Schemas. The schemas are used to validate the data files. Each schema is defined in a file ending with `.schema.json`.

The schemas are organized according to two principles, which are reflected in the directory structure of the repository:

- Technical building blocks: *enums*, *types* and *objects*
- Conceptual blocks: *entities*, **transactions**, *conversion mechanisms, rights and triggers*, and *vestings*

Technically, types (in OCF terminology) define structures (expected keys and associated validation) that are reused in primitive objects (in the sense JSON objects and documents) and in Enum (enumerations of constant values).

2.3.4 Opening example of the complete lifetime of a security

As we analyze the format, we will see how it can store a case such as the following:

1. Issuance Event: 1,000 shares of preferred stock are issued to Bob (generates a new **security** ID)
2. Acceptance Event: Bob accepts the shares (refers to the **security** ID generated in transaction 1)
3. Conversion Event: Bob converts 500 shares to common stock (refers to the **security** ID generated in transaction 1)

   Since this is a partial transaction, beyond issuing a **security** for 500 shares of common stock, we must issue new **preferred stock** for the remaining 500 shares as well
4. (a) Issuance Event: 500 shares of common stock are issued to Bob (a new **security** ID is generated)

   (b) Issuance Event: 500 shares of **preferred stock** issued to Bob (a new **security** ID is generated)
5. Transfer Event: Bob transfers 500 shares of **preferred stock** to Alice

   Again, this is a partial transaction, so we must issue new **preferred stock** for the remaining 500 shares as well
6. (a) Issuance Event: 500 shares of **preferred stock** are issued to Frank (a new **security** ID is generated)

   (b) Issuance Event: 500 shares of **preferred stock** issued to Bob (a new **security** ID is generated)
7. Repurchase Event: **Issuer** repurchases 500 shares of common stock from Bob (This is a complete transaction, so no new **security** ID is generated)

Conceptually, the model defines entities which are referenced to in **transactions**. Entities include **issuers**, stakeholders, **stock classes**, stock option plans, vesting terms and stock legend templates. **Transactions** include **security** trades, conversions, grants, **exercises**, cancellations, terminations, and vesting events. Conversion mechanisms, rights, and triggers are used to define the conversion of **securities**. Vestings are used to define vesting terms.

By considering all **transactions**, we can derive the state of the entities at any point in time.

2.3.5 A note on the current folder structure of the OCF

The two principles are mixed in the original OCF distribution, regarding the folder structure. Nevertheless, the principles are useful enough for our purposes. We will use them to describe the OCF schemas. Figure 2.2 shows the directory structure of the OCF distribution.

```
schema/
├── schema/enums/
├── schema/files/
├── schema/objects/
│   └── schema/transactions/
├── schema/primitives/
│   ├── schema/files/
│   ├── schema/objects/
│   └── schema/types/
└── schema/types/
    ├── schema/conversion_mechanisms/
    ├── schema/conversion_rights/
    ├── schema/conversion_triggers/
    └── schema/vesting/
```

Figure 2.2: Directory structure of the OCF distribution.

2.3.6 Transactions

Transactions are the actions performed on **securities**. **Transactions** are organized in primitive **transactions** and specific transactions. Primitive **transactions** represent the different *kinds* of **transactions** in the financial domain. Figure 2.3 shows the directory structure of the primitive **transactions**.

```
schema/
└─ primitives/
   └─ objects/
      └─ transactions/
         ├─ acceptance/Acceptance.schema.json
         ├─ cancellation/Cancellation.schema.json
         ├─ conversion/Conversion.schema.json
         ├─ exercise/Exercise.schema.json
         ├─ issuance/Issuance.schema.json
         ├─ reissuance/Reissuance.schema.json
         ├─ release/Release.schema.json
         ├─ repurchase/Repurchase.schema.json
         ├─ retraction/Retraction.schema.json
         ├─ SecurityTransaction.schema.json
         ├─ StockClassTransaction.schema.json
         ├─ StockPlanTransaction.schema.json
         ├─ Transaction.schema.json
         └─ transfer/Transfer.schema.json
```

Figure 2.3: Primitive transactions in the OCF

The primitive **transactions**, with additional fields, are specialized to each of four different types of securities: stocks, convertible notes, warrants, and plan **securities**. We group them by the **asset class** they specialize in. Figure 2.4 shows the directory structure of the specialized **transactions**.

```
schema/
└─ objects/
   └─ transactions/
      ├─ cancellation/PlanSecurityCancellation.schema.json
      ├─ repurchase/PlanSecurityRepurchase.schema.json
      ├─ retraction/PlanSecurityRetraction.schema.json
      ├─ transfer/PlanSecurityTransfer.schema.json
      ├─ release/PlanSecurityRelease.schema.json
      ├─ reissuance/PlanSecurityReissuance.schema.json
      ├─ acceptance/PlanSecurityAcceptance.schema.json
      ├─ issuance/PlanSecurityIssuance.schema.json
      └─ exercise/PlanSecurityExercise.schema.json
```

Figure 2.4: Transactions relevant to stock options and vesting

Figure 2.5 shows the directory structure of the specialized **transactions** for stocks.

```
schema/
└─ objects/
   └─ transactions/
      ├─ acceptance/StockAcceptance.schema.json
      ├─ cancellation/StockCancellation.schema.json
      ├─ conversion/StockConversion.schema.json
      ├─ issuance/StockIssuance.schema.json
      ├─ reissuance/StockReissuance.schema.json
      ├─ repurchase/StockRepurchase.schema.json
      ├─ retraction/StockRetraction.schema.json
      └─ transfer/StockTransfer.schema.json
```

Figure 2.5: Transactions relevant to stock

Figure 2.6 shows the directory structure of the specialized **transactions** for warrants.

```
schema/
└─objects/
  └─transactions/
    ├─acceptance/WarrantAcceptance.schema.json
    ├─cancellation/WarrantCancellation.schema.json
    ├─exercise/WarrantExercise.schema.json
    ├─issuance/WarrantIssuance.schema.json
    ├─retraction/WarrantRetraction.schema.json
    └─transfer/WarrantTransfer.schema.json
```

Figure 2.6: Transactions relevant to warrants

Figure 2.7 shows the directory structure of the specialized **transactions** for convertibles.

```
schema/
└─objects/
  └─transactions/
    ├─acceptance/ConvertibleAcceptance.schema.json
    ├─cancellation/ConvertibleCancellation.schema.json
    ├─conversion/ConvertibleConversion.schema.json
    ├─issuance/ConvertibleIssuance.schema.json
    ├─retraction/ConvertibleRetraction.schema.json
    └─transfer/ConvertibleTransfer.schema.json
```

Figure 2.7: Transactions relevant to convertibles

Additional **transactions** adjust parameters of **stock classes** and stock option plans. Figure 2.8 shows the directory structure of the adjustment **transactions**.

```
schema/
└─objects/
  └─transactions/
    └─adjustment/
      ├─StockClassConversionRatioAdjustment.schema.json
      ├─StockClassAuthorizedSharesAdjustment.schema.json
      └─StockPlanPoolAdjustment.schema.json
```

Figure 2.8: Adjustment transactions in the OCF

2.3.7 Entities and other objects

Beyond **transactions** being entities, the OCF defines entities for each one of the files in the manifest (cf. page 29).

2.4 Key components and patterns in the OCF

The OCF has three key logical components: tracing of transactions, rules for vesting, and rules for convertible securities, as shown in table 2.14.

Transaction & tracing	**Transactions** that are linked by **security** identifiers (i.e., the issuance and cancellation of a **security** refer to a common **security** identifier)
Vesting	Composable rules for both schedule-based and event-based vesting
Convertible securities	Composable rules for converting securities, typically applied in the case of **debt** that can be converted to stock shares

Table 2.14: Key components in the OCF

2.4.1 Transaction tracing system

Since a **capitalization table** is a snapshot after a set of **transactions** have been accumulated, one component of the OCF is the support of traceable **transactions**. The general principle is that **transactions** are objects which are stored in the **transactions** file (cf. page 29), which refer to **securities**. A **security** is a financial **asset** that can be bought and sold. Stocks, options, and **debt** notes are all **securities**.

Securities do not have a specific schema in the OCF—they are given implicitly as correlated identifiers.

The key ideas behind the transaction tracing systems are:

- Securities have a *initial* and a *terminal* transaction.
- Issuances (including re-issuances) and **exercises** are initial **transactions**.
- Cancellations, retractions, repurchases are terminal **transactions**.

- Transfers are both initial and terminal **transactions**, as they extinguish the initial **security** and create a new **security** (the result security).
- Partial cancellations are possible by extinguishing the original **security** and generating a new **security** with the remaining balance.
- Partial transfers are possible by extinguishing the original **security** and generating a new **security** with the remaining balance

To compute the state of a **security**, all transactions that refer to it must be traced. The state of a **security** is the sum of all issuances minus the sum of all cancellations, retractions, repurchases, and transfers. This chaining is what enables auditability and traceability in the system.

Pictorially, the scheme is as follows:

An issuance

Figure 2.9 shows an issuance as an input-output system (in a special case with no inputs).

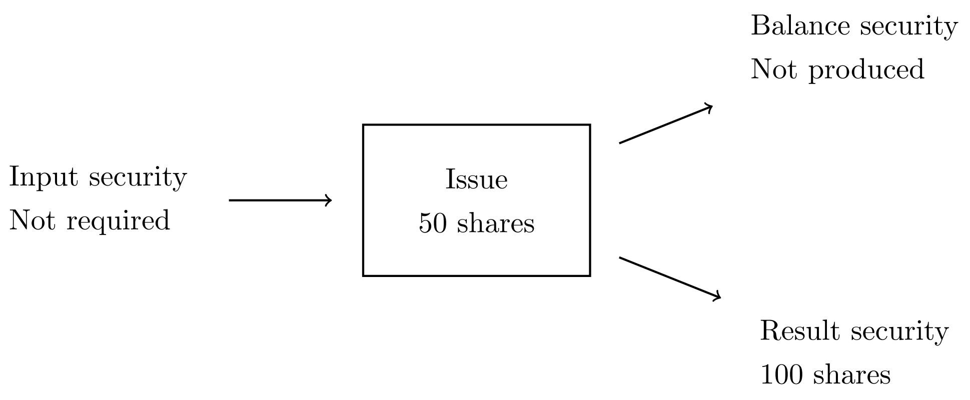

Figure 2.9: An issuance
A issuance requires no **security** as input, and always produces a resulting security

A partial cancellation

Figure 2.10 shows a partial cancellation as an input-output system. The existing **security**, of which some number of shares will be cancelled, is the input. The output is the **security** with the balancing shares.

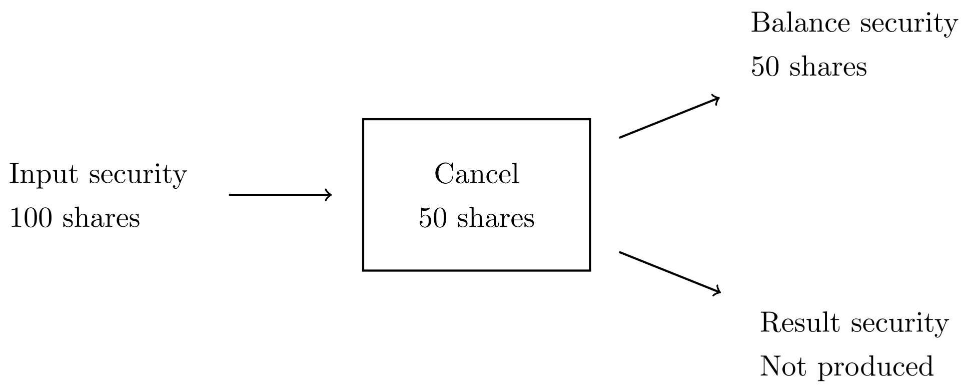

Figure 2.10: A partial cancellation
Cancelling 50 of 100 shares implies a partial cancellation, and a balance **security** is thus generated

A partial transfer

Figure 2.11 shows a partial transfer as an input-output system. The existing security, of which some number of shares will be transferred, is the input. The outputs are the new **security** and the balancing shares.

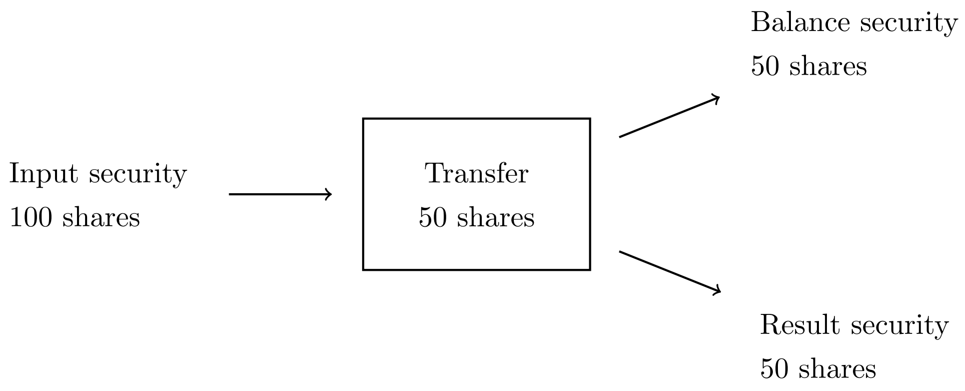

Figure 2.11: A partial transfer
Transferring 50 of 100 shares implies a partial transfer, and a balance **security** is thus generated

Example —the transfer schema

The full listing for the transaction schemas can be found in the appendix, but by giving the transfer schema, we can depict the general pattern of the transaction schemas.

Listing 1 shows the primitive transfer schema, which must be extended by the specific transfer schemas for each **asset class**. It states that every transaction must present a `security_id` field.

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "$id": "$/primitives/objects/transactions/SecurityTransaction.schema.json",
  "title": "Primitive - Security Transaction",
  "description": "Abstract transaction object to be extended by all transaction objects
↪  that deal with individual securities",
  "type": "object",
  "properties": {
    "security_id": {
      "type": "string"
    }
  },
  "required": ["security_id"],
}
```

Listing 1: Primitive transaction schema

In listing 2, we see the transfer schema, which extends the primitive transaction schema.

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "$id": "$/primitives/objects/transactions/transfer/Transfer.schema.json",
  "type": "object",
  "properties": {
      "consideration_text": {
      "type": "string"
      },
      "balance_security_id": {
      "type": "string"
      },
      "resulting_security_ids": {
      "title": "Security Transfer - Resulting Security ID Array",
      "type": "array",
      "items": {
    "type": "string"
      },
      "minItems": 1,
      "uniqueItems": true
      }
  },
  "required": ["resulting_security_ids"]
}
```

Listing 2: Transfer schema in the OCF

In the listing above, we can clearly see the pattern of an input **security** being consumed and a new **security** being created. The `balance_security_id` field is used to indicate that a partial transfer is being made, and the `resulting_security_ids` field is used to indicate that multiple **securities** are being created.

Transactions that affect different **asset classes** (e.g. common stock, **preferred stock**, options, warrants, etc.) have different behavior: options have **transactions** specific to vesting and **exercises** and objects to represent the vesting schedule, warrants have **transactions** to represent the vesting schedule, and convertible **securities** have also objects representing the conversion mechanisms and parameters.

The full cross-referenced table of **asset classes** and their transaction types is given below. For clarity, we omit concepts in the OCF that are **transactions** with no affected security, such as **stock class** splits and adjustments and stock option pool increases. Table 2.15

Transaction Type	**Convertible**	**Options**	**Stock**	**Warrant**
Acceptance	Yes	Yes	Yes	Yes
Cancellation	Yes	Yes	Yes	Yes
Conversion	Yes		Yes	
Exercise		Yes		Yes
Issuance	Yes	Yes	Yes	Yes
Reissuance			Yes	
Release		Yes		
Repurchase			Yes	
Retraction	Yes	Yes	Yes	Yes
Transfer	Yes	Yes	Yes	Yes

Table 2.15: Transactions and their supported security types

shows the transactions and their supported security types.

2.4.2 Vesting system

As we discussed above in 2, stock options follow a vesting system whereby options are released to optionees over time.

Figure 2.12 shows the directory structure of the vesting schemas.

```
schema/
├─ enums/
│  ├─ VestingDayOfMonth.schema.json
│  └─ VestingTriggerType.schema.json
├─ files/
│  └─ VestingTermsFile.schema.json
├─ objects/
│  └─ transactions/vesting/
│     │  ├─ VestingAcceleration.schema.json
│     │  ├─ VestingEvent.schema.json
│     │  └─ VestingStart.schema.json
│     └─ VestingTerms.schema.json
└─ primitives/
   ├─ types/vesting/
   │     ├─ VestingConditionTrigger.schema.json
   │     └─ VestingPeriod.schema.json
   └─ vesting/
      ├─ VestingCondition.schema.json
      ├─ VestingConditionPortion.schema.json
      ├─ VestingEventTrigger.schema.json
      ├─ VestingPeriodInDays.schema.json
      ├─ VestingPeriodInMonths.schema.json
      ├─ VestingScheduleAbsoluteTrigger.schema.json
      ├─ VestingScheduleRelativeTrigger.schema.json
      └─ VestingStartTrigger.schema.json
```

Figure 2.12: Vesting schemas in the OCF

The key object is the Vesting Term object. This object is used to represent the vesting terms of a **security** (vesting term is a synonym for a schedule). The responsibility of the vesting term is factored into more schemas. The vesting term itself defines an allocation type and points to an array of vesting conditions. Listing 3 shows the vesting term schema.

```
{
    "$schema": "http://json-schema.org/draft-07/schema",
    "$id": "$
    /objects/VestingTerms.schema.json",
    "type": "object",
    "properties": {
        "allocation_type": {
        "$ref": "$/enums/AllocationType.schema.json"
        },
        "vesting_conditions": {
            "type": "array",
            "items": {
                "$ref": "$/types/vesting/VestingCondition.schema.json"
            },
        },
    },

}
```

Listing 3: Abridged vesting term schema

The Vesting Condition, in turn, defines a number of shares and points to an array of vesting triggers. Listing 4 shows the vesting condition schema.

```
{
    "$schema": "http://json-schema.org/draft-07/schema",
    "$id": "$/types/vesting/VestingCondition.schema.json",
    "type": "object",
    "properties": {
        "id": "string",
        "portion": {
            "$ref": "$/types/vesting/VestingConditionPortion.schema.json"
        },
        "quantity": {
            "$ref": "$/types/Numeric.schema.json"
        },
        "trigger": {
            "oneOf": [
                { "$ref": "/types/vesting/VestingStartTrigger.schema.json" },
                { "$ref": "/types/vesting/VestingScheduleAbsoluteTrigger.schema.json" },
                { "$ref": "/types/vesting/VestingScheduleRelativeTrigger.schema.json" },
                { "$ref": "/types/vesting/VestingEventTrigger.schema.json" }
            ]
            "type": "array"
        },
        "oneOf": ["portion", "quantity"]
    }
}
```

Listing 4: Abridged vesting condition schema

Each vesting trigger, in turn, is one of four kinds:

- **Vesting Start** triggers after the **Cliff period** has passed
- **Vesting Schedule Absolute** triggers on a specific date
- **Vesting Schedule Relative** triggers after a specific period of time has passed relative to another date
- **Vesting Event** triggers after a specific event has occurred

In the most typical case, that of one year **Cliff period** and monthly vesting over the next three years, the corresponding vesting term would contain 1 + 36 vesting conditions, the first of which is triggered by a vesting start event, and the remaining 36 of which are triggered by a vesting schedule relative event (regarding the vesting start), with a period of one month.

By composing the triggers described above, one can define a variety of vesting terms. This composition of triggers is nothing short of a proto-domain specific language. That is why the Open Cap Table Coalition uses the term “vesting graph” for the resulting composition of triggers and conditions (see [12]).

The vesting term system can accommodate the most basic vesting scenarios —we will improve upon, so it can support a richer set of vesting terms, by expanding the idea of a domain-specific language.

2.4.3 Convertible securities

In this section, we show how conversion mechanisms model transformation of **securities** from one type to another, which usually involves either the conversion of **preferred stock** into common or the conversion of **debt** into glsequity. Figure 2.13 shows the directory structure of the convertible **securities** system.

The building blocks of conversions are mechanisms, rights, and triggers. There are mechanisms for converting to a given percentage of a company's capitalization, as a fixed number of shares, at a given ratio, and in the case of **convertible debt** notes, the mechanism is given in terms of a valuation cap and a discount, and also gives concepts related to interest payments.

A conversion right complements the conversion mechanism by giving the **stock class** of the resulting security. Conversion triggers are more interesting, and specify whether the conversion is triggered electively, at a given date or event, or whether it is automatic.

2.5 Discussion

2.5.1 Advantages and achievements of the OCF

We see OCF as incredibly valuable given how much knowledge it gathers in a single specification. The choice of design patterns results in a data model that can fulfill the requirements of auditability (in special regarding transaction tracing) and flexibility (via the domain-specific languages implicit in the vesting and conversion systems).

After reviewing the Open Cap Table Format, we can agree that:

```
schema/
├── objects/transactions/conversion/
│         ├── ConvertibleConversion.schema.json
│         └── StockConversion.schema.json
├── primitives/
│   └── objects/
│       └── transactions/
│           └── conversion/Conversion.schema.json
├── primitives/
│   └── types/
│           ├── conversion_mechanisms/ConversionMechanism.schema.json
│           ├── conversion_rights/ConversionRight.schema.json
│           └── conversion_triggers/ConversionTrigger.schema.json
└── types/
    ├── conversion_mechanisms/
    │   ├── CustomConversionMechanism.schema.json
    │   ├── FixedAmountConversionMechanism.schema.json
    │   ├── NoteConversionMechanism.schema.json
    │   ├── PercentCapitalizationConversionMechanism.schema.json
    │   ├── RatioConversionMechanism.schema.json
    │   └── SAFEConversionMechanism.schema.json
    ├── conversion_rights/
    │   ├── ConvertibleConversionRight.schema.json
    │   ├── StockClassConversionRight.schema.json
    │   └── WarrantConversionRight.schema.json
    └── conversion_triggers/
        ├── AutomaticConversionOnConditionTrigger.schema.json
        ├── AutomaticConversionOnDateTrigger.schema.json
        ├── ElectiveConversionAtWillTrigger.schema.json
        ├── ElectiveConversionInDateRangeTrigger.schema.json
        ├── ElectiveConversionOnConditionTrigger.schema.json
        └── UnspecifiedConversionTrigger.schema.json
```

Figure 2.13: The convertible **securities** system

1. The OCF consolidates a wealth of expert knowledge in the **capitalization tables** domain.
2. The data format was explicitly designed to be auditable: ultimately there is a set of immutable entities that when replayed together with the **transactions** give the current state of the **capitalization table**.
3. The vesting system and the conversion system, in particular, are based on a complex tangle of business rules which are modeled in the OCF (such is the case that the corresponding part of the model looks like a domain-specific language).
4. Many useful auxiliary types are provided to model dates, addresses, monetary values, which are useful in itself.

2.5.2 Disadvantages and limitations of the OCF

The choice of JSON Schema can arguably be defended because it is widespread, easy to use and can validate data, by its purpose.

However, we believe that the use of JSON Schema is a limitation to the OCF, because of the characteristics of JSON Schema's semantics.

JSON Schema fails to express invariants of the system. The JSON Schema itself cannot distinguish wrong transaction traces from correct ones. In fact, the samples given in the repository are given only syntactically correct. The documents in the samples are well-formed, so to say. But they are not semantically correct. The examples are trivial.

In order for the OCF to reach its full potential as a standard, correct validators must be available. In this regard, we must note that the choice of JSON Schema carries a number of downsides.

For any language to be able to validate the OCF data, it must be capable of expressing some facts that are not at all possible to validate with JSON Schema:

1. To validate that no more options have been exercised than those granted, a language must be able to reason about the fields of associated entities, such a stock plan and its plan securities, and perform sums. JSON Schema has no way to apply this validation. Accounting equations cannot be verified in any practical way with JSON Schema.
2. There is nothing that JSON Schema can do to make sure that there are no cycles as we trace **securities**. Transitive relations must be considered when tracing **securities**.

Critically, JSON Schema is incapable of validating the OCF properly.

Secondly, a format also must be able to express a wider range of business logic: although the support rules for vesting are expressive enough for a broad range of cases, it can only express a conjunctive condition (all triggers must be satisfied) and it can only refer to points or intervals going forward and unbounded in the future. But this precludes the combination of a vesting condition that triggers when a milestone is met within a deadline because we would have to model an "until" operator.

We can improve the model by making the vesting term more expressive, and more capable of modeling the conditions that can be found in legal documents in the wild.

We can improve on how easy it is to validate and improve the expressiveness of the original OCF by adopting a stronger modeling framework: Alloy and the Alloy Analyzer. This should keep the business knowledge that is available in the OCF, while making it more rigorous.

Part II

A new model for capitalization tables

Chapter 3

Key Concepts and Methodology

3.1 The new model *versus* the Open Cap Table Format

If a model that operates inside the same domain as the Open Cap Table Coalition (OCF) and integrates numerous insights from it were to be developed in a more comprehensive language, it may enhance the process of specifying and implementing tools for validating, managing, and precisely reporting capitalization tables.

The Open Cap Table Format (OCF) is limited to elementary data validation and lacks the expressive power to codify community-accepted business rules. While it can conduct basic field-value validation, it is unable to enforce structural or arithmetic rules, thereby constraining its applicability. Such limitations are inherent to its chosen modeling language, JSON Schema.

On the other hand, our improved model makes use of the ability of the Alloy language to formalize limitations as well as update rules specific to a **capitalization table**. The expanded variety of capabilities offered by Alloy, in comparison to JSON Schema, can be credited to its higher degree of expressiveness. Table 3.1 outlines a comprehensive comparison between the two models, highlighting the contributions made by our research.

New Model	**Original Model (OCF)**
Explicit representation of current state	Necessitates transaction replay for state
Structural Constraints	Field-Value Validation
Accounting Constraints	Rudimentary Integer Range Constraints
Advanced Vesting System with AND/OR/NOT	Basic Vesting System with OR

Table 3.1: Comparison between our model and the original model

Although the new model draws significant inspiration from the OCF principles and ground structure, it has enhanced the original model by incorporating semantics related to transactions and vesting. These enhancements will be further elaborated upon in the subsequent sections.

3.2 Key Concepts in the Model

Like its Open Cap Table Coalition predecessor, our Alloy model centers on securities and transactions. However, transactions in our model are state-transformative; they invalidate previous securities and instantiate new or balanced ones. This ensures an always-current, consistent state, encapsulated by structural and accounting constraints.

Our state-management approach diverges significantly from the original model. Transactions result in the deprecation of input securities and the generation of output and balance securities, thus enabling local state validation. This removes the need for historical transaction replay and streamlines implementation, as the validity of each transaction becomes locally deterministic.

We focus on stock and plan securities, omitting less common types like warrants and convertible debts. Our model, however, remains extensible for future inclusion of these types, owing to the local nature of the constraints. This will become clear as we examine the specific signatures in Section 4.1.

3.2.1 Treatment of Securities

In the Alloy-based model, securities are managed according to the following computational principles:

1. Each security is anchored to an *initial transaction*, constituting its instantiation within the system. Optionally, a *terminal transaction* may demarcate its termination.
2. Securities are flagged as either *current* or *expired.*
3. Transition to an expired state occurs solely upon association with a *terminal transaction*, thus invalidating the security for subsequent transactions.
4. A security is instantiated either as a new or derived from a *parent security*, thereby inheriting attributes such as type and constraints.
5. Share quantities must remain positive, enforcing accounting integrity and regulatory compliance.

3.2.2 Treatment of transactions

In the formal Alloy-based model, transactions adhere to the following computational rules:

1. Transactions are composed of input, output, and balance securities. Input securities undergo transformation, output securities are instantiated, and balance securities capture residual shares not transferred or voided.
2. Transactions may function as either the *initial transaction* or *terminal transaction* for securities. The former activates a security, while the latter marks its termination.
3. Transactions may possess a *parent transaction*, which is accountable for generating the input securities. Attributes like transaction type and constraints can be inherited from this parent transaction.

These rules induce a directed graph, with parent-child relationships between securities and transactions forming its edges. By representing these edges as a partial ordering relation, cycle detection and prevention are facilitated. Cycles introduce logical contradictions, such as securities self-referencing as parents. Leveraging Alloy's capacity for transitive closure analysis, cycles can be automatically identified and invalidated. By conforming to these rules and utilizing Alloy's analytical capabilities, we uphold the integrity of share accounting, disallowing both negative and zero quantities.

3.3 Constructing the Model

Alloy, first introduced by Daniel Jackson in 2002 [13], is today a complete framework for modeling and analyzing software systems. The fundamental concept that underlies the Alloy environment is the notion that software development should be based on abstractions, and the process of programming should arise naturally from the design phase through the careful selection of appropriate abstractions.

Alloy is distributed as the Alloy Analyzer, which features:

- A modeling language
- A visualizer for meta-models and model instances
- A bounded model checker
- A standard library for common data structures (sequences, graphs, relations, etc.)

Through the integration of the aforementioned features, anyone using Alloy is able to efficiently engage in the iterative process of modeling, visually and comprehensively examine instances, and express and verify properties. Through the modeling of interactions between two or more states, an Alloy model can focus on more operational features of the system as well as the concepts underlying the system under consideration.

3.3.1 Parts of an Alloy model

Every alloy model is composed of signatures, predicates, facts, functions, assertions, and commands:

1. Signatures: the basic building blocks of Alloy models, represent entities in the domain. Signatures have *fields* that relate them to other signatures.
2. Functions: A function in Alloy relates instances of signatures to relations. An alloy function is a function like in any programming language.
3. Predicates: A predicate in Alloy is a function that returns a boolean value. Predicates are used to express the properties of the model.
4. Facts: A fact is a predicate that must always be true in the model. A fact actively constrains the model by discarding instances that do not satisfy it.
5. Assertions: An assertion is a predicate that can be given to a check command, and

expresses our *expectations* of the model. Assertions do not constrain the model effectively.

6. Commands (Checks and runs): The `run` finds *examples* of the model that satisfy all constraints, while the `check` command finds *counter-examples* that violate a given assertion.

A full presentation of Alloy is beyond the scope of this work but can be found in the Software Abstractions book by Daniel Jackson [14].

3.3.2 A brief overview of Alloy-related literature

Alloy is cited in more than 1,200 papers, with many extensions and applications in software design and modeling. Alloy∗ is a higher-order extension of Alloy that can be used for program synthesis, since it supports higher-order quantification [15]. *αRby* is an embedding of Alloy in Ruby [16].

A number of other modeling languages have been translated to or partially modeled in Alloy[17], including UML, $i^\star$, CVL, Event-B and, Z, OCL. Alloy can also be used in conjunction with verification languages such as ACL2[18] or Isabelle[19].

Alloy has been used in a wide range of applications in software engineering, database design, **security** analysis [20], [21], multiagent negotiations [22]. It has also been applied to modeling beyond computer science such as a model for central bank policy [23]. A model of the same-origin-policy used in web browsers can be found in the 500 Lines or Less open-source book [24].

We were not able to find previous models of ***capitalization tables*** *in Alloy.*

The model's capacity to accurately validate and determine the current state of the system without requiring the re-execution of all transactions is of utmost significance. This is a significant improvement compared to the original model. In contrast, the architecture proposed by the Open Cap Table Coalition does not provide adequate specifications for accurately determining the present condition of a capitalization table. This is a crucial requirement for the effective development of optimal tools. .

The Alloy environment significantly contributed to the iterative process of evaluating various design issues, including those outlined by the Open Cap Table Coalition, resulting in the development of the current model.

3.3.3 Methodology

Our methodology for model construction adopts an iterative approach, consistent with best practices for employing Alloy:

1. Initially, we isolate each discrete concept within the Open Cap Table Coalition's original model. Corresponding Alloy signatures are created for these concepts, and anticipated invariant properties are defined.
2. Subsequent to establishing initial properties and signatures, we assess their validity. In case a property is demonstrated to be inconsistent, we amend the relevant signature with additional constraints to rectify the discrepancy.
3. Instance model visualizations are rigorously examined to confirm alignment with our theoretical expectations and to identify any conspicuous inconsistencies.
4. This iterative process continues until the model fulfills our pre-established conditions and criteria.

Several factors contribute to the efficacy of our methodology. First, Alloy enables swift feedback loops, thereby streamlining the iterative development cycle. Second, the amalgamation of automated verifications and manual scrutiny serves as a robust validation mechanism. Additionally, Alloy's capacity to transform problems into boolean satisfiability issues and identify unsatisfiable subsets (UNSAT cores, in the literature) within the original constraints is invaluable for debugging and for isolating root causes when the model diverges from expected behavior.

Chapter 4

The New Model

This chapter presents a conceptual framework for a system intended for managing capitalization tables, focusing on the fundamental ideas of securities and transactions.

The proposed model serves as an initial framework for the development of validation tools and information management applications. Additionally, it lays the groundwork for code-normative documents, such as smart contracts.

4.1 Signatures for Securities, Transactions, and Stakeholders

In the model, signatures serve as the foundational entities. Specifically, we define signatures for three main categories: securities, transactions, and stakeholders. Within these, we further classify different types of securities and transactions through additional signatures.

The signatures for both securities and transactions are designed as abstracts. Subsequently, the signatures for their respective types are extensions of these abstract foundations. This structure adheres to a prevalent pattern in Alloy modeling.

Beyond securities and transactions, there are signatures for the vesting system, constraints, and checks (properties to be verified under a bound scope size).

4.1.1 Abstract security signature

All transactions and the two kinds of securities extend an abstract signature. The signature for abstract security is shown in Listing 5. It contains the relationships with transactions (initial and terminal), and the parent security (which might be none, in the case of issuances). Figure 4.1 illustrates a visual depiction of the signature's abstract security.

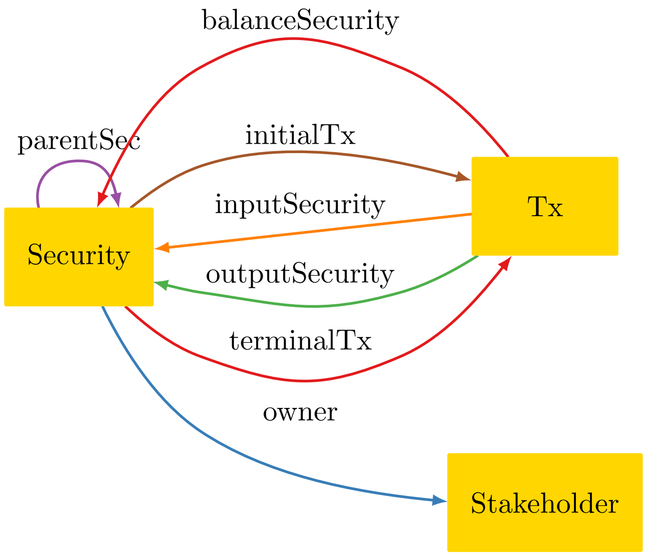

Figure 4.1: Metamodel of the abstract security

```
abstract sig Security {
    initialTx : one Tx,
    terminalTx : lone Tx,
    parentSec : lone Security,
    owner : one Stakeholder,
    id : disj one Id
}
```

Listing 5: Abstract signature for securities

The `Security` abstract signature is extended by the `StockTx` and `PlanTx` signatures, shown in Listings 6 and 7, respectively. Structural constraints are defined in the transaction signatures, while accounting constraints are defined in both.

A security must always represent some shares; instances with zero shares are not allowed. Take into account that there might not be zero shares in the event of a partial cancellation or transfer. In this scenario, the outcome entails a complete cancellation or transfer, resulting in the absence of newly generated securities, and thus, the absence of securities with zero shares.

```
sig StockSecurity extends Security {
  shares : Int
} {
  pos[shares]
}
```

Listing 6: Signature for a stock security

We enforce that the sum of granted, vested and exercised shares is always positive and that the number of vested shares is always less than or equal to the number of granted shares, which is always less than or equal to the number of exercised shares. This means that $0 \leq exercisedShares \leq vestedShares \leq grantedShares$.

```
sig PlanSecurity extends Security {
  grantedShares : Int,
  vestedShares : Int,
  exercisedShares : Int,
  conditions : set Condition
} {
  nonneg[exercisedShares]
  lte[exercisedShares, vestedShares]
  lte[vestedShares, grantedShares]
}
```

Listing 7: Signature for a plan security

4.1.2 Abstract signature for transactions

The signature for an abstract transaction is similar to the abstract security signature. It shows relationships to input, output, and balance securities, as well as to its parent transaction. In order to account for the possible existence of transactions having input, output, and balance securities, the *lone* keyword is employed to denote the optional nature of these relationships. The signature can be observed in Listing 8. Figure 4.2 depicts a visual representation of the signature.

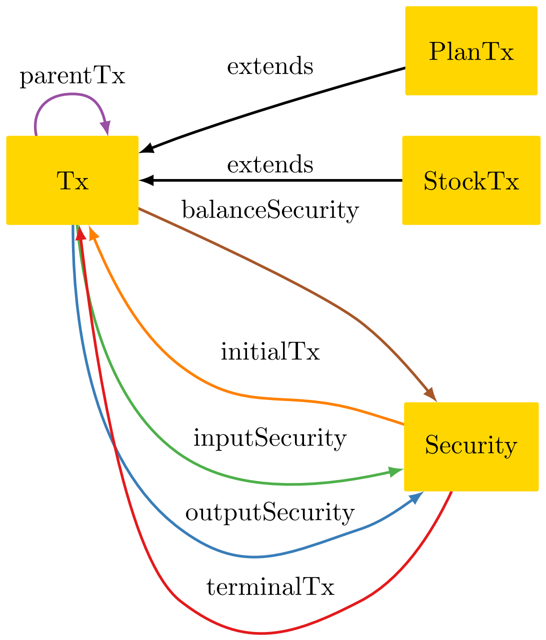

Figure 4.2: Metamodel of the transaction signature, including relationships between transactions and securities, and the extensions PlanTx and StockTx

```
abstract sig Tx {
  inputSecurity : disj lone Security,
  outputSecurity : disj lone Security,
  balanceSecurity : disj lone Security,
  parentTx : lone Tx,
  id : disj one Id
}
```

Listing 8: Abstract signature for a transaction

4.1.3 Concrete transactions signatures

We will consider nine different transactions in our model, sufficient for both stock and plan securities. The transactions are listed in Table 4.1. We show which transactions have (or do not have) input, output and balance securities. This shows how every transaction follows a general form. Issuance transactions exclusively generate new securities, whereas cancellation transfers consume the original security and, in the case of partial cancellation, may result in the creation of a balance security. Vesting operations result in the creation of a novel security that reflects revised quantities of shares that have been provided, vested, or exercised. Transfers invariably result in the establishment of a fresh security arrangement for the incoming stakeholder, and in cases of partial transfers, might result in a security balance. Engaging in exercises can be considered as an intriguing transaction, since it is the only transaction that involves the consumption of a planned security while producing a stock security.

Table 4.1: Transactions in the model

Transaction Type	**Input Security?**	**Output Security?**	**Balance Security?**
Stock Issuance	No	Yes	No
Plan Issuance	No	Yes	No
Stock Total Cancellation	Yes	No	No
Stock Total Transfer	Yes	Yes	No
Plan Vesting Event	Yes	Yes	No
Plan Vesting Date	Yes	Yes	No
Stock Partial Transfer	Yes	Yes	Yes
Stock Partial Cancellation	Yes	Yes	Yes
Plan Exercise	Yes	Yes	Yes

Figure 4.3 illustrates the graphical depiction of various forms of stock security transactions and how they are linked through the extension relation. The figure depicting the corresponding representation for plan security transactions is presented in Figure 4.4. It is important to realize that the only direct association between transactions is the parent relationship.

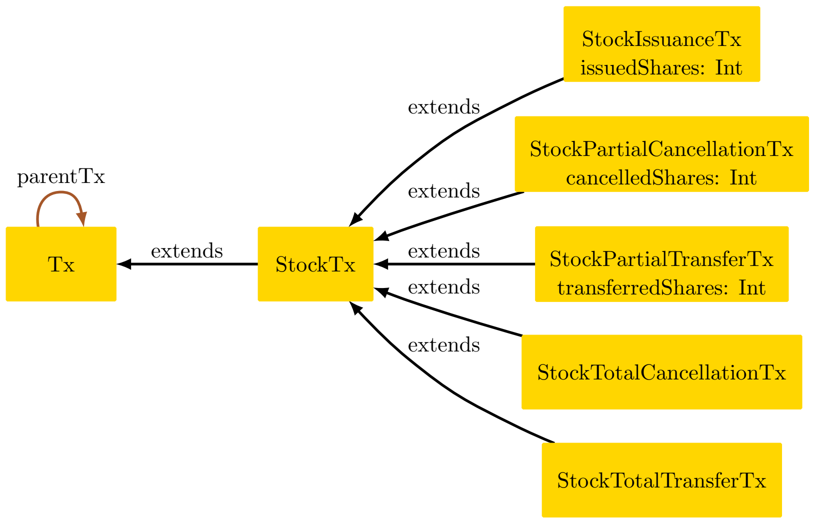

Figure 4.3: Metamodel of the stock transaction signature, showing the specific transaction types and their share fields

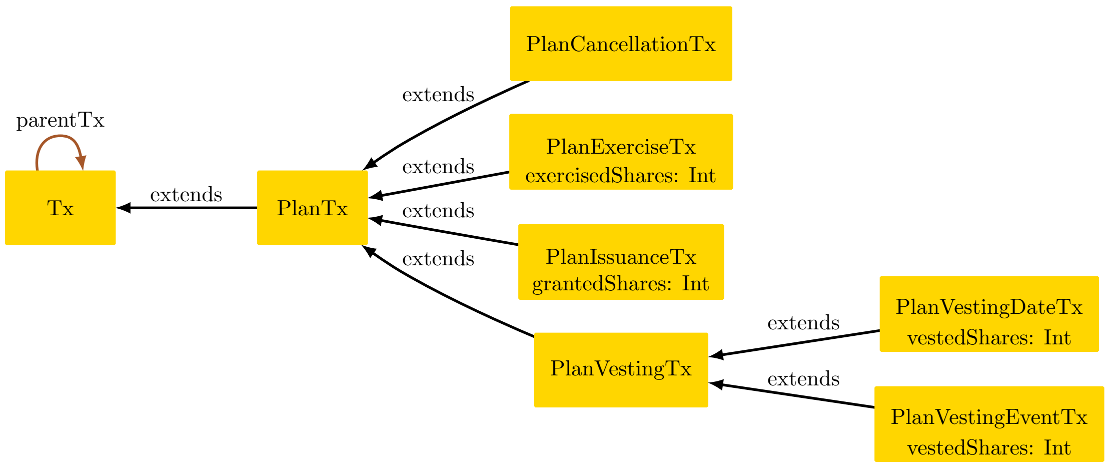

Figure 4.4: Metamodel of the plan transaction signature, showing the specific transaction types and their share fields

From the perspective of securities, transactions are associated through the `initialTx` and `terminalTx` relations. Conversely, from the standpoint of transactions, the `inputSecurity`, `outputSecurity`, and `balanceSecurity` relations establish connections to securities.

The `parentSec` and `parentTx` relations induce graphs over the domains of securities and transactions, respectively. Furthermore, the `initialTx` and `terminalTx` relations serve as the linking mechanisms between the graph of securities and the graph of transactions. The bipartite structure that emerges is illustrated in Figure 4.5.

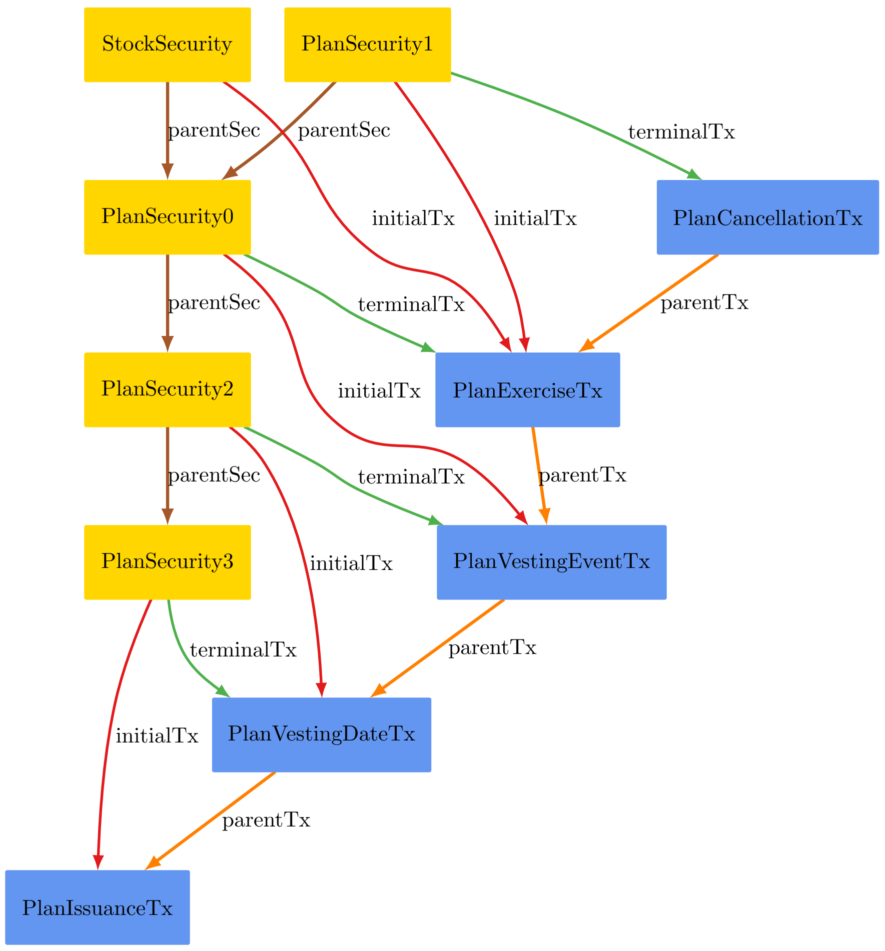

Figure 4.5: The graph formed by securities and transactions.

Now we will detail the specific signatures for each transaction type, and provide examples of instances where those transactions are used.

The Stock Security Issuance Transaction

Stock securities are instantiated by Stock Issuance Transactions, which inject a specified number of shares, denoted as `issuedShares`, into the system. These transactions allocate the newly created security to a designated stakeholder, referred to as `issuedTo`. Additionally, stock securities can be introduced through plan security exercises, as elaborated in

Section 4.1.3.

The restrictions outlined in the signature state that a Stock Issuance Transaction is linked to a single stock-type output security. This stock security designates the issuance transaction as its initial transaction. The subsequent constraint block stipulates that no other securities are linked to this particular transaction. The third constraint block maps the attributes of the security to the corresponding attributes in the transaction. Since this operation is related to an issuance, there is no requirement for input security or a parent transaction. Finally, the accounting requirement is unambiguous, requiring that the quantity of shares be greater than zero. The source code for the Alloy signature is depicted in Listing 9.

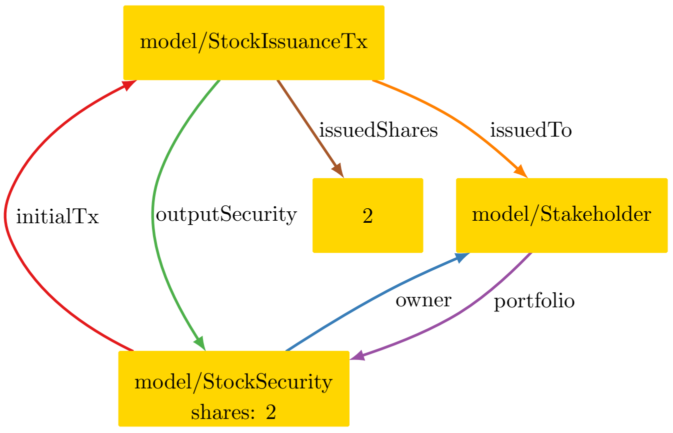

Figure 4.6: An example instance of the stock issuance transaction

```
sig StockIssuanceTx extends StockTx {
  issuedTo : one Stakeholder,
  issuedShares : Int
} {
  no inputSecurity && no balanceSecurity && one outputSecurity
  outputSecurity in StockSecurity
  no outputSecurity.parentSec
  outputSecurity.initialTx = this

  all s : Security | s.initialTx = this iff s = outputSecurity
  no s : Security | s.terminalTx = this

  outputSecurity.owner = issuedTo
  outputSecurity.shares = issuedShares

  no parentTx

  pos[issuedShares]
}
```

Listing 9: The stock issuance transaction signature

The Stock Security Partial Cancellation Transaction

In Stock Partial Cancellation Transactions, represented as `StockPartialCancellationTx`, the model extends `StockTx` by introducing `cancelledShares` to indicate the shares intended for cancellation. The constraints in the signature articulate that each transaction is linked to exactly one `balanceSecurity` and `inputSecurity`, while disallowing any `outputSecurity`. Additionally, both `inputSecurity` and `balanceSecurity` must be of the `StockSecurity` type. The model sets `balanceSecurity` as parented to `inputSecurity`, and uniquely ties their `initialTx` and `terminalTx` attributes to this transaction. Figure 4.7 presents an instance of this transaction.

Accounting constraints further refine the model's rigor, and are a bit more involved than in the Stock Issuance Transaction. Specifically, `cancelledShares` must be positive and less than the total shares in `inputSecurity`. The resulting `balanceSecurity` shares are calculated as the difference between the shares in `inputSecurity` and `cancelledShares`, and this value also needs to be greater than zero. Listing 10 shows the source code for this signature.

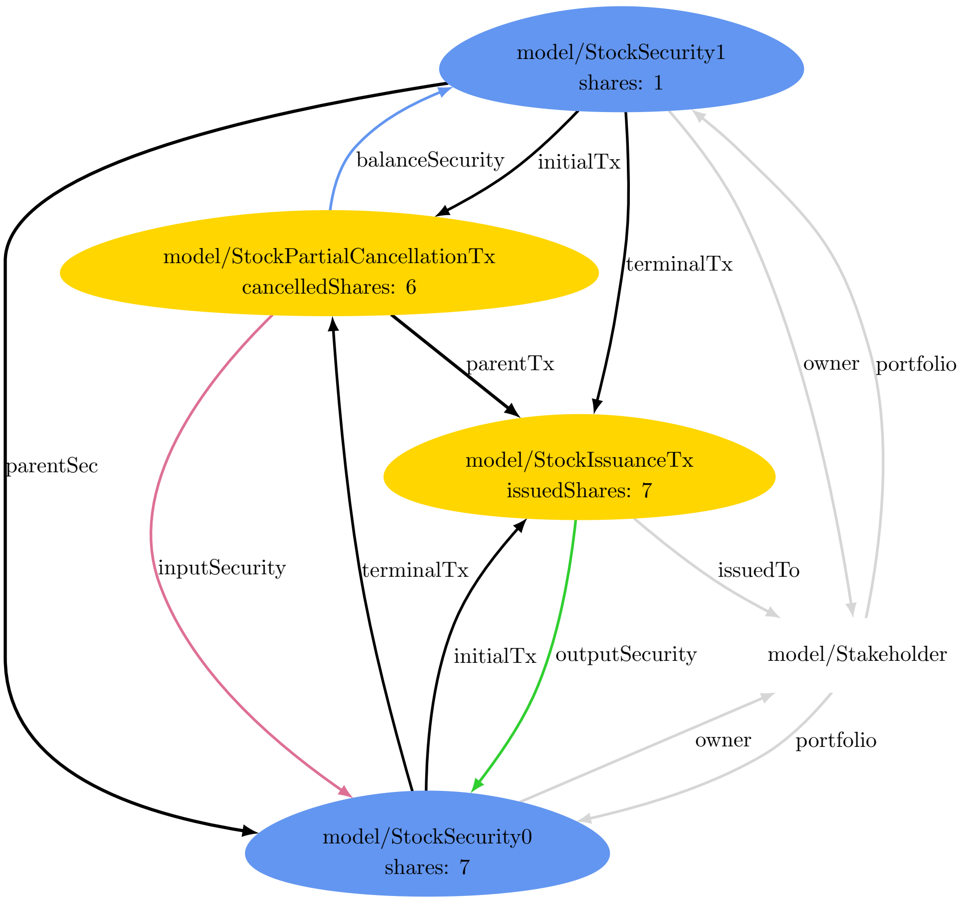

Figure 4.7: An example instance of the stock partial cancellation transaction

```
sig StockPartialCancellationTx extends StockTx {
  cancelledShares : Int
} {
  one balanceSecurity && one inputSecurity && no outputSecurity
  inputSecurity in StockSecurity && balanceSecurity in StockSecurity
  balanceSecurity.parentSec = inputSecurity


  balanceSecurity.initialTx = this
  inputSecurity.terminalTx = this

  no s : Security - balanceSecurity | s.initialTx = this
  no s : Security - inputSecurity | s.terminalTx = this

  balanceSecurity.owner = inputSecurity.owner

  parentTx = inputSecurity.initialTx

  pos[cancelledShares]
  lt[cancelledShares, inputSecurity.shares]
  eq[balanceSecurity.shares, sub[inputSecurity.shares, cancelledShares]]
  pos[balanceSecurity.shares]

}
```

Listing 10: The stock partial cancellation transaction signature

The Stock Security Partial Transfer Transaction

In the Stock Partial Transfer Transactions, represented as `StockPartialTransferTx`, the model extends the foundational `StockTx` to include `transferredTo`, denoting the receiving stakeholder, and `transferredShares`, specifying the quantity of shares transferred. According to the signature constraints, each transaction is associated with exactly one `balanceSecurity`, `inputSecurity`, and `outputSecurity`, all of which must be of the `StockSecurity` type. These securities are interconnected in such a way that `outputSecurity` and `balanceSecurity` are both parented to `inputSecurity`, and their `initialTx` attributes are set to this transaction. Meanwhile, the `terminalTx` of `inputSecurity` is also specified as this transaction. Figure 4.8 shows an instance of a stock partial transfer.

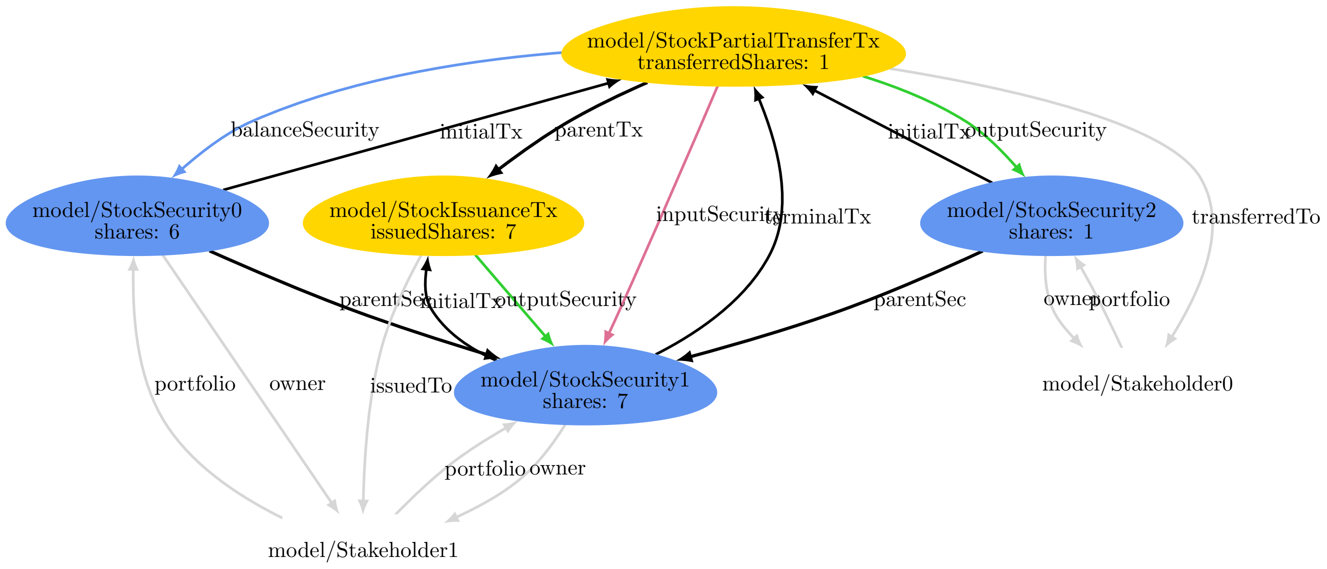

Figure 4.8: An example instance of the stock partial transfer transaction

The model imposes additional accounting and ownership constraints for internal consistency. Specifically, `transferredShares` must be a positive integer and less than the total shares in `inputSecurity`. The securities resulting from the transfer—the `outputSecurity` and `balanceSecurity`—are determined based on `transferredShares` and the remaining shares in `inputSecurity`, respectively. Both resulting securities must have a positive number of shares. Ownership rules are also enforced: `outputSecurity` is owned by the `transferredTo` stakeholder, while `balanceSecurity` retains the same owner as `inputSecurity`. The source code is available in Listing 11.

```
sig StockPartialTransferTx extends StockTx {
  transferredTo : one Stakeholder,
  transferredShares : Int
} {

  one balanceSecurity && one inputSecurity && one outputSecurity
  inputSecurity in StockSecurity && outputSecurity in StockSecurity && balanceSecurity in
↪  StockSecurity
  outputSecurity.parentSec = inputSecurity && balanceSecurity.parentSec = inputSecurity
  balanceSecurity.initialTx = this && outputSecurity.initialTx = this
  inputSecurity.terminalTx = this

  all s : Security | s.initialTx = this implies (s = outputSecurity or s =
↪  balanceSecurity)
  all s : Security | s.terminalTx = this implies s = inputSecurity

  outputSecurity.owner != inputSecurity.owner && balanceSecurity.owner =
↪  inputSecurity.owner && outputSecurity.owner = transferredTo
  outputSecurity.shares = transferredShares

  parentTx = inputSecurity.initialTx

  pos[transferredShares]
  lt[transferredShares, inputSecurity.shares]
  eq[outputSecurity.shares, transferredShares]
  eq[balanceSecurity.shares, sub[inputSecurity.shares, transferredShares]]
  pos[outputSecurity.shares]
  pos[balanceSecurity.shares]
  eq[balanceSecurity.shares, sub[inputSecurity.shares, transferredShares]]
}
```

Listing 11: The stock partial transfer transaction signature

The Stock Security Total Cancellation Transaction

In Stock Total Cancellation Transactions, encapsulated as `StockTotalCancellationTx`, the model defines that each transaction is exclusively associated with a single `inputSecurity` of the `StockSecurity` type. The schema expressly prohibits the inclusion of either `outputSecurity` or `balanceSecurity`, and assigns this transaction as the `terminalTx` for the `inputSecurity` involved.

Figure 4.9 illustrates a representative instance of this transaction, and Listing 12 provides the corresponding source code. As depicted, the complexity of this particular transaction is markedly reduced compared to other transaction types.

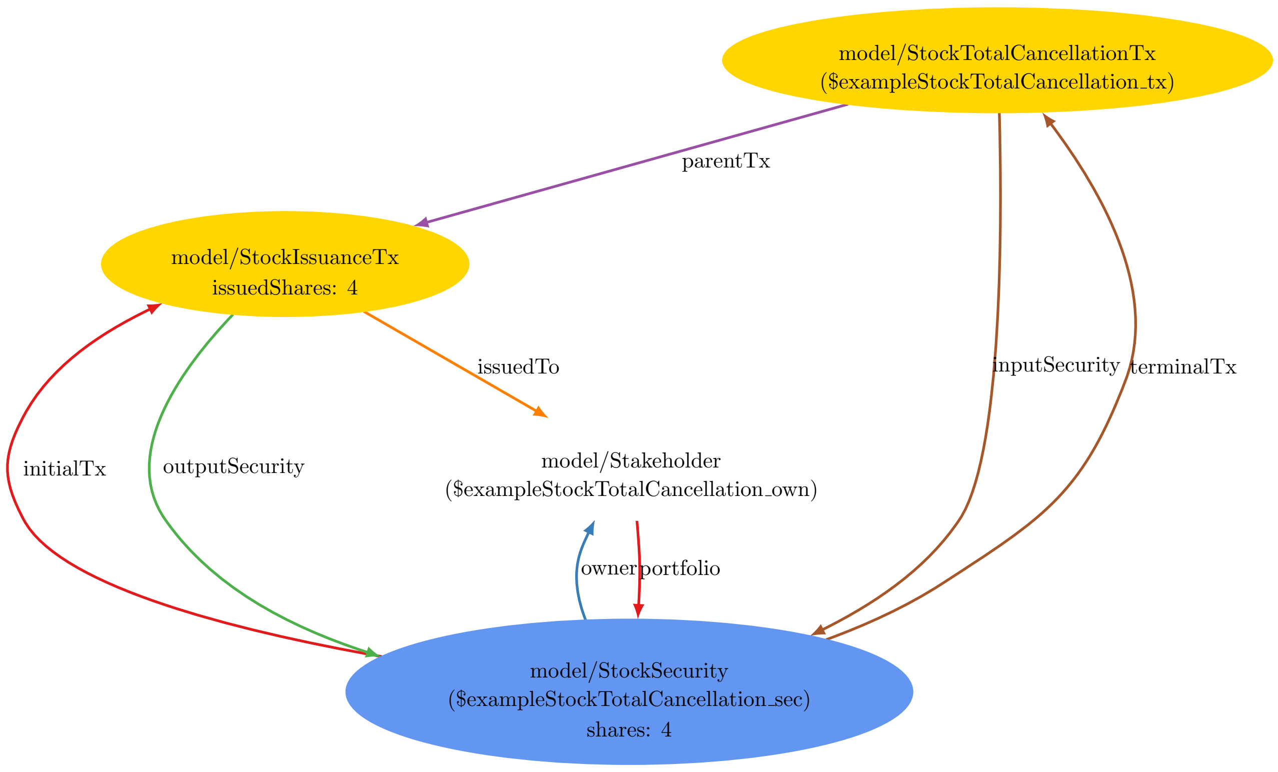

Figure 4.9: An example instance of the stock total cancellation transaction

```
sig StockTotalCancellationTx extends StockTx {
} {
  one inputSecurity && no outputSecurity && no balanceSecurity
  inputSecurity in StockSecurity
  inputSecurity.terminalTx = this

  no s : Security | s.initialTx = this
  all s : Security | s.terminalTx = this implies s = inputSecurity

  parentTx = inputSecurity.initialTx
}
```

Listing 12: The stock total cancellation transaction signature

The Stock Security Total Transfer Transaction

In Stock Total Transfer Transactions, denoted as `StockTotalTransferTx`, the model extends `StockTx` by introducing `transferredTo` to specify the receiving stakeholder. According to the constraints, each transaction uniquely links to one `inputSecurity` and one `outputSecurity`, while deliberately excluding any `balanceSecurity`. Both `inputSecurity` and `outputSecurity` are required to be of the `StockSecurity` type. The `outputSecurity` is parented to the `inputSecurity` and has its `initialTx` attribute set to this transaction, while the `inputSecurity`'s `terminalTx` is also defined as this transaction.

Figure 4.10 presents an instance of the transaction, and Listing 13 presents the source code.

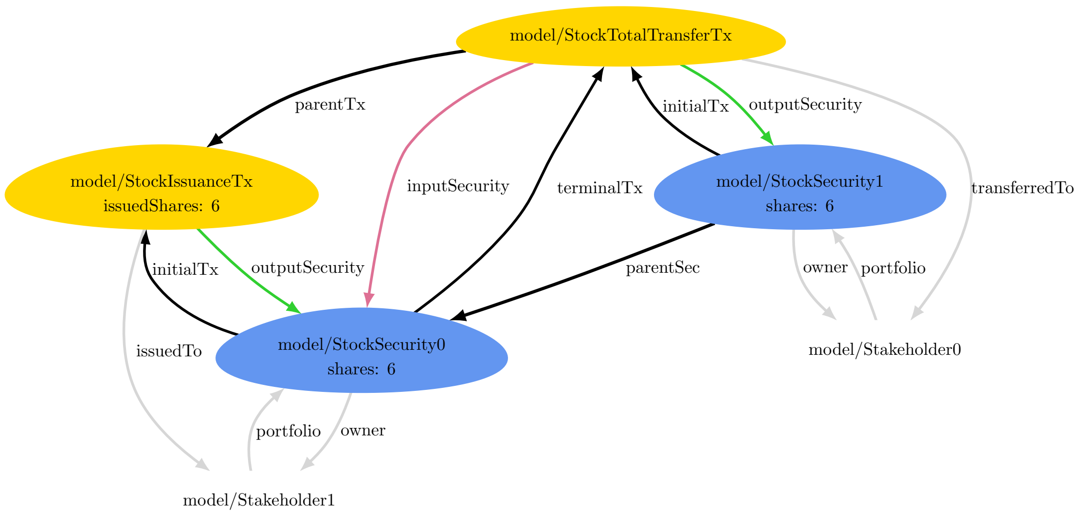

Figure 4.10: An example instance of the stock total transfer transaction

```
sig StockTotalTransferTx extends StockTx {
  transferredTo : one Stakeholder
} {
  no balanceSecurity && one inputSecurity && one outputSecurity
  inputSecurity in StockSecurity && outputSecurity in StockSecurity
  outputSecurity.parentSec = inputSecurity
  outputSecurity.initialTx = this
  inputSecurity.terminalTx = this

  all s : Security | s.initialTx = this iff s = outputSecurity
  all s : Security | s.terminalTx = this iff s = inputSecurity

  outputSecurity.owner = transferredTo && inputSecurity.owner != transferredTo

  parentTx = inputSecurity.initialTx

  pos[outputSecurity.shares]
  eq[outputSecurity.shares, inputSecurity.shares]
}
```

Listing 13: The stock total transfer transaction signature

The Plan Security Issuance Transaction

The Plan Security Issuance transaction exhibits greater complexity compared to standard stock security transactions due to its adherence to a vesting system. As elaborated in

Chapter 2, the vesting system establishes temporal regulations for the gradual release of securities, commonly observed in employee stock option grants.

In order to maintain accuracy and consistency between the specified number of shares in the conditions (see Section 4.2) and the number of shares granted, it is imperative to include a verification phase in the transaction process. Considering that shares are subjected to a vesting schedule, which must be completed before they may be exercised, the initial count of vested and exercised shares is set at zero.

In the Alloy model, `PlanIssuanceTx` extends `PlanTx` and specifies additional fields such as `issuedTo`, `grantedShares`, and `conditions`. The constraints require that there is no `inputSecurity`, exactly one `outputSecurity`, and no `balanceSecurity`. Moreover, `outputSecurity` must belong to the category of `PlanSecurity` and should not have a `parentSec`. Other constraints ensure the consistency and integrity of the transaction, such as the initialization of `vestedShares` and `exercisedShares` to zero and the requisite equality between `vestedShares` and the sum of `conditionedShares` specified in the `conditions`.

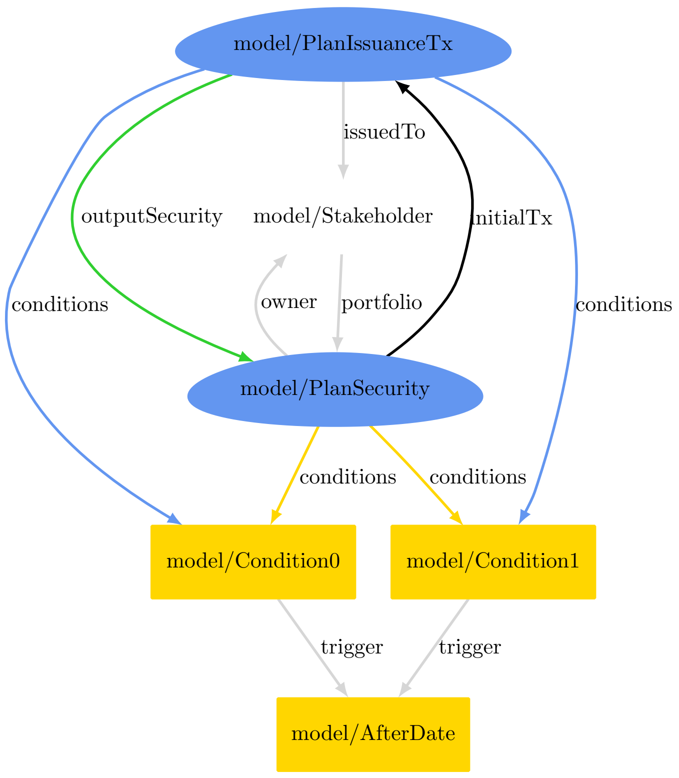

Figure 4.11: An example instance of the plan issuance transaction

Figure 4.11 shows an example instance of a plan security issuance transaction, and Listing 14 presents the source code.

```
sig PlanIssuanceTx extends PlanTx {
  issuedTo : one Stakeholder,
  grantedShares : Int,
  conditions : set Condition
} {
  no inputSecurity && one outputSecurity && no balanceSecurity
  outputSecurity in PlanSecurity
  no outputSecurity.parentSec
  outputSecurity.initialTx = this

  no s : Security | s.terminalTx = this
  all s : Security | s.initialTx = this implies s = outputSecurity

  outputSecurity.owner = issuedTo
  outputSecurity.grantedShares = grantedShares
  outputSecurity.conditions = conditions

  no parentTx

  zero[outputSecurity.vestedShares]
  zero[outputSecurity.exercisedShares]

  let x = sum c : outputSecurity.conditions | c.conditionedShares {
    eq[outputSecurity.vestedShares, x]
  }
}
```

Listing 14: The plan issuance transaction signature

The Plan Security Cancellation Transaction

This signature is directly analogous to the total stock security cancellation (see Section 4.1.3). Figure 4.12 shows an example instance, and Listing 15 presents the Alloy source code.

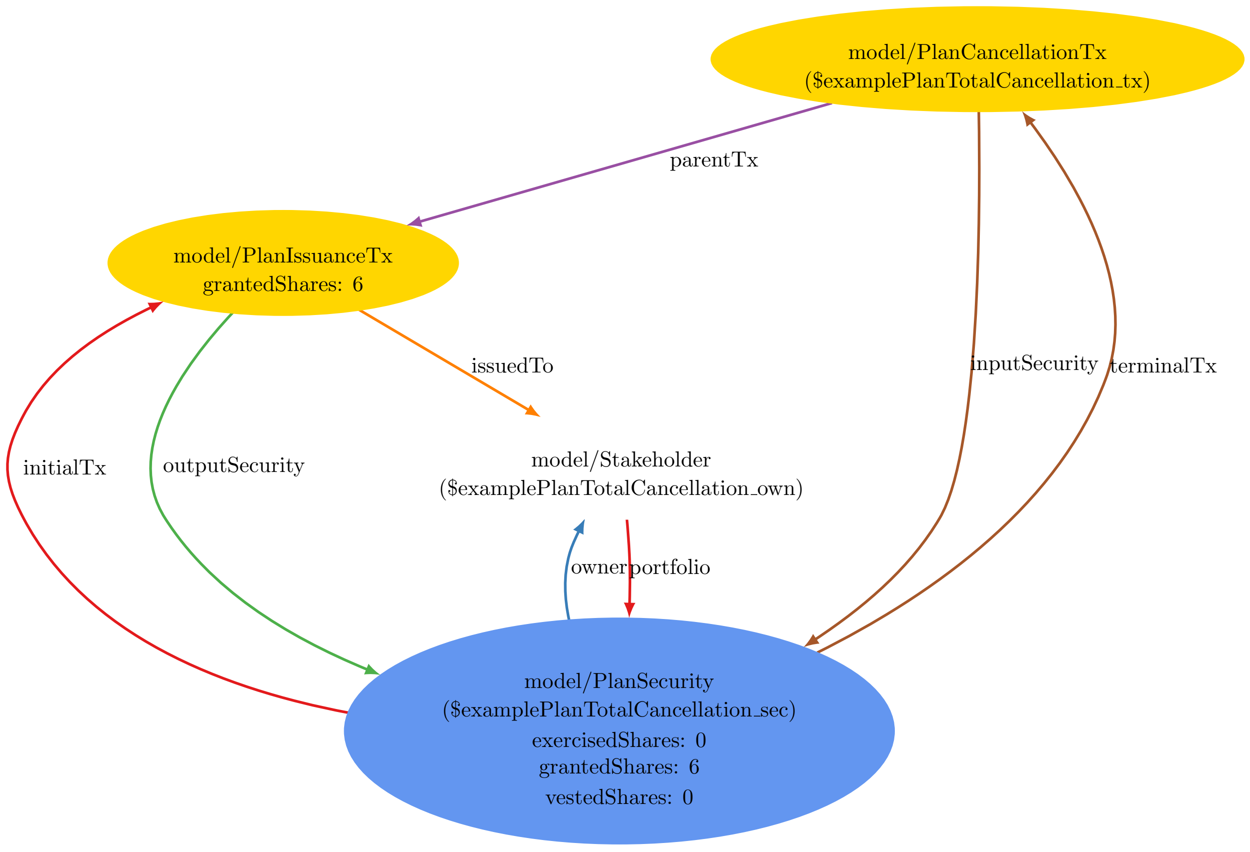

Figure 4.12: An example instance of the plan cancellation transaction

```
sig PlanCancellationTx extends PlanTx {
} {
  one inputSecurity && no outputSecurity && no balanceSecurity
  inputSecurity in PlanSecurity
  inputSecurity.terminalTx = this

  all s : Security | s.terminalTx = this iff s = inputSecurity
  no s : Security | s.initialTx = this

  parentTx = inputSecurity.initialTx

  lt[inputSecurity.exercisedShares, inputSecurity.grantedShares]
}
```

Listing 15: The plan cancellation transaction signature

The Plan Security Exercise Transaction

The Exercise Transaction for plan securities adheres to the foundational principles and methodologies common to other transactions. However, it necessitates added accounting verifications to manage distinct metrics, such as exercised shares, vested shares, and granted shares. This increased scrutiny is required due to the intricate structure of plan securities.

```
sig PlanExerciseTx extends PlanTx {
  exercisedShares : Int
} {
  one inputSecurity && one outputSecurity && one balanceSecurity
  inputSecurity in PlanSecurity && balanceSecurity in PlanSecurity && outputSecurity in
↪  StockSecurity
  outputSecurity.parentSec in inputSecurity && balanceSecurity.parentSec = inputSecurity
  outputSecurity.initialTx = this && balanceSecurity.initialTx = this
  inputSecurity.terminalTx = this

  all s : Security | s.initialTx = this iff (s = outputSecurity or s = balanceSecurity)
  all s : Security | s.terminalTx = this iff s = inputSecurity

  outputSecurity.owner = inputSecurity.owner && balanceSecurity.owner =
↪  inputSecurity.owner
  outputSecurity.shares = exercisedShares

  parentTx = inputSecurity.initialTx

  eq[balanceSecurity.exercisedShares, add[inputSecurity.exercisedShares, exercisedShares]]
  eq[balanceSecurity.vestedShares, inputSecurity.vestedShares]
  eq[balanceSecurity.grantedShares, inputSecurity.grantedShares]
}
```

Listing 16: The plan exercise transaction signature

Figure 4.13 illustrates a sample instance of this transaction, while Listing 16 provides the corresponding source code.

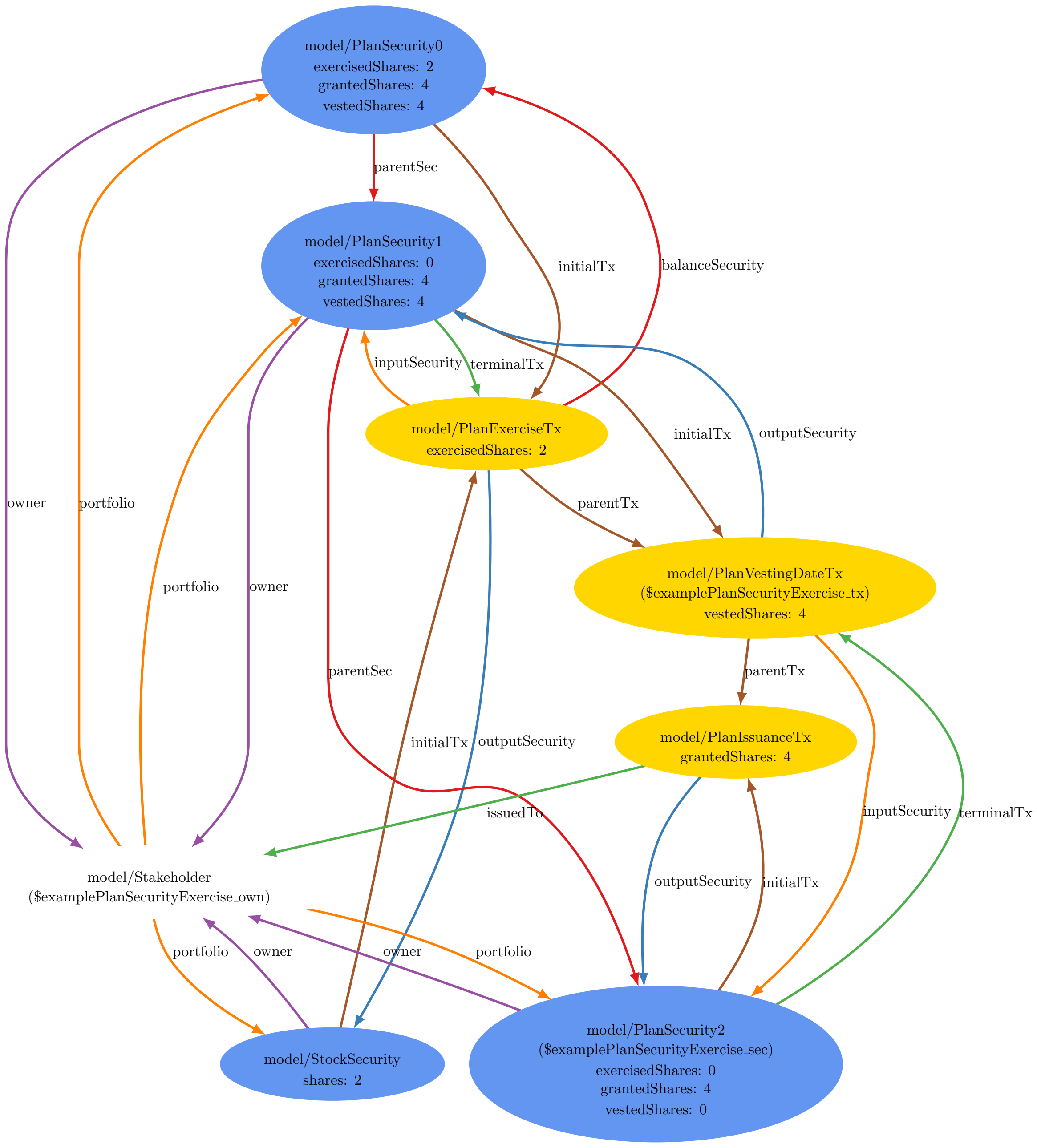

Figure 4.13: An example instance of the plan exercise transaction

The Plan Security Vesting Transaction

The Vesting Transaction for plan securities is the most intricate within the system. When the transaction occurs, any conditions that are triggered by the specified date or event are marked as satisfied and subsequently removed from the resultant, updated plan security transaction. As these conditions are fulfilled, the corresponding shares transition into a vested state, thereby becoming eligible for exercise. Apart from this specialized treatment of conditional triggers, the transaction aligns with the typical structure of other transactions, featuring one input and one output security.

The specific triggers and a presentation of the vesting system is provided in Section 4.2.

Listings 17 provides the basic signature for vesting transactions.

```
abstract sig PlanVestingTx extends PlanTx {
  vestedShares : Int,
  satisfiedConditions : set Condition
} {
}
```

Listing 17: The plan vesting transaction signature

The Plan Security Vesting Date Transaction

The Plan Security Vesting Date Transaction serves as an indicator that a specified date has elapsed, potentially satisfying certain triggers embedded within the conditions of the plan securities. Figure 4.14 and Listing 18 respectively showcase an example instance and the source code corresponding to this transaction signature.

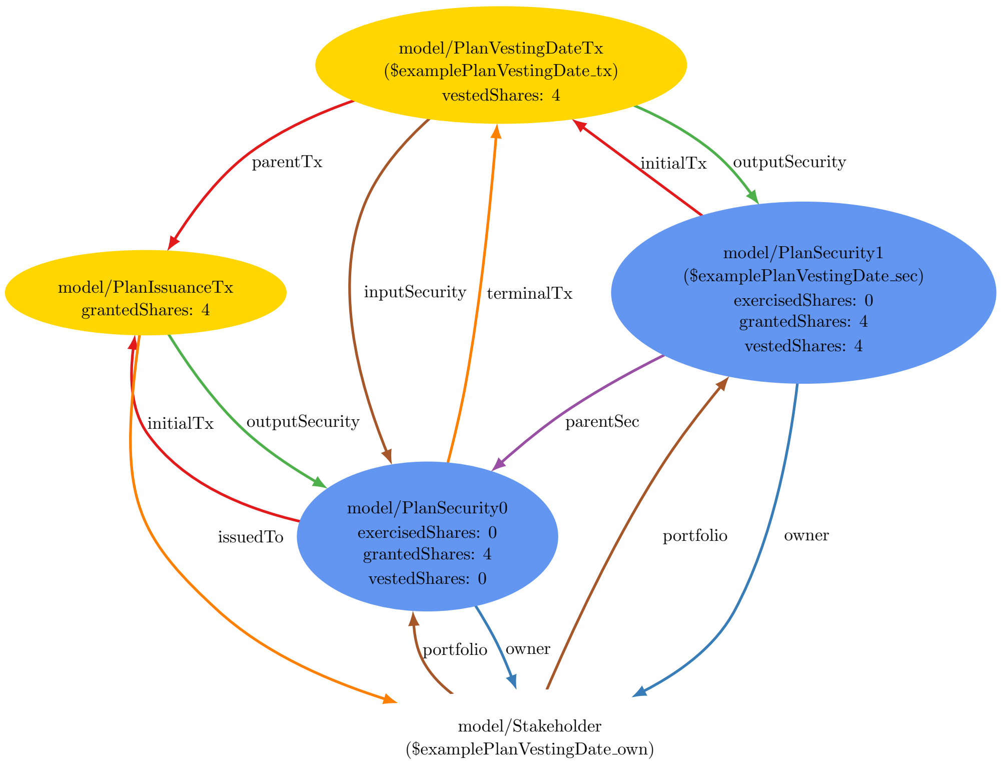

Figure 4.14: An example instance of the plan vesting date transaction

```
sig PlanVestingDateTx extends PlanVestingTx {
  vestingDate : Date
} {
  one inputSecurity && one outputSecurity && no balanceSecurity
  inputSecurity in PlanSecurity
  outputSecurity in PlanSecurity
  outputSecurity.parentSec = inputSecurity
  outputSecurity.initialTx = this
  inputSecurity.terminalTx = this

  all s : Security | s.initialTx = this iff s = outputSecurity
  all s : Security | s.terminalTx = this iff s = inputSecurity

  outputSecurity.owner = inputSecurity.owner

  parentTx = inputSecurity.initialTx

  satisfiedConditions = { c : inputSecurity.conditions |
↪  isTriggerSatisfiedByDate0[c.trigger, vestingDate] }
  outputSecurity.conditions = inputSecurity.conditions - satisfiedConditions

  pos[vestedShares]
  eq[outputSecurity.vestedShares, add[inputSecurity.vestedShares, vestedShares]]
  eq[outputSecurity.grantedShares, inputSecurity.grantedShares]
  eq[outputSecurity.exercisedShares, inputSecurity.exercisedShares]
  eq[sum[satisfiedConditions.conditionedShares], vestedShares]
}
```

Listing 18: The plan vesting date transaction signature

The Plan Security Vesting Event Transaction

Analogously to the vesting date transaction, The Plan Security Vesting Event Transaction serves as an indicator that a specified event has occurred. Figure 4.15 and Listing 19 respectively showcase an example instance and the source code corresponding to this transaction signature.

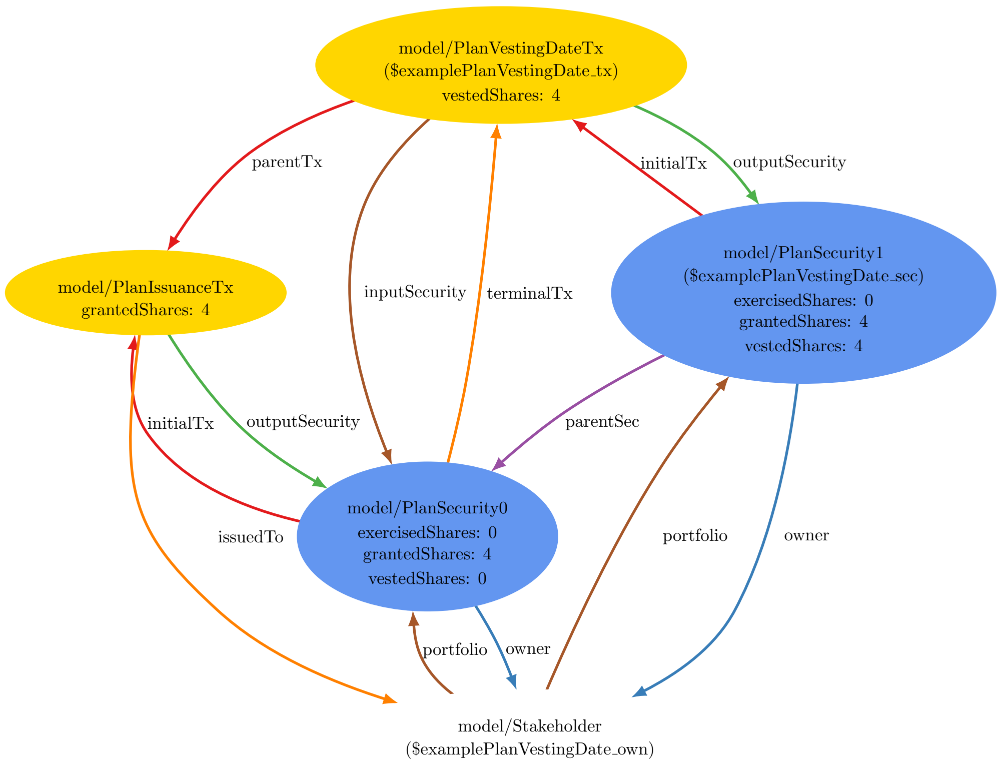

Figure 4.15: An example instance of the plan vesting event transaction

```
sig PlanVestingEventTx extends PlanVestingTx {
  event : Event
} {
  one inputSecurity && one outputSecurity && no balanceSecurity
  inputSecurity in PlanSecurity && outputSecurity in PlanSecurity
  outputSecurity.parentSec = inputSecurity
  outputSecurity.initialTx = this
  inputSecurity.terminalTx = this

  all s : Security | s.initialTx = this iff s = outputSecurity
  all s : Security | s.terminalTx = this iff s = inputSecurity

  outputSecurity.owner = inputSecurity.owner
  satisfiedConditions = { c : inputSecurity.conditions |
↪  isTriggerSatisfiedByEvent0[c.trigger, event] }
  outputSecurity.conditions = inputSecurity.conditions - satisfiedConditions

  parentTx = inputSecurity.initialTx

  pos[vestedShares]
  eq[outputSecurity.vestedShares, add[inputSecurity.vestedShares, vestedShares]]
  eq[outputSecurity.grantedShares, inputSecurity.grantedShares]
  eq[outputSecurity.exercisedShares, inputSecurity.exercisedShares]
  eq[vestedShares, sum[satisfiedConditions.conditionedShares]]
}
```

Listing 19: The plan vesting event transaction signature

4.1.4 Constraints

Beyond the constraints provided as part of signatures, we enforce a few more constraints as stand-alone facts to ensure that the securities and transactions are consistent with each other.

Two very important facts, `orderingParentSec` and `orderingParentTx`, ensure that the parent relations are acyclic by establishing the partial ordering between securities and securities, and transactions and transactions. This is shown in Listings 20 and 21.

As we establish that children are always $\geq$ than their parents, we ensure that there are no cycles in the graph induced by the `parentSec` and `parentTx` relations. This is an example where Alloy's expressiveness shines, as we can express the fact in a very concise manner.

```
fact orderingParentSec {
  all sec : Security {
    some sec.parentSec implies {
      lt[sec.parentSec, sec]
    }
  }
}
```

Listing 20: Ordering of parent securities

```
fact orderingParentTx {
  all tx : Transaction {
    some tx.parentTx implies {
      lt[tx.parentTx, tx]
    }
  }
}
```

Listing 21: Ordering of parent transactions

By stating all the signatures and constraints we have defined the possible instances of the model. But the model is only interesting if it possesses the properties that we expect to hold. We will now define the properties that we expect to hold in the model.

4.2 Vesting system

The original OCF vesting system can handle at most a combination of dates and events. We support both triggers for vesting, but add operators for the conjunction, disjunction and negation of triggers. The vesting trigger for vesting after a date thus can be negated and express conditions that might happen before a date.

In our model, each `PlanSecurity` has a `Condition`, which has a `Trigger` and a number of shares `conditionedShares`. The triggers form a simple expression language with two base cases (dates and events) and 3 propositional logic operators. The metamodel for our improved vesting system is shown in Figure 4.16.

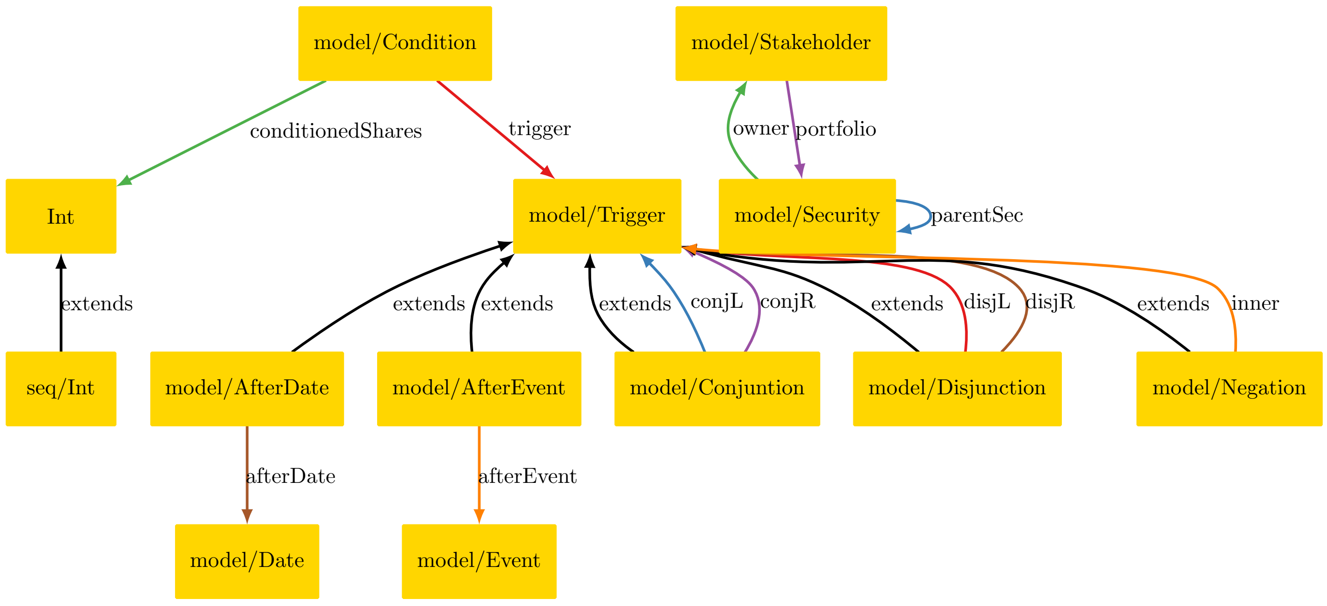

Figure 4.16: Metamodel of the vesting system.

4.2.1 Conditions

The structure of a condition is straightforward and serves the purpose of associating a quantity of shares, denoted as `conditionedShares`, with specific triggers. The signature for such a condition is displayed in Listing 22. Its primary function is to link a specific number of shares—`conditionedShares`—with a `trigger` that, when satisfied, allows for the release of those shares. It's important to note that the `Condition` signature is the one referenced within plan security transactions. The source code representation for the condition is also provided in Listing 22.

```
sig Condition {
  conditionedShares : Int,
  trigger : one Trigger
} {
  pos[conditionedShares]
}
```

Listing 22: Signature of a condition.

4.2.2 Triggers

The triggers are designed to be straightforward signatures as well. A `_children` relation is introduced to guarantee that the trigger expressions remain acyclic (by pairing it with a constraint). The signature for a trigger is presented in Listing 23. Signatures for additional types of triggers are depicted in Listings 24 and 25.

```
abstract sig Trigger {
  _children : set Trigger
}
```

Listing 23: Signature of a trigger.

4.2.3 Event and Date-related

Basic triggers necessitate either the occurrence of an event or the passing of a specific date. These triggers are categorized as ”after” triggers, meaning they are activated following the event or date in question. These triggers can also be negated and combined either conjunctively or disjunctively, as discussed in Section 4.2.4. This added flexibility significantly expands the scope of possibilities compared to the original model, especially since the negation of an ”after” trigger effectively translates to ”until”. Listings 24 and 25 present code for both base triggers.

Given that the triggers constitute an expression language subject to recursive evaluation (as detailed in Section 4.3), these two basic triggers serve as the base cases for the evaluation process.

```
sig AfterEvent extends Trigger {
  afterEvent : Event
} {
  no _children
}
```

Listing 24: Signature of an `AfterEvent` trigger.

```
sig AfterDate extends Trigger {
  afterDate : Date
} {
  no _children
}
```

Listing 25: Signature of an `AfterDate` trigger.

4.2.4 Propositional logic

In addition to the basic triggers, logical operators "AND," "NOT," and "OR" are introduced. As previously noted, the negation operator, when applied to an "after date" or "event," means "until". Through the use of conjunction, it becomes possible to express vestings that are conditioned upon both achieving a milestone and meeting a deadline. The original model was limited to expressing only conjunctions of triggers, lacking the capability for disjunction and negation. The source code for these extended trigger types is provided in Listings 26, 28, and 27.

```
sig Conjunction extends Trigger {
  conjL : one Trigger,
  conjR : one Trigger
} {
  _children = conjL + conjR
}
```

Listing 26: Signature of a `Conjunction` trigger.

```
sig Disjunction extends Trigger {
  disjL : one Trigger,
  disjR : one Trigger
} {
  _children = disjL + disjR
}
```

Listing 27: Signature of a `Disjunction` trigger.

```
sig Negation extends Trigger {
  inner : one Trigger
} {
  _children = inner
}
```

Listing 28: Signature of a `Negation` trigger.

4.3 Unrolled evaluation function

Alloy is limited in its ability to natively support recursion. To address this, we adopt a strategy of unrolling the recursion through varying depths until we reach the base case, as exemplified in Listing 29. For the scope of this study, a tenfold unrolling of the function is deemed adequate. This unrolled function is employed within the `isTriggerSatisfiedByEventN` predicates, which are designed to verify the fulfillment of specific conditions. The methodology echoes techniques used in model finders for code synthesis by bounding loop depths.

In the source code, the base case—termed `isTriggerSatisfiedByEvent0`—serves as the point where events and dates are actually verified.

```
pred isTriggerSatisfiedByEvent0[t : Trigger, e : Event] {
  (
  isTriggerSatisfiedByEvent_BASE_CASE[t, e]
  ) or (
  t in Conjunction && isTriggerSatisfiedByEvent1[t.conjL, e] &&
↪  isTriggerSatisfiedByEvent1[t.conjR, e]
  ) or (
  t in Disjunction && isTriggerSatisfiedByEvent1[t.disjL, e] ||
↪  isTriggerSatisfiedByEvent1[t.disjR, e]
  ) or (
  t in Negation && not isTriggerSatisfiedByEvent1[t.inner, e]
  )
}

// until...
pred isTriggerSatisfiedByEvent9[t : Trigger, e : Event] {
  (
  isTriggerSatisfiedByEvent_BASE_CASE[t, e]
  ) or (
  t in Conjunction && isTriggerSatisfiedByEvent10[t.conjL, e] &&
↪  isTriggerSatisfiedByEvent10[t.conjR, e]
  ) or (
  t in Disjunction && isTriggerSatisfiedByEvent10[t.disjL, e] ||
↪  isTriggerSatisfiedByEvent10[t.disjR, e]
  ) or (
  t in Negation && not isTriggerSatisfiedByEvent10[t.inner, e]
  )
}

// Base case
pred isTriggerSatisfiedByEvent_BASE_CASE[t : Trigger, e : Event] {
  t not in AfterDate
  t in AfterEvent && t.afterEvent = e
}
```

Listing 29: Unrolled evaluation function.

4.4 Checks

The model's integrity is validated through a comprehensive suite of checks, segmented into accounting, count, and structure checks.

Accounting Checks, elaborated in Section 4.4.1, perform arithmetic verifications that are relevant to the quantification of shares.

Count Checks, discussed in Section 4.4.2, validate the accurate cardinality of instances for each type, providing substantial analytical utility despite their conceptual straightforwardness. For example, the quantity of `initialTx` relations should align with the number of securities.

Structure Checks, expounded in Section 4.4.3, examine the topological soundness of the securities and transactions graph, assuring, for instance, the absence of cyclical structures.

Model validation is facilitated by the Alloy Analyzer, which utilizes bounded-model checking by translating the Alloy model into an isomorphic SAT problem. In this SAT framework, each solution corresponds to a unique model instance. The Alloy Analyzer confirms the properties under specification by identifying counterexamples, thus offering a robust methodology for model verification.

4.4.1 Accounting checks

Accounting identities serve to validate the model. These identities are domain-derived and are implemented as checks within the model. These identities operate based on the following categorizations of shares at various junctures in the model.

In the context of plan securities, shares are classified into exercised, vested, and granted categories. The categorization for shares related to plan securities is delineated in Listing 30.

```
fun _exercisedShares : set Int {
  sum i : PlanExerciseTx | i.exercisedShares
}

fun _vestedShares : set Int {
  sum i : PlanVestingTx | i.vestedShares
}

fun _grantedShares : set Int {
  sum i : PlanIssuanceTx | i.grantedShares
}
```

Listing 30: Plan security shares

The issuance of new shares into the system is encapsulated by the `issuedShares` and `newShares` functions, as illustrated in Listing 31.

```
fun _issuedShares : set Int {
  sum i : StockIssuanceTx | i.issuedShares
}

fun _newShares : set Int {
  add[_issuedShares, _grantedShares]
}
```

Listing 31: New shares

Shares that have been cancelled are represented by the `cancelledShares` function, as delineated in Listing 32.

```
fun _totallyCancelledShares : set Int {
  sum i : StockTotalCancellationTx | i.inputSecurity.shares
}

fun _partiallyCancelledShares : set Int {
  sum i : StockPartialCancellationTx | i.cancelledShares
}

fun _cancelledShares : set Int {
  add[_totallyCancelledShares, _partiallyCancelledShares]
}
```

Listing 32: Cancelled shares

Lastly, the measure of all shares in circulation, also known as outstanding shares, is encapsulated by the `outstandingShares` function, as illustrated in Listing 33.

```
fun _currentStockSecurities : set StockSecurity {
  StockSecurity - _expiredStockSecurities
}

fun _expiredStockSecurities : set StockSecurity {
  terminalTx.StockTx
}

fun _currentShares : set Int {
  sum s : _currentStockSecurities | s.shares
}
```

Listing 33: Outstanding shares

We can now delineate the accounting properties subject to verification. These properties are derived from domain equations:

$$0 \leq \texttt{exercisedShares} \leq \texttt{vestedShares} \leq \texttt{grantedShares} \tag{4.1}$$

$$0 \leq \texttt{cancelledShares} \leq \texttt{issuedShares} + \texttt{exercisedShares} \tag{4.2}$$

$$\texttt{currentShares} \leq \texttt{issuedShares} + \texttt{exercisedShares} - \texttt{cancelledShares} \tag{4.3}$$

The checks are implemented as in Listings 34, 35 and 36.

```
check {
    lte[0, _exercisedShares]
    lte[_exercisedShares, _vestedShares]
    lte[_vestedShares, _grantedShares]
}
```

Listing 34: Check plan shares

```
check {
    lte[0, _cancelledShares]
    lte[_cancelledShares, add[_issuedShares, _exercisedShares]]
}
```

Listing 35: Check cancelled shares

```
check {
    lte[_currentShares, sub[add[_issuedShares, _exercisedShares], _cancelledShares]]
}
```

Listing 36: Check outstanding shares

4.4.2 Count checks

Count-based verification is instrumental in precluding a large swath of invalid instances, bearing resemblance to techniques such as reference counting. These checks are computationally efficient and serve as valuable tools for model validation.

The count-based checks are formulated to evaluate the following equations:

$$\#terminalTx = \#inputSecurity \tag{4.4}$$

$$\#initialTx = \#outputSecurity + \#balanceSecurity \tag{4.5}$$

$$\#inputSecurity + \#balanceSecurity + \#outputSecurity = \#terminalTx + \#initialTx \tag{4.6}$$

$$\#parentTx \leq \#Tx \tag{4.7}$$

$$\#initialTx \leq \#Security \tag{4.8}$$

The source code for the checks is shown in Listing 37.

```
chkTerminalTxCount : check {
  eq[#inputSecurity, #terminalTx]
}

chkInitialTxCount : check {
  eq[
    add[#outputSecurity, #balanceSecurity],
    #initialTx
  ]
}

chkTxSecurityCountEq : check {
  eq[
    add[#inputSecurity, add[#balanceSecurity, #outputSecurity]],
    add[#terminalTx, #initialTx]]
}

chkParentTxLeqTxCount : check {
  lte[#parentTx, #Tx]
}

chkInitialTxCountLeqSecurityCount : check {
  lte[#initialTx, #Security]
}
```

Listing 37: Count checks

4.4.3 Structure checks

Structural verifications serve a pivotal role, confirming that the graph of securities and transactions constitutes a Directed Acyclic Graph (DAG), that no securities exist without corresponding transactions, and that all securities are anchored in an issuance transaction.

The simplicity of the resulting graph is intrinsically linked to its traceability, underlining the efficacy of the model in ensuring transparent and accountable transactions.

No cycles in graph

Alloy provides a dedicated graph module comprising predicates and functions tailored for graph analysis. Among these is the `dag` predicate, which ascertains whether a graph derived from a binary relation constitutes a Directed Acyclic Graph (DAG). The model can be invoked as described in Listing 38. Utilizing the `dag` predicate allows for a succinct articulation and verification of the graph's acyclic nature, as demonstrated in Listing 39.

```
open util/graph[Security]
open util/graph[Tx]
```

Listing 38: Opening the util/graph module for securities and transactions

```
check { dag[parentTx] } for 5
check { dag[parentSec] } for 5
```

Listing 39: Check that the `parent` relationships induce an acyclical graph.

Graph roots

We validate that the graph engendered by the `parentSec` and `parentTx` relationships forms a Directed Acyclic Graph (DAG). According to domain-specific principles, every share can be traced back to an original issuance. Therefore, each security invariably contains an issuance in its historical lineage. This invariant holds across multiple securities and extends to the base case involving a single security within a trace.

By transitive implication, if each security has an issuance in its lineage, the initial security must be valid. Subsequent securities in the sequence must either be issuances themselves or descendants of the preceding security, paralleling the logic of natural induction.

```
check {
    {
        all s : StockSecurity | some i : StockIssuanceTx + PlanIssuanceTx | i in
↪   s.*parentSec.initialTx
    }
    {
        all p : PlanSecurity | some i : PlanIssuanceTx | i in p.*parentSec.initialTx
    }
} for 5
```

Listing 40: Checking that all securities contain an issuance in their lineage

No overlap in portfolios

Within the given domain, it is axiomatic that each security can have a singular owner. Formally, this necessitates that all portfolios—defined as the aggregated securities belonging to an individual stakeholder—must be disjoint.

```
check {
    all disj s1, s2 : Stakeholder
    | no s1.portfolio & s2.portfolio
} for 5
```

Listing 41: No overlap in portfolios

4.5 Discussion and contributions

Systems managing security transactions, including capitalization table systems, necessitate both traceability to origin and adherence to accounting principles. Our formalized model, validated through Alloy, successfully fulfills these prerequisites. By articulating the properties in a precise manner, we facilitate effective communication with both domain experts and software engineers. Furthermore, employing Alloy's verification capabilities significantly enhances quality assurance levels, which were not only lacking in the Open Cap Table Coalition's original model but also inherently inexpressible using JSON Schema.

Part III

Reflections on the Model Usage and Conclusions

Chapter 5

Towards Legal Contracts and Code Normativity

The inherent difficulty of encoding legal norms into software stems from the ambiguity and complexity of legal language, coupled with the potential for context-dependent interpretation. Traditional contracts, although comprehensive, often rely on a broader legal framework to fill in gaps and resolve ambiguities, features that are challenging to replicate in code.

Our work engages with the field of code normativity, which aims to translate legal rules into executable code. We address similar challenges of interpretation and execution in the context of capitalization tables. By modeling essential contract elements in Alloy, we contribute to both cap table management and the broader issue of legal-to-code translation. Our approach serves as a practical case study for how formal methods can bridge the legal and technical realms, streamlining transactions and compliance.

5.1 Our Work in the Context of Code Normativity

In our model, we have gathered the fundamental concepts of contracts that are relevant to capitalization tables into a formal model in Alloy. By doing so, we not only serve the immediate purpose of better cap table management but also contribute to the more generalized problem of translating legal norms into executable software. Our model, therefore, serves as a case study of how formal methods can be applied to encode complex legal frameworks in a machine-readable format, thereby facilitating automatic execution and validation.

Role of Contracts in Investment Activities

Investment instruments specific to start-ups, such as venture rounds and stock option plans, are formalized through contracts written in natural language. The execution of these contracts, subsequent to being ratified by all involved parties, activates various provisions and stipulations, including the allocation and pricing of instruments like shares or options.

Incompleteness and Reliance on Legal Frameworks

Traditional contracts, although exhaustive in defining roles, obligations, and privileges, are intrinsically incomplete. They function under the overarching structure of the law, which serves as the foundational contract dictating the rules of interaction. Very often, contracts may include direct or implied references to legal clauses or other contractual documents to fill in the gaps in their stipulations.

The Challenge for Software Engineers: From Prose to Code

For software engineers responsible for developing cap table management systems, the narrative-driven contracts present an obstacle. Translating these contracts into functional data models and algorithms is neither trivial nor direct, given the lack of formal language and machine-readability in the contracts. The same legal construct might be represented in myriad ways across different contracts, adding to the complexity of translation.

Simplifying Contracts Through Our Model

Our computational model distills contracts to their essential elements, focusing on what is pertinent to cap tables:

- Stock issuances and both partial and total transfers correlate with share purchase agreements.
- Vesting systems and plan transactions can be represented by stock option plan grant awards.
- Option exercises correspond to stock option exercise agreements.

We do not assert that traditional contracts are no longer relevant; nonetheless, the integration of formal specifications in conjunction with traditional contracts would undoubtedly enhance the efficiency of the execution process.

5.2 Code Normativity and Community Efforts

Two emerging trends within the community underscore the concept of code normativity in the context of financial transactions.

5.2.1 Smart Contracts

The notion of smart contracts, conceptualized by Nick Szabo in 1994, encapsulates computer protocols that automate, verify, or even eliminate the need for conventional contracts. The advent of Bitcoin propelled the development and adoption of smart contracts, with Ethereum pioneering the field by offering a general-purpose blockchain featuring an embedded programming language. This enabled the creation of smart contracts in a high-level language called Solidity, which is compiled to Ethereum Virtual Machine (EVM) bytecode.

However, Solidity was initially susceptible to security vulnerabilities, notably reentrancy attacks, which were exploited in the notorious DAO hack. The incident was not only financially devastating—resulting in the loss of 3.6 million Ether—but also socially divisive, as it led to an Ethereum hard fork.

We can certainly find plenty research on verification of smart contracts. KEVM represents a formal semantics for the Ethereum Virtual Machine (EVM) and is grounded in the K framework, a tool for modeling programming languages [5]. On the other hand, Lolisa [6] provides formal syntax and semantics for a specific subset of the Solidity programming language, a key component of Ethereum smart contracts. In the realm of smart contract development, there's also notable work on safer contract programming, exemplified by Safer Smart Contracts through Type-Driven Development [25]. Furthermore, Scilla, based on Coq, serves as a language designed explicitly for smart contracts and emphasizes safety in its programming model [7].

Indeed, these contributions significantly enhance the formal foundations of blockchain and smart contract technologies by focusing on the program level to prevent bugs and errors. However, it's important to note that their primary emphasis is on the technical aspects of smart contract development, ensuring correctness and safety within the programming language and execution environment.

Addressing domain-level concerns, such as the interpretation and management of financial and contractual obligations, often remains a separate challenge that requires a combination of technical and legal expertise.

Our model not only captures individual contracts but also models the global dynamics of the system, which individual smart contracts often overlook. It integrates domain-level business rules.

5.2.2 Tax and Accounting: The Catala Approach

The second trend focuses on leveraging formal languages for tax and accounting, exemplified by Catala, a domain-specific language[4]. Catala facilitates the definition of rules in a manner that is both interpretable by humans and machines. A notable feature of Catala is its ability to identify contradictions within the rules, addressing a common issue in the domain of tax and accounting.

Similarly, our use of Alloy serves analytical purposes, allowing us to examine the collective consistency of various contracts and rules. This analytical power aligns our work closely with the spirit of the Catala approach.

By harnessing formal languages like Alloy, we aim to bridge the gap between natural language contracts and computational representations, thereby contributing to the broader discourse on code normativity in financial systems.

5.3 An example rendition of our model as prose contracts

In this section, we illuminate the process of outlining a stock option plan utilizing an instance of an Alloy model, showcased in Listing 42.

```
pred sop1 {

  one p : PlanIssuanceTx {

    p.strikePrice = 10

    // Vesting Schedule
    one c : p.conditions | c.trigger in AfterDate && c.trigger.afterDate = Date1 and
↪   eq[c.shars, 20]
    one c : p.conditions | c.trigger in AfterDate && c.trigger.afterDate = Date2 and
↪   eq[c.shars, 20]
    one c : p.conditions | c.trigger in AfterDate && c.trigger.afterDate = Date3 and
↪   eq[c.shars, 20]
    one c : p.conditions | c.trigger in AfterDate && c.trigger.afterDate = Date4 and
↪   eq[c.shars, 20]

    // A bonus of 10 shares is issued if the share price is greater than £20 by Date2
↪   (Event1)
    one c : p.conditions | c.trigger in AfterEvent && c.trigger.afterEvent = Event1 and
↪   eq[c.shars, 10]

    // A bonus of 10 shares is issued if the share price is greater than £30 by Date2
↪   (Event1 again)
    one c : p.conditions | c.trigger in AfterEvent && c.trigger.afterEvent = Event1 and
↪   eq[c.shars, 10]

    #p.conditions = 5
  }

}
```

Listing 42: Predicate describing an stock option grant

This model is further converted into a closely analogous prose rendition, offering an illustration of how the logical and structured format of an Alloy model can be translated into the legal language used in contractual agreements.

STOCK OPTION AGREEMENT

This Stock Option Agreement (the "Agreement") is made and entered into as of [Date], by and between [Company Name], a [State of Incorporation] corporation (the "Company"), and [Name of Optionee], an employee of the Company (the "Optionee").

Grant of Option. The Company hereby grants to the Optionee an option (the "Option") to purchase a total of up to 60 shares of common stock, $10 strike price per share, of the Company (the "Shares") on the terms and conditions set forth in this Agreement.

Vesting Schedule. Subject to the terms and conditions set forth herein, the Option will vest in accordance with the following schedule:

a. 20 Shares shall vest on [Date1];

b. An additional 20 Shares shall vest on [Date2];

c. An additional 20 Shares shall vest on [Date3];

d. An additional 20 Shares shall vest on [Date4].

Additional Vesting Conditions.

a. A bonus of 10 additional Shares shall be issued to the Optionee if the closing price of the Company's common stock is greater than $20 on [Date2];

b. An additional bonus of 10 Shares shall be issued to the Optionee if the closing price of the Company's common stock is greater than $30 on [Date2].

While this example is not exhaustively detailed, it demonstrably highlights the feasible potential that tools might aptly employ our model to specify particular instances and proficiently generate legal text, contingent upon their validation. This facilitates the seamless transition from structured Alloy models to comprehensive and legally sound prose contracts, enhancing the efficiency and efficacy of contract creation and management.

Chapter 6

Conclusion

This thesis commenced with an examination of a data-centric domain model, endowed with extensive business logic and domain-specific knowledge, yet constrained by its technological underpinning. It was posited that the Open Cap Table Format (OCF) is inherently limited in its computational capabilities; its constraints are particularly manifest in its inability to be fully specified using JSON Schema. While the OCF excels in data format validation—consonant with its role as a data interchange format—it lacks the mechanism for managing capitalization tables in a semantically rigorous manner.

By employing Alloy, a lightweight formal modeling language, this research has led to the development of a more robust model capable of calculating the state of a capitalization table, validating transaction and security input data, and subjecting the model to automated validation to ensure its correctness. Contrary to the original model, the resultant model facilitates exhaustive tests concerning business logic.

Transitioning from a data specification to a domain specification, this research took the implicit domain knowledge of the original model as its foundation. It then identified and formalized the key abstractions in Alloy, effectively moving from a syntactic to a semantic model that accounts for relationships between different types of entities and expected invariants based on domain knowledge.

The benefits accrued from this transition include:

1. Computational ability to derive the state of a capitalization table given a set of securities and transactions.

2. Preservation of immutable data, akin to the original model, while ensuring a readily computed state.
3. Assured traceability back to the original issuance of shares for any given security.
4. Implementation of accounting constraints to prevent the erroneous creation or elimination of shares.
5. Introduction of count checks for additional validity by eliminating implausible graphs.
6. Explicit, machine-checkable assertions of model correctness.
7. Enhancement of the vesting system to express a broader range of conditions through logical operators.

Consequently, this research not only offers a system specification for capitalization tables but also lays the foundation for formalizing financial transaction contracts. The resulting transaction constraints, when articulated as contract clauses, yield a representation that is more explicit, rigorous, and concise than conventional legal language, focusing particularly on operational facets of contracts.

Moreover, as shown in Chapter 5, our work has implications for the field of code normativity, which are reflected in the future work as opportunities for contract generation from Alloy or other formal models.

6.1 Limitations of our Model

Our model exhibits certain limitations that are essential to acknowledge:

1. **Scope Size Constraints:** The scope size for model checking and model finding is inherently constrained by the current capabilities of modern SAT technology.

2. **Execution Time Challenges:** As our Alloy models increase in complexity and scope size, the execution time also experiences a substantial increase. Consequently, running large Alloy models may become impractical, particularly for tasks such as simulation or validation involving more than a few transactions.

3. **Arithmetic Constraints:** Alloy's arithmetic capabilities are rooted in set-based representations rather than dedicated arithmetic theory solvers. This is due to Alloy's reliance on SAT solving. This constraint imposes limitations on our model, especially when dealing

with real capitalization tables, which often involve substantial financial data and numbers much larger than what Alloy supports.

4. **Over-Specification:** It is plausible that our model is more extensive than necessary, displaying characteristics of over-specification, since during development we found predicates that where implied by other predicates, that is, are already a consequence of the model. In principle, there is an opportunity to refactor the code to promote the reuse of common logic. The presence of such over-specification introduces redundancy into the model, which is generally considered undesirable in formal modeling and specification.

Recognizing these limitations is crucial for effectively leveraging our model in practical applications within the domain of the Open Cap Table Format.

6.2 The New Model: technical issues

A few aspects of the model are worth mentioning, because they solve complicated implementation problems in a simple manner.

Mostly local constraints

Most constraints are stated locally, in the sense that they are stated in terms of the fields of a single transaction. This makes it easier to write validation code because we can check each transaction in isolation. Global constraints are much harder to reason about.

Current state is always kept reified

The current state is always kept reified, avoiding the need to replay the whole history to get the current state. This is a big win, not only performance-wise but also in terms of simplicity: it makes the model much more understandable.

Each security issuance gives rise to a separate graph of transactions

It is clear that each security issuance gives rise to a graph of transactions that is separate. This can be a boundary of transactions in the sense of concurrency since it identifies orthogonal partitions of the state. This allows concurrency, which aids both in performance and in high availability.

It is time to consider what have we gained by using Alloy to model the system.

6.2.1 What have we gained?

The original model had ambiguities in its interpretation and no properties of any kind specified. The state of the system had to be reconstructed by replaying all transactions, for which no unambiguous interpretation is available.

We have gained the following:

1. We have a formal model of the system that is unambiguous and that can be used to reason about the system.
2. Our model is practical from the point of view of implementation.
3. We can ensure that the model behaves correctly by respecting constraints over the bipartite graph of transactions and securities and over accounting restrictions.
4. The visualizations provided by the Alloy Analyzer are very useful to understand the model and to comunicate it to others.
5. The current state of the system is now reified and easy to access.

Both were possible only after:

1. Carefully analyzing the original model and identifying the key concepts and mapping them conceptually as entities and constraints.
2. Establishing what properties define the system as correct.
3. Using model-checking and simulations to refine the model.

All three steps above require using Alloy as a modeling language. The kinds of properties and constraints we require are not expressible in JSON Schema, and we certainly do not enjoy the benefits of model checking and simulation without Alloy.

6.3 Future Work

The research presented herein serves as an initial platform for continued exploration into the application of lightweight formal methods in financial software design. Possible extensions and directions for future work are:

- **Model Extension:** Incorporate additional securities and transaction types.

- **Temporal Modeling:** Utilize the temporal logic capabilities introduced in Alloy version 6 for constructing state-machine-based capitalization table models.
- **Automated Contract Generation:** Investigate the feasibility of auto-generating textual contracts through Alloy-to-text translators, thereby eliminating the laborious manual verification process.
- **Code Generation:** Explore automated code generation using Alloy's Java API. Given that most business logic is succinctly encapsulated as constraints, the target programming language would require minimal features to support the generated code.
- **Alternative Formalisms:** Examine the applicability of other formal methods, such as TLA plus, for specifying similar problems from different perspectives.

Appendix A

Types and enums in the OCF

Below, we provide a overview of the common usage types and enumerations found within the Open Cap Table Format (OCF). These are utilized as attributes for both securities and transactions, aiding in better understanding the format's structure and purpose.

General use types

There are 50 available *types* in the OCF, of which 22 are types for general use, such as address types, currencies, monetary values et cetera. Figure A.1 details the directory structure of the general use types.

```
schema/
└── types/
    ├── Address.schema.json
    ├── CapitalizationDefinition.schema.json
    ├── ContactInfo.schema.json
    ├── CountryCode.schema.json
    ├── CountrySubdivisionCode.schema.json
    ├── CurrencyCode.schema.json
    ├── Date.schema.json
    ├── Email.schema.json
    ├── File.schema.json
    ├── InterestRate.schema.json
    ├── Md5.schema.json
    ├── Monetary.schema.json
    ├── Name.schema.json
    ├── Numeric.schema.json
    ├── Percentage.schema.json
    ├── Phone.schema.json
    ├── Ratio.schema.json
    ├── SecurityExemption.schema.json
    ├── ShareNumberRange.schema.json
    ├── StockParent.schema.json
    ├── TaxID.schema.json
    └── TerminationWindow.schema.json
```

Figure A.1: Types in the OCF for general use

Types supporting transactions and securities

Beyond the general use types for common concepts such as addresses and monetary values, there are types defining the entities and **transactions** that participate in the lifetime of a **capitalization table**. Figure A.2 shows the directory structure of the types supporting transactions and securities.

```
schema/
└─ types/
    ├─ conversion_mechanisms/
    ├─ CustomConversionMechanism.schema.json
    ├─ SAFEConversionMechanism.schema.json
    ├─ NoteConversionMechanism.schema.json
    ├─ PercentCapitalizationConversionMechanism.schema.json
    ├─ FixedAmountConversionMechanism.schema.json
    └─ RatioConversionMechanism.schema.json
```

Figure A.2: Conversion mechanisms

Figure A.3 shows the directory structure of the conversion rights.

```
schema/
└─ types/
    └─ conversion_rights/
        ├─ WarrantConversionRight.schema.json
        ├─ ConvertibleConversionRight.schema.json
        └─ StockClassConversionRight.schema.json
```

Figure A.3: Conversion rights

Figure A.4 shows the directory structure of the vesting types.

```
schema/
└─ types/
    └─ vesting/
        ├─ VestingPeriodInMonths.schema.json
        ├─ VestingConditionPortion.schema.json
        ├─ VestingEventTrigger.schema.json
        ├─ VestingStartTrigger.schema.json
        ├─ VestingScheduleRelativeTrigger.schema.json
        ├─ VestingCondition.schema.json
        ├─ VestingPeriodInDays.schema.json
        └─ VestingScheduleAbsoluteTrigger.schema.json
```

Figure A.4: Vesting events and conditions, periods and portions

Figure A.5 shows the directory structure of the conversion triggers.

```
schema/
└── types/
    └── conversion_triggers/
        ├── ElectiveConversionInDateRangeTrigger.schema.json
        ├── AutomaticConversionOnDateTrigger.schema.json
        ├── AutomaticConversionOnConditionTrigger.schema.json
        ├── UnspecifiedConversionTrigger.schema.json
        ├── ElectiveConversionAtWillTrigger.schema.json
        └── ElectiveConversionOnConditionTrigger.schema.json
```

Figure A.5: Conversion triggers, divided in elective and automatic

Enums

Enums are constant values. Figure A.6 shows the directory structure of the enums.

```
schema/
└── enums/
    ├── AccrualPeriodType.schema.json
    ├── AddressType.schema.json
    ├── AllocationType.schema.json
    ├── CompensationType.schema.json
    ├── CompoundingType.schema.json
    ├── ConversionMechanismType.schema.json
    ├── ConversionRightType.schema.json
    ├── ConversionTimingType.schema.json
    ├── ConversionTriggerType.schema.json
    ├── ConvertibleType.schema.json
    ├── DayCountType.schema.json
    ├── EmailType.schema.json
    ├── FileType.schema.json
    ├── InterestPayoutType.schema.json
    ├── ObjectType.schema.json
    ├── OCFVersionType.schema.json
    ├── OptionType.schema.json
    ├── ParentSecurityType.schema.json
    ├── PeriodType.schema.json
    ├── PhoneType.schema.json
    ├── RoundingType.schema.json
    ├── StakeholderRelationshipType.schema.json
    ├── StakeholderType.schema.json
    ├── StockClassType.schema.json
    ├── TerminationWindowType.schema.json
    ├── ValuationType.schema.json
    ├── VestingDayOfMonth.schema.json
    └── VestingTriggerType.schema.json
```

Figure A.6: Enums in the OCF

They at principle look simple or trivial; actually, they consolidate important domain expertise. The `Allocation` type, for example, defines that shares in a vesting term can be allocated with different rounding methods, and whether to allocate shares unevenly divisible by the number of **vesting periods** by front-loading (i.e., allocating the remainder to the first vesting period) or back-loading (i.e., allocating the remainder to the last **vesting periods**). From the author's experience, this sort of low-level business rule detail takes a large part of developing systems for financial domains. There are specific enums for periodicity classes, different methods of calculating the day of vesting in the month (i.e., business days versus calendar days), and different methods of calculating interest accruals, rounding methods et cetera. These details are often overlooked during the design of a system for managing financial information, so we value that they are explicitly defined in the OCF.

Bibliography

[1] D. Caines, *Global venture capital investment shatters records — kpmg.com*, `https://kpmg.com/xx/en/home/media/press-releases/2022/01/global-venture-capital-annual-investment-shatters-records-following-another-healthy-quarter.html`, [Accessed 10-Jun-2023].

[2] *Open Cap Table Coalition (OCT) — opencaptablecoalition.com*, `https://www.opencaptablecoalition.com/`, [Accessed 27-May-2023].

[3] S. P. Jones, J.-M. Eber, and J. Seward, "Composing contracts," *ACM SIGPLAN Notices*, vol. 35, no. 9, pp. 280–292, Sep. 2000. DOI: `10.1145/357766.351267`. [Online]. Available: `https://doi.org/10.1145/357766.351267`.

[4] D. Merigoux, N. Chataing, and J. Protzenko, "Catala: A programming language for the law," *Proceedings of the ACM on Programming Languages*, vol. 5, no. ICFP, pp. 1–29, 2021. DOI: `10.1145/3473582`. [Online]. Available: `https://doi.org/10.1145%2F3473582`.

[5] E. Hildenbrandt, M. Saxena, N. Rodrigues, *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 204–217. DOI: `10.1109/CSF.2018.00022`.

[6] Z. Yang and H. Lei, "Lolisa: Formal syntax and semantics for a subset of the solidity programming language," *ArXiv*, vol. abs/1803.09885, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:4398330`.

[7] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, "Safer smart contract programming with scilla," vol. 3, no. OOPSLA, 2019. DOI: `10.1145/3360611`. [Online]. Available: `https://doi.org/10.1145/3360611`.

[8] A Metrick, *Venture capital and the finance of innovation, third edition*, 3rd ed. Nashville, TN: John Wiley & Sons, Apr. 2021.

[9] *JSON Schema — json-schema.org*, `https://json-schema.org/`, [Accessed 10-Jun-2023].

[10] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," IETF, RFC 8259, Dec. 2017. [Online]. Available: `http://tools.ietf.org/rfc/rfc8259.txt`.

[11] *Implementations — json-schema.org*, `https://json-schema.org/implementations.html`, [Accessed 27-May-2023].

[12] O. C. T. Coalition, *Vesting System - Open Cap Table Format Documentation*, `https://open-cap-table-coalition.github.io/Open-Cap-Format-OCF/explainers/VestingTerms/`, [Accessed 08-Jun-2023].

[13] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Software Engineering Methodology*, vol. 11, no. 2, 256–290, 2002, ISSN: 1049-331X. DOI: `10.1145/505145.505149`. [Online]. Available: `https://doi.org/10.1145/505145.505149`.

[14] D. Jackson, *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2012, ISBN: 0262017156.

[15] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy∗: A general-purpose higher-order relational constraint solver," *Formal Methods in System Design*, vol. 55, no. 1, pp. 1–32, Jan. 2017. DOI: `10.1007/s10703-016-0267-2`. [Online]. Available: `https://doi.org/10.1007/s10703-016-0267-2`.

[16] I. Milicevic Aleksandar Erfrati and D. Jackson, "Arby—an embedding of alloy in ruby," in *Abstract State Machines, Alloy, B, VDM, and Z*, ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014.

[17] *Alloy: Case studies – http://alloytools.org/citations/case-studies.html*, `http://alloytools.org/citations/case-studies.html`.

[18] A. Spiridonov and S. Khurshid, "Pythia : Automatic generation of counterexamples for acl 2 using alloy," 2007. [Online]. Available: `https://api.semanticscholar.org/CorpusID:18896003`.

[19] J. C. Blanchette and T. Nipkow, "Nitpick: A counterexample generator for higher-order logic based on a relational model finder," in *Interactive Theorem Proving*, M. Kaufmann and L. C. Paulson, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 131–146, ISBN: 978-3-642-14052-5.

[20] R. Carpio and I. Alsmadi, "Websites security policies implementation using alloy analyzer," *SSRN Electronic Journal*, 2021. DOI: `10.2139/ssrn.3939856`. [Online]. Available: `https://doi.org/10.2139/ssrn.3939856`.

[21] C. Chen, P. Grisham, S. Khurshid, and D. Perry, "Design and validation of a general security model with the alloy analyzer," Jan. 2006.

[22] R. Podorozhny, S. Khurshid, D. Perry, and X. Zhang, "Verification of multi-agent negotiations using the alloy analyzer," in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 501–517. DOI: 10.1007/978-3-540-73210-5_26. [Online]. Available: https://doi.org/10.1007/978-3-540-73210-5_26.

[23] J. Johnson and I. Alsmadi, "Formal modeling of banking policies using alloy analyzer," *SSRN Electronic Journal*, 2021. DOI: 10.2139/ssrn.3939880. [Online]. Available: https://doi.org/10.2139/ssrn.3939880.

[24] E. Kang, S. Perez De Rosso, and D. Jackson, *500 lines or less - the same-origin policy*, https://aosabook.org/en/500L/the-same-origin-policy.html, (Accessed on 05/23/2023).

[25] J. Pettersson and R. Edström, "Safer smart contracts through type-driven development," 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:62247546.