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

Formal Methods for Validating Capitalization Tables and Transactions in the Venture Capital Market: Enhancing the Open Cap Table Format

Rodrigo Ehrich 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 dissertation presents a model of the domain of startup financing, in particular a model for the transactions that compose to form capitalization tables. We present a recently released data interchange format (Open Cap Table format, or OCF) proposed by an industry coalition (Open Cap Table Coalition, or OCTC), and proceed to translate it from the original JSON Schema format to Alloy, a formal but lightweight specification language. We find that the Alloy model much more expressive and useful than the original JSON Schema, since it is more precise, captures subtle invariants in the problem domain, and can be explored and checked with the Alloy Analyzer, which features model search and bounded model checking. Our research contributes to the field of formal methods as it shows how one can proceed from a semi-formal specification to a formal one, evaluates what the results, and contributes to the organization behind the data interchange format by providing a formal specification of their format.

Contents

List of Figures

List of Tables

List of Listings

Todo list

Chapter 1

Introduction

1.1 Background and Motivation

The venture capital market revolves around acquiring financial resources (capital for investment). This capital is invested in startup companies that the investor hopes will be sold for a large profit in the feature. It is an investment modality with high risk, but high returns.

Startup companies obtain resources in stages that correspond to the progress made in achieving specific milestones. A funding round involves not only the startup itself and the investors, but an ecossystem of law firms, financial advisors and auditing firms.

The transactions that take place in a funding round are complex, and the resulting data are difficult to maintain and validate. At each point in time in general, and immediately before and after a transaction, there is a relation between investors in the company and the number of shares they own (which might have different classes and rights). For any given company, the set of such relations at a given moment in time is commonly called a capitalization table.

Capitalization tables are typically kept in spreadsheets, and are exchanged frequently during negotiations. Given the limitations of spreadsheets, it is easy to make mistakes when updating them. This introduces risks of legal and financial liability, and can lead to disputes between investors and companies.

Giving the need for mitigating this risks, an ecossystem of software-as-a-service solutions has emerged to help companies manage their capitalization tables. However, these solutions are

not interoperable, and there is no standard for exchanging capitalization table data.

The Open Cap Table Coalition[?], a recently formed alliance of key venture capital firms, law firms and software solution providers released what is a candidate for an industry standard for exchanging capitalization table data, known as the Open Cap Table Format[?] (which we will refer to as OCF). It is an open-source standard described semi-formally as schemas and validation rules for JavaScript Object Notation (JSON) documents. JSON is a popular data interchange format, and is used in many web applications. Specifically, the format follows the JSON Schema standard.

The OCF is a step forward in the direction of making capitalization tables reliable and easy to validate. However, it is not a formal specification, and it is inherently limited by the semantics of JSON Schema. A number of validations that emerge as constraints on the domain model, as we shall see, can not be captured by the semantics of JSON Schema, limiting the application of the format as a base for automated verification of the transactions that build a capitalization table.

In this thesis, we propose an alternative model for validating capitalization tables and transactions using the Alloy[?] language. Alloy is part of what is called lightweight formal methods, which are formal methods that are easier to use and require less training to apply than traditional formal methods (compared to other languages such as Z, B, TLA+)[?]. It still allows sophisticated analysis of the constructed model. We show that the Alloy model is more expressive and useful than the original JSON Schema, since it is more precise, captures subtle invariants in the problem domain, and can be explored and checked with the Alloy Analyzer, which features model search and bounded model checking.

Since in the research questions I have already given more specific results what we might expect, I can be more specific above

1.2 Research Questions and Contributions

In this thesis, we explore the possibilities of using a language designed for formally expressing domains and concepts versus using a language designed for validating data structures. We use Alloy to model capitalization tables and transactions, and compare the results with the original JSON Schema model.

Given this setting, we aim to answer the following questions:

1. Can Alloy be be used to assure that the model holds transactional integrity, that is, can it be used do defined the financial operations and constraints and ensure that they are consistent?

2. Can the higher expressivity of Alloy be used to enhance the model's own expressivity, such as support for finer-grained business rules?

3. What concepts can be expressed in Alloy that cannot be expressed on JSON Schema but are key for constraining the model to be more consistent with reality?

4. Can we evaluate the results of the Alloy model in a way that is useful for the domain experts, and compare them to other forms of assuring correctness and realibity?

1.2.1 Contributions

In investigating the research questions, we make the following contributions:

1. We avoid double-issuances and double-spending of securities, as well as enforcing the correct order of transactions and the DAG-ness of the transaction graph

2. We enhance the expressivity of business rules by explicitly modeling the AST that those rules would have if they where depicted as a domain-specific language

3. We show how basic accounting equations can be checked for correctness over the resulting model, and how important queries can be specified and checked, such as querying for the complete history for a security, and show that those queries are correct

4. We evaluate the output of Alloy Analyzer, that checks the models using a SAT solver, and includes parameters related to the size of the underlying bit-blasted model that can be compared to the coverage of regular unit tests

Those are all properties that ultimately lie in legal contracts. Thus, our thesis helps bridge the gap between the ambiguities inherent to both natural language and the law, an important step into the development of accurate models of the domain, which is critical if we consider that having accurate and correct conceptual models is of paramount importante now to check the correctness of large language models and generative AI in general.

1.3 Thesis Organization

The remainder of this thesis is organized as follows.

In the appendix, the reader will find full code listings for the Alloy model, and the JSON Schema for the Open Cap Table Format, the grammar and description of JSON Schema and a quick reference for Alloy.

In item 4, we have good references to quote from Ammans and Offutt, and we can also quote from the

Chapter 2

Background

This thesis is meant for the general computer science public, and so we will provide some background on the tools and technologies used in this thesis. We will also provide some background on capitalization tables and ecossystem of software solutions, including the OpenCapTable Format.

2.1 Overview of Alloy and JSON Schema

We use two languages in this thesis: Alloy and JSON Schema.

Alloy is a language for describing models in first-order logic and the Alloy Analyzer is a tool for analyzing such models. JSON Schema is a data validation format for JSON documents, with a JSON-based syntax.

2.1.1 Overview of Alloy

Alloy[?] is a formal specification language and analysis tool developed by Daniel Jackson and his colleagues at the Massachusetts Institute of Technology (MIT). Alloy allows users to define abstract models of complex systems and to automatically check their properties using a bounded model checking algorithm. Alloy's syntax is based on first-order logic and set theory, and includes constructs for defining relations, functions, and constraints.

An Alloy model consists of the following parts:

- Signature declarations, labeled by a `sig` keyword, introduce a set of *atoms* and a set of *fields* relating atoms. A signature is similar but more general than a SQL table, for example.

- Constraint declarations, introduced by the keywords `fact`, `fun`, and `predicate`, that limit the possible instances of the model. Facts are assumed to always hold. Functions are reusable, named expressions, that return a relation. Predicates are functions that must evaluate to a boolean.

- Assertion declarations, introduced by the keyword `assert`, that establish constraints that are expected to hold as consequences of the facts. Assertions can be checked by the Alloy Analyzer using the `check` command.

Using the `run` and `check` commands, the user can ask the Alloy Analyzer to find examples and counter examples to expressions (usually built up from predicates) in a given *scope*. The scope determines how many atoms can be created in the model (by signature). A larger scope is populated by more instances, but takes longer to check.

Advantages of Alloy

One of the key benefits of Alloy is its ability to quickly generate and evaluate different scenarios and configurations, allowing designers to iteratively refine their models and identify potential issues.

Alloys' value proposition is to take the ideal of precise and expressive notation based on a tiny core of simple and robust concepts, but it replace basic analysis based on theorem proving with a fully automatic analysis with immediate feedback.[?]

Applications of Alloy

Alloy has been used in a wide range of applications in computer science, including software engineering, database design, security analysis[?][?], multi-agent negotiations [?]. It has also been applied to modeling beyond computer science such as a model for central bank policy[?]. A model of the same-origin-policy used in web browsers can be found in the 500 Lines or Less open-source book[?]. Alloy$*$ is a higher-order extension of Alloy that can be used for program synthesis, since it supports higher-order quantification[?]. αRby is an embedding of Alloy in Ruby[?].

2.1.2 Overview of JSON Schema

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

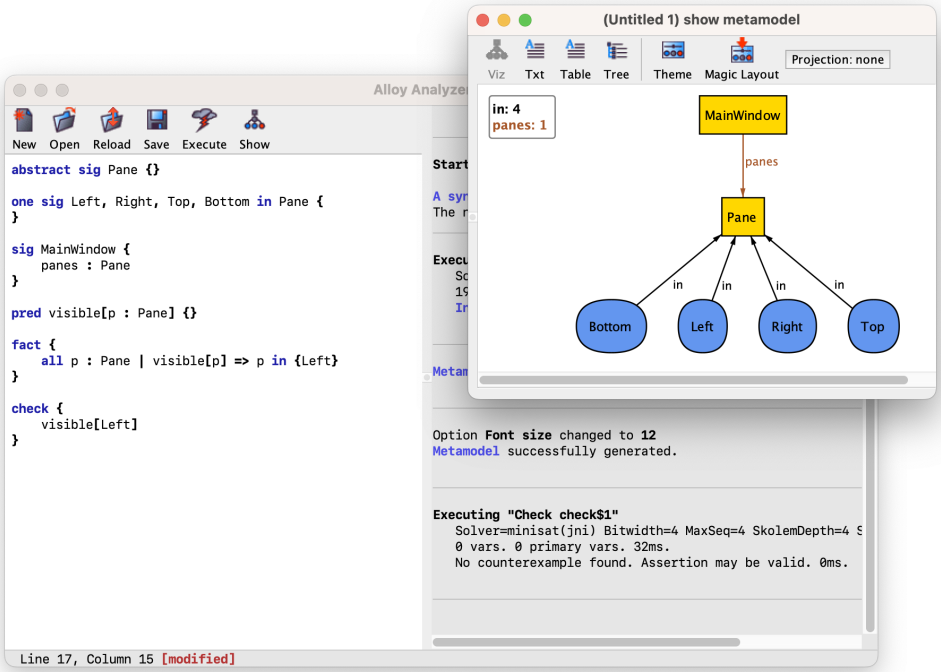


Figure 2.1: Alloy Analyzer

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 proves to be 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.

See table 2.1 for a list of validation keywords.

Validation Keyword	Description
type	Specifies the data type (e.g., string, number, object, array, boolean, null)
enum	Specifies a predefined list of acceptable values
const	Specifies a constant value that the data must match
multipleOf	Specifies that a numeric instance is divisible by this keyword's value
maximum	Specifies the maximum numeric value
exclusiveMaximum	Specifies a numeric instance to be strictly less than this keyword's value
minimum	Specifies the minimum numeric value
exclusiveMinimum	Specifies a numeric instance to be strictly greater than this keyword's value
maxLength	Specifies the maximum length of a string
minLength	Specifies the minimum length of a string
pattern	Specifies a regular expression that a string must match
items	Specifies constraints for array items
maxItems	Specifies the maximum number of items in an array
minItems	Specifies the minimum number of items in an array
uniqueItems	Specifies that all items in an array must be unique
properties	Specifies constraints for object properties
required	Specifies required properties in an object
additionalProperties	Specifies whether additional properties are allowed

Table 2.1: Available Validations in JSON Schema

Tooling for JSON Schema

JSON Schema is supported by a large number of tools[?]: 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.

There is also a noteworthy repository of JSON Schemas for hundreds of APIs, called JSON Schema Store[?].

2.1.3 Mapping from JSON Schema to Alloy

Here we describe our general approach for translating JSON Schema into Alloy. Basically, we follow the steps:

Our scheme implements a depth-first search of the original schema, and when completed, the Alloy model should be a roughly direct translation of the original schema. Once this is done,

1. Translate the *schemas* into *abstract signatures* in Alloy, devoid of any fields.
2. Iterate over each *abstract signature* from step 1. For each, instantiate it into a *concrete signature* and incorporate the corresponding fields as specified in the JSON schema.
3. Examine each field from step 2. Consequently, introduce the pertinent signatures into our *stack* and return to step 2.
4. Formulate *assertions* which are anticipated to be upheld given the domain currently being modeled.
5. Employ the *Alloy Analyzer* to assess whether the assertions from step 4 are satisfied. If not, progress to the subsequent step.
6. Extend the Alloy model with a fresh *constraint* if the assertions from step 4 were not satisfied. Repeat steps from 2 to 6 until all assertions hold true.

Figure 2.2: Process of creating an Alloy model from schemas

we can start improving the model using the more expressive semantics of Alloy, always checking if the assertions hold.

2.2 What are Capitalization Tables?

Formally, a capitalization table can be defined as a mapping of the set of stakeholders in a company to the number of shares they own in each asset class in the company. In this thesis we consider common stocks and stock options as asset classes. Mathematicall, assuming S is the set of stakeholders, A is the set of asset classes, a capitalization table at date t is a function $C_t : S \times A \to \mathbb{N}$.

2.2.1 Overview of Capitalization Tables

A capitalization table is a snapshot of the relationship of stakeholders in a company and the number of shares they own in each asset class in the company. In this thesis we consider common stocks and stock options as asset classes.

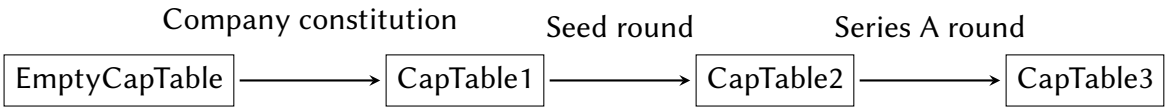
Investor	Asset Class	Shares	Cost (USD)	% of Company
Founders	Common Stock	10,000	-	71.43%
Seed Investors	Preferred Stock	2,000	2,000,000	14.29%
Option Pool	Common Stock	2,000	-	14.29%
Total	-	14,000	2,000,000	100.00%

Table 2.2: Example of a capitalization table

Here, the set of stakeholders S is $\{\text{Founders}, \text{Seed Investors}, \text{Option Pool}\}$, and the set of asset classes A is $\{\text{Common Stock}, \text{Preferred Stock}\}$. The capitalization table is a function $C : S \times A \to \mathbb{N}$, where $C(\text{Founders}, \text{Common Stock}) = 10,000$, $C(\text{Seed Investors}, \text{Preferred Stock}) = 2,000$, and so on.

Changes in a capitalization table are effects from transactions (both in the business sense,

but also in the computing sense): issuance of shares to new investors, transfers of shares between investors, and formation of pools for and exercise of stock options by employees.



Why are Capitalization Tables Important?

Cap tables are important for several reasons. First, they provide a clear and accurate picture of the ownership structure of the company, which is crucial for decision-making and financial reporting. Second, they help to ensure that equity is distributed fairly and transparently among shareholders. Finally, they can be used to calculate the potential payouts and returns for different scenarios, such as a sale of the company or an initial public offering (IPO). A venture capital firm may use a cap table to track the ownership and value of its portfolio companies. The cap table can help the firm to make informed decisions about future investments, and can help to identify potential exit opportunities.

Critically, the cap table can help to ensure that the ownership and value of the company is distributed fairly among investors and founders, and can help to identify potential issues or conflicts.

How are Capitalization Tables Used?

Here are the typical operations in startup financing. All have effects on the final capitalization table.

Table 2.3: Common startup financing operations

Operation	**Description**
Equity financing	Money from investors is converted into shares at a given price-per-share
Convertible note financing	Money from investors is lent to the company as debt, but with a provision to convert into shares in the future
Employee stock options	Employees are given the right to purchase shares at a given price-per-share in the future

2.2.2 Examples

Seed round with convertible debt

Here a new startup has only founders in its capitalization table. The founders have 10,000 shares. The company raises $1,0M in a convertible note financing. The convertible note has a $20M valuation cap. The convertible note converts into shares at the next equity financing round. Thus, shares can be given as actually issued shares and fully diluted shares, which are estimates based on corresponding schedule stock issuances.

Shareholder	Shares	Percentage
Founders	10,000	100%
Total	10,000	100%
Common Stock	10,000	100%

Shareholder	Shares (Actual)	Shares (Fully Diluted)	Percentage
Founders	10,000	10,000	95%
Investor I (Debt)	0	526.32	5%
Total	10,000	10,526.32	100%

Problems in managing capitalization tables

Cap tables may change frequently, reflecting changes in ownership, investment, and valuation over time. They are complex to manage and validate, because they are only a snapshot, a state. A cap table can only be verified in trivial ways sums and percentages not adding up.

But the financial operations and transactions that evolve a capitalization table must comply with a series of legal and business rules that are ultimately defined in business contracts, and software to do so must take into account the following:

1. Avoid double spending and ensuring the overall life of a security is correct (it must be created once and spent only if ever created, and it can undergo non-terminal transactions)

2. No cycles should be possible in the life cycle of any security (a security can not be both the input and output of a single transaction or a set of transactions such as a partial cancellation that creates a residual security)

3. Support very complex business rules that are defined in business contracts, and that are not trivial even to specify

4. Respect basic accounting equations such as that the number of shares in circulation must

never exceed the number of shares created

Which are reflected in our research questions.

2.3 Ecossystem and The Open Cap Table Format

In the last decade (2010-2020), a number of software-as-a-service solutions have emerged in the capitalization table management and related spaces in various geographies. These solutions are not interoperable, and there is no standard for exchanging capitalization table data.

The Open Cap Table Format (OCF) is a data interchange format for capitalization tables. It is a JSON Schema-based format that is designed to be easy to use and understand. The OCF is intended to be used by startups, investors, and other stakeholders in the venture capital market.

In the next chapter, we will give a detailed description of the OCF, and in the following chapters we will provide a translation of the most important parts of the OCF into an Alloy specification.

Company	**Founded**	**Geography**
Shareworks[?]	1999	USA
Carta[?]	2012	USA
Long-term Stock Exchange[?]	2015	USA
Raise[?]	2018	Africa
Distu[?]	2021	Brazil

Chapter 3

Open Cap Table Format

The Open Cap Table format is composed of 143 JSON Schema documents, which are organized in 8 files. The files are organized in a directory structure, which is shown below. Thus the files package objects defined in the JSON Schemas.

Files (total 8)	**Schemas (total 143)**
• Issuers file • Manifest file • Stakeholders file • Stock classes file • Stock legend templates file • Stock option plans file • Transactions file • Vesting terms file	• Types • Enums • Objects

Our work is based on commit `20f3ede62d1f5bdbef16ae1edfa98c34fbda2610`, from 2022-Dec-30.

3.1 Objects, Types, and Enums

In this section, we will explore the different categories within the data model, including objects, types, and enums. Each category plays a distinct role and contributes to the overall structure and functionality of the model.

3.1.1 Types

The `types` directory contains schema files that define various types used in the model. These types serve as the foundation for creating structured data elements within the model. They define the format and constraints for specific data elements, ensuring consistency and integrity.

For example, the `Numeric.schema.json` file defines the type for numeric values, while the `Date.schema.json` file defines the type for representing dates. Other types may include addresses, monetary values, fractions, and more. These types not only provide the structure for the data but also enable validation by specifying required properties and data formats.

A useful type, for example, is the Monetary type. It defines a Monetary value as document with a Numeric value and a Currency Code value:

```
{
  "\$schema": "http://json-schema.org/draft-07/schema",
  "\$id": "https://opencaptablecoalition.com/schema/types/Monetary.schema.json",
  "title": "Type - Monetary",
  "description": "Type representation of an amount of money in a specified currency",
  "type": "object",
  "properties": {
    "amount": {
      "description": "Numeric amount of money",
      "\$ref": "https://opencaptablecoalition.com/schema/types/Numeric.schema.json"
    },
    "currency": {
      "description": "ISO 4217 currency code",
      "\$ref": "https://opencaptablecoalition.com/schema/types/CurrencyCode.schema.json"
    }
  },
  "additionalProperties": false,
  "required": ["amount", "currency"],
  "\$comment": "Copyright © 2022 Open Cap Table Coalition (https://opencaptablecoalition
}
```

Listing 1: The Monetary type

3.1.2 Enums

The `enums` directory contains schema files that define enumerations within the model. Enums are used to represent a predefined set of values or options for specific properties. They provide a limited and consistent set of choices that can be assigned to certain attributes within objects.

For instance, the `TerminationWindowType.schema.json` file defines different types of termination windows, such as "Termination for Cause" or "Termination without Cause." Enums help

to standardize and control the possible values for specific attributes, ensuring data consistency and providing a clear set of options to choose from.

```
{
"$schema": "http://json-schema.org/draft-07/schema",
"$id": "https://opencaptablecoalition.com/schema/enums/TerminationWindowType.schema.jso
"title": "Enum - Termination Window Type",
"description": "Enumeration of termination window types",
"type": "string",
"enum": [
  "VOLUNTARY\_OTHER",
  "VOLUNTARY\_GOOD\_CAUSE",
  "VOLUNTARY\_RETIREMENT",
  "INVOLUNTARY\_OTHER",
  "INVOLUNTARY\_DEATH",
  "INVOLUNTARY\_DISABILITY",
  "INVOLUNTARY\_WITH\_CAUSE"
],
}
```

Listing 2: The Termination Window Type enum

3.1.3 Objects

The `objects` category represents the entities or concepts within the data model. Objects are composed of properties, which describe the attributes or characteristics of the entity they represent. Examples of objects include issuers, stakeholders, securities, and transactions.

Objects serve as the core components of the model, encapsulating both the data and behavior related to specific entities. They allow for the representation of real-world concepts and enable the modeling of relationships and interactions between different entities.

As an example, we show the Issuer type (abriged):

By understanding the distinctions between objects, types, and enums, we gain a clearer perspective on the overall structure and purpose of the data model. These categories work together to define the data elements, provide validation, restrict attribute values, and represent the entities within the model.

```
{
    "$schema": "http://json-schema.org/draft-07/schema",
    "$id": "https://opencaptablecoalition.com/schema/objects/Issuer.schema.json",
    "title": "Object - Issuer",
    "description": "Object describing the issuer of the cap table (the company whose cap
    "type": "object",
    "allOf": [
        {
            "$ref": "https://opencaptablecoalition.com/schema/primitives/objects/Object.
        }
    ],
    "properties": {
        "object_type": {
            "const": "ISSUER"
        },
        "tax_ids": {
            "title": "Issuer - Tax ID Array",
            "description": "The tax ids for this issuer company",
            "type": "array",
            "items": {
                "$ref": "https://opencaptablecoalition.com/schema/types/TaxID.schema.jso
            }
        },
        "email": {
            "description": "A work email that the issuer company can be reached at",
            "$ref": "https://opencaptablecoalition.com/schema/types/Email.schema.json"
        },
        "phone": {
            "description": "A phone number that the issuer company can be reached at",
            "$ref": "https://opencaptablecoalition.com/schema/types/Phone.schema.json"
        },
        "address": {
            "description": "The headquarters address of the issuing company",
            "$ref": "https://opencaptablecoalition.com/schema/types/Address.schema.json"
        },
        "id": {},
        "comments": {}
    },
    "required": ["legal_name", "formation_date", "country_of_formation"],
}
```

Listing 3: The Issuer object

3.2 File Structure

3.2.1 OCF Manifest File

The `OCF Manifest File` packages all data in a single artifact (a JSON file). Indeed, the OCF represents a data model, that is kept in files, which are collections of JSON documents. It is this file that is to be exchanged by the parties involved in the transactions.

The `OCFManifestFile` contains the following components:

File	Description
Issuer	The issuer of the securities.
Stakeholders file	A file containing stakeholders.
Stock classes file	A file containing stock classes.
Stock legend templates file	A file containing stock legend templates.
Stock plans file	A file containing stock plans.
Transactions file	A file containing transactions.
Valuations file	A file containing valuations.
Vesting terms file	A file containing vesting terms.

3.2.2 Stakeholders File

Contains stakeholders, which are persons or organizations that own securities. Stakeholders are assigned to securities in issuance transactions.

3.2.3 Stock Classes

Contains stock classes, which are types of securities. Stock classes can be preferred or common. Preferred stock classes gives the stakeholder certain rights, and preferred stock securities can usually be converted to common stock securities.

3.2.4 Stock Legend Templates

Contains stock legend templates, which are templates for stock legends. Stock legends are text that is printed on the stock certificate.

3.2.5 Stock Plans

Contains stock plans, which are pools for granting stock options to employees. Stock options give the right but not the obligation to buy a certain amount of stock at a certain price, and as the stock

appreciates, the stock options become more valuable. Stock option plans are a key instrument in startup financing.

3.2.6 Transactions File

Contains transactions, which can be initial, non-terminal and terminal. Initial transactions are transactions that create securities. Non-terminal transactions are transactions that accept or adjust securities, including vesting events and exercises. If a non-terminal transaction generates a security, a matching initial transaction must be present. Terminal transactions are transactions that terminate securities, including sales and transfers.

Transaction Type	Convertible	PlanSecurity	Stock	Warrant
Acceptance	Yes	Yes	Yes	Yes
Cancellation	Yes	Yes	Yes	Yes
ClassAuthorizedSharesAdjustment			Yes	
ClassConversionRatioAdjustment			Yes	
ClassSplit			Yes	
Conversion	Yes		Yes	
Exercise		Yes		Yes
Issuance	Yes	Yes	Yes	Yes
PlanPoolAdjustment			Yes	
Reissuance			Yes	
Release		Yes		
Repurchase			Yes	
Retraction	Yes	Yes	Yes	Yes
Transfer	Yes	Yes	Yes	Yes

3.2.7 Valuations

Contains valuations, which are observations of the value of the company. Valuations are used to calculate the value of securities.

3.2.8 Vesting Terms

Contains vesting terms, which are the representation of legal documents that define schedules for vesting events. Vesting events are events that cause securities to vest, which means that the securities become unrestricted and can be sold or transferred.

Vesting terms form a small domain-specific language for describing vesting schedules. The OCF can model both event-based vesting and time-based vesting. Event-based vesting is when

vesting events are triggered by events, such as the company being acquired. Time-based vesting is when vesting events are triggered by time, such as the passage of time.

3.3 Tracing Security Transactions

In this section, we will explore the process of tracing security transactions within the data model. Tracing security transactions is crucial for accurately tracking the ownership and movement of securities throughout their lifecycle.

The case below shows the story of a single security, as traced by the transactions:

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)

Each of those transactions has a `security_id` field. Securities, in the OCF, are correlation identifiers for tracing transactions. By querying the transactions file for all transactions refering to a given security id, we can trace the history of a security.

Transactions might also have two additional fields: `balancing_security _id` and `resulting_securi` Partial transactions such as partial cancellations and partial transfers effectively destroy the original security, and balances are kept in a new balancing security. Transfers also create new securities, so they have a `resulting_security_ids` field.

Thus the general form of a transaction, in the OCF, is:

Here limitations of JSON Schema appear:

Thus, in principle, the transaction tracing system which is a key enabler of auditability, can not be validated completely within the current implementation in JSON Schema.

Constraint	Explanation
Ordering of transactions	Transactions must appear in a certain order and must be linked by their security ids. JSON Schema cannot express this constraint, at least not naturally[1].
Balancing of transactions	The sum of shares entering and exiting a transaction must be balanced. JSON Schema can not perform arithmetic checks.
(Non-)cyclicality of the implied transaction graph	More subtly, the graph formed by transactions can not be checked for cycles. This is a problem because cycles in the graph of transactions would mean that a security is being created out of thin air, which is not possible.

[1]We have not ruled out the possibility that such constraint could be encoded in JSON Schema some how.

Show how transactions can create securities, and how cancellations and transfers use a balancing security to allow partial cancellations and partial trans-

3.4 Vesting Terms

In this section, we show how the vesting terms define the conditions and schedules for the gradual vesting of securities. Here, the OCF is immensely valuable for accomodating the complexity of vesting terms, which are usually defined in legal documents.

3.4.1 Vesting transactions

3.4.2 Vesting objects, types and enums

3.4.3 The DSL embedded in the vesting system

This is clearly a (proto-)domain specific language for vesting terms, which is embedded in the OCF. As we will develop in the following chapters, this idea of a DSL can be expanded to accomodate even more complex vesting terms, by introducing propositional logic operators and more date operators.

3.4.4 Useful types and enums in the vesting system

Beyond the way that the vesting terms compose, the OCF also provides useful types for specifying dates, rounding methods and quantities that can be given as percentages, fractions or absolute values.

3.5 Conversion Mechanisms

In this section, we show how conversion mechanisms enable the 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 equity.

3.5.1 Conversion transactions

3.5.2 Conversion objects, types and enums

Give the transactions for the vesting systems

types and enums involved in vesting transactions

Show the composition of vesting terms, vesting conditions and vest-

3.5.3 The conversion process

3.6 Evaluation of the Data Model

3.6.1 Review of the building blocks

In this chapter, we showed how the OCF is based upon:

- **Transactions** and their tracing, which are the basic operations that can be performed on securities;
- **Vesting terms**, which are the representation of legal documents that define schedules for vesting events;
- **Conversion mechanisms**, which are the rules for converting securities from one type to another.

That are built on the following basic blocks:

- **Objects**, **types** and **enums**, which are the basic building blocks of the data model;
- **Files**, which are collections of JSON documents that are exchanged between parties;

3.6.2 Design patterns

1. The use of objects and types is similar to the use of entities and values in domain-driven design
2. The modeling of securities as correlation identifiers is similar to the use of correlation identifiers in event sourcing
3. The consuming and producing of securities in transactions is somewhat similar to BitCoin's unspent transaction outputs (UTXO) where every transaction consumes a set of UTXOs and produces a set of UTXOs, similar to how transactions consume securities and produce resulting securities and balancing securities

Show how conversion always means consuming a security in exchange for another in another class, still respecting security tracing

Show how complex it is to define the rules for conversion, because they

3.6.3 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 (particulary regarding transaction tracing) and flexibility (via the domain-specific languages implicit in the vesting and conversion systems).

3.6.4 Disadvantages and limitations of the OCF

The choice of JSON Schema can arguably be defended on the grounds that 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 of the OCF.

JSON Schema fails to express invariants of the system. The JSON Schema itself is not able to 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. Examples are trivial.

Give simple examples that the OCF accepts but should reject

3.7 Next steps

In the next chapter, we will explore how certain parts of the OCF can be modeled in Alloy with a much higher degree of precision and expressiveness.

Give appendix for a network analysis that gives the components de-

```
{
        "$id": "objects/transactions/issuance/StockIssuance.schema.json",
        "type": "object",
        "allOf": [
                {
                "$ref": "primitives/objects/Object.schema.json"
                },
                {
                "$ref": "primitives/objects/transactions/Transaction.schema.json"
                },
                {
                "$ref": "primitives/objects/transactions/SecurityTransaction.schema.jso
                },
                {
                "$ref": "primitives/objects/transactions/issuance/Issuance.schema.json"
                }
        ],
        "properties": {
                "object_type": {
                "const": "TX_STOCK_ISSUANCE"
                },
                "stock_class_id": {
                "type": "string"
                },
                "share_numbers_issued": {
                "type": "array",
                "items": {
                        "$ref": "types/ShareNumberRange.schema.json"
                }
                },
                "share_price": {
                "$ref": "types/Monetary.schema.json"
                },
                "quantity": {
                "$ref": "types/Numeric.schema.json"
                },
                "vesting_terms_id": {
                "type": "string"
                },
                "cost_basis": {
                "$ref": "types/Monetary.schema.json"
                },
                "stock_legend_ids": {
                "type": "array",
                "items": {
                        "type": "string"
                }
                },
                "id": {},
                "comments": {},
                "security_id": {},
                "date": {},
                "custom_id": {},
                "stakeholder_id": {}
```

Chapter 4

The Alloy model

4.1 Design of the Model

In this section, we discuss the design of our capitalization table model in detail. Our model is based on the Open Cap Table format, and we discuss the general strategy for the translation and design of our model.

We migrate objects and transactions as Alloy signatures, and add constraints and queries to the model to capture the invariants of the problem domain and show how certain information can be extracted from the model. Those invariants are implied either by the author's business expertise or by the validations defined in JSON Schema.

We do not migrate parts of the model that contribute little to the thesis, such as unusual or uncommon but trivially implementable transactions, or types of values that require only equality to be supported (id) or that carry values that are not restricted by invariants.

4.1.1 Selected features

Feature	Description	Improvements
		vs OCF
Transaction Tracing	A security entity that contains a sequence of transactions	Invariants that discard invalid or meaningless sequences of transactions
Vesting System	A model for composing rules of both schedule-based and event-based vesting	Additional expressivity by allowing logical operators over vesting rules (AND, OR, NOT)
Conversion Mechanisms	A model for converting between different types of securities	Direct translation
Basic entities	A model for basic entities such as securities, shareholders, and issuers (companies)	Constraints over the possible relationship graph of instances, such as requiring disjointness of certain sets

Features that are left aside:

Feature	Description	Rationale
Types of Values	Types of values such as addresses, dates, and numbers	Alloy can directly refer to instances without ids, and many types can be subsumed by a single type (e.g., Address instead of a full record with all address components)
Uncommon transactions	Some transactions in OCF which are less common, like reissuances and retractions of securities	Leaving them out simplifies our model without reducing its usefulness

4.2 Implementation in Alloy

The implementation of our capitalization table model in Alloy is the focus of this section. We discuss how we translated the model into Alloy code, and provide examples of how to use Alloy to analyze the model. We also discuss the benefits and limitations of using Alloy for this type of modeling.

4.3 Evaluation and Testing of the Model

In the final section of this chapter, we evaluate our experience in using Alloy to model capitalization tables. We discuss the benefits and limitations of using Alloy for this type of modeling, and provide recommendations for further development and refinement of the model.

What we find is that:

1. The reliability of the model is much higher than that of the original OCF specification, as the model carries invariants that reject invalid transactions.

2. With Alloy, the resulting model can be used to characterize and check specific business cases and generate instances, greatly improving how the format can be used by practitioners via rich examples.

3. During the construction of the model, the understanding of the problem domain grows, because the feedback from the tool is very quick and the modeler can test assumptions.

4. Tests, both unitary or integrative, can be defined as checks and asserts, with useful feedback on negative cases (the model's UNSAT core, technically)

In terms of testing, we calculate that each check and assertion, considering that Alloy allows fine control of the size of the scope as it carries a bounded model checker. This gives the following estimates as to how many instances are eliminated by each check.

For example, a model that has two signatures with scope=3 has 6 bits of information, or 64 cases. But a model with 8 signatures and scope=4 has actually 32 bits, or 4 billion cases.

Chapter 5

Transaction Tracing

Here we present a model of transaction tracing in Alloy, based on the OCF. The model is a stylized version of the OCF, and it is not intended to be a complete model of the OCF.

In this chapter, we choose to reduce the transaction system do its essential form[?]. The key abstraction in the transaction system is that of *securities* which under go *transactions*.

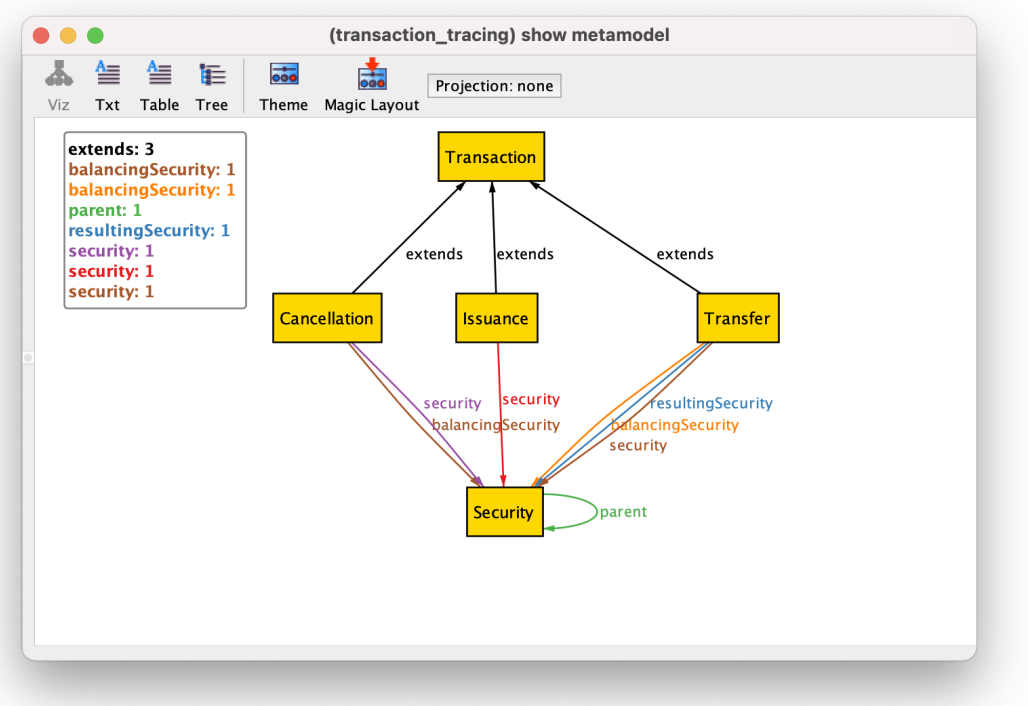
In the original model, transactions take input securities and have output securities. Although there is a large number of different specialized transactions in the model, having Issuances, Cancellations and Transfers is enough to find the deep abstraction that allows securities to be traced to their original creation event (using Alloy), and thus form a history of a security including its lineage of parent securities.

One key problem in this is that writing a bunch of abstractions and then merging them in an actual programming language seldom works[?]. The reason, according to Daniel Jackson, is that:

> The environment of programming is so much more exacting than the environment of sketching design abstractions. The compiler admits no vagueness whatsoever.[?]

Alloy can readily provide a visualization of what is called the meta-model (Alloy's representation of the model).

Give a more precise de-scrip-



5.1 Signatures

5.1.1 The Security

```
sig Security {
    shares : one Int,
    parent : lone Security
}
```

Listing 5: The Security signature

The `Security` signature represents a security and includes the number of shares (`shares`) and a reference to its parent security (`parent`).

5.1.2 The Abstract Transaction

```
abstract sig Transaction {}
```

Listing 6: The Abstract Transaction signature

The `Transaction` signature is an abstract signature representing a transaction.

5.1.3 The Issuance

```
sig Issuance extends Transaction {
        security : one Security,
        amount : one Int
}
```

Listing 7: The Issuance signature

The Issuance signature represents an issuance transaction, which creates a security with a specified amount of shares.

5.1.4 The Cancellation

```
sig Cancellation extends Transaction {
        security : one Security,
        amount : one Int,
        balancingSecurity : lone Security
}
```

Listing 8: The Cancellation signature

The Cancellation signature represents a cancellation transaction, which destroys a certain amount of shares from a security. In case of a partial cancellation, a balancingSecurity is created to balance the number of shares.

5.1.5 The Transfer

```
sig Transfer extends Transaction {
        security : one Security,
        amount : one Int,
        from : one Stakeholder,
        to : one Stakeholder,
        balancingSecurity : lone Security,
        resultingSecurity : one Security
}
```

Listing 9: The Transfer signature

The Transfer signature represents a transfer transaction, which moves a certain amount of shares from one stakeholder to another. If the transfer is partial, a balancingSecurity is created to balance the number of shares, and a resultingSecurity represents the resulting security after the transfer.

5.1.6 The Stakeholder

```
abstract sig Stakeholder {}
```

Listing 10: The Stakeholder signature

The Stakeholder signature is an abstract signature representing a stakeholder.

5.2 Constraints

5.2.1 Requiring that all securities are created by an issuance

These constraints ensure that every security involved in a cancellation or transfer is created by an issuance.

```
fact giveIssuanceToBalancingSecurity {
        all c : Cancellation | some c.balancingSecurity implies one i : Issuance
}

fact giveIssuanceToResultingSecurity {
        all t : Transfer | one i : Issuance | i.security = t.resultingSecurity
}
```

Listing 11: Requiring that all securities are created by an issuance

5.2.2 Avoiding cycles

```
fact noSelfParent {
        no s : Security | s in s.^parent
}
```

Listing 12: Avoiding cycles

This constraint ensures that there are no cycles in the parent-child relationship of securities.

5.2.3 Constraining the parent-child relationship

```
fact establishParent {
        all t : Transfer | some t.balancingSecurity => t.balancingSecurity.paren
        all t : Transfer | t.resultingSecurity.parent = t.security
        all i : Issuance | no i.security.parent
        all c : Cancellation | some c.balancingSecurity => c.balancingSecurity.p
}
```

Listing 13: Constraining the parent-child relationship

These constraints establish the parent-child relationship between securities and their balancing securities, ensuring that the model is prepared for tracing.

5.2.4 Giving meaningful values to amounts

```
fact issuancesAlwaysPositive {
        all i : Issuance | pos[i.amount]
        all c : Cancellation | pos[c.amount]
        all t : Transfer | pos[t.amount]
}
```

Listing 14: Giving meaningful values to amounts

These constraints enforce that the amounts in issuances, cancellations, and transfers are always positive.

5.2.5 Balancing the number of shares

These constraints ensure that the number of shares is balanced in cancellations and issuances.

```
fact cancellationBalances {
        all c : Cancellation
        | some c.balancingSecurity <=> add[c.balancingSecurity.shares, c.amount]

        all c : Cancellation
        | no c.balancingSecurity <=> c.amount = c.security.shares
}

fact issuanceBalances {
        all i : Issuance | i.security.shares = i.amount
}
```

Listing 15: Balancing the number of shares

5.3 Funs, or queries

```
fun history[s : Security] : set Transaction {
        s.*parent
}
```

Listing 16: The `history` fun

The `history` function returns the set of transactions that form the history of a given security.

5.4 Examples

```
e1 : run {
    some Cancellation
}
```

Listing 17: Example 1

```
e2 : run {
        some Transfer
}
```

Listing 18: Example 2

```
e3 : run { some Issuance }
```

Listing 19: Example 3

These examples demonstrate the usage of the model by running certain queries.

As a last example, we show how the improved model can picture long sequences of transactions, implied by many partial cancellations and transfers:

Figure 5.1: An example of a cancellation transaction

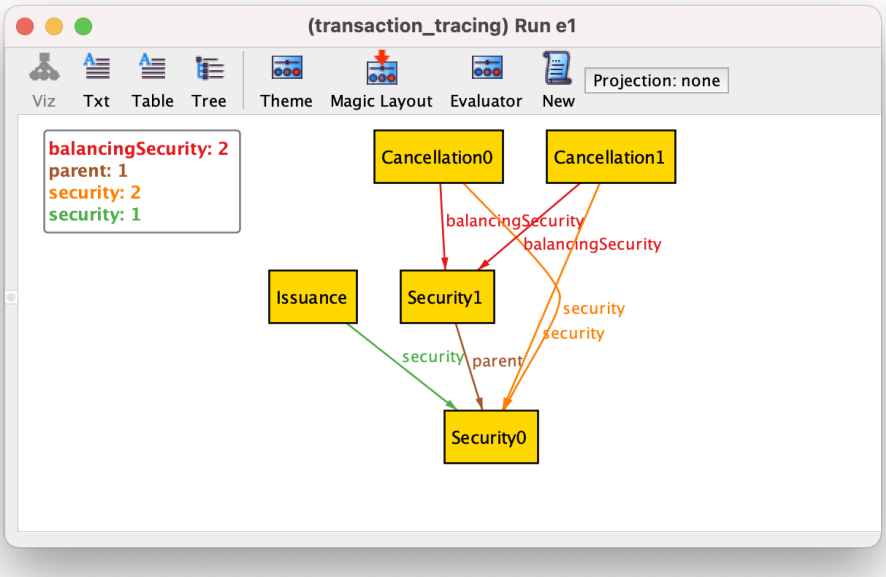


Figure 5.2: An example of a transfer transaction

5.5 Discussion

By introducing a parent relationship to the security model, we were able to use Alloy's relational semantics to constrain the whole transitive closure of the parent relationship. This allowed us to model transaction tracing in a very robust way.

Furthermore, the constraints over the specific transaction types are required to validate them, and would not be possible in JSON Schema.

The translation was straightforward by declaring signatures and proceeding to constrain the signatures and relations based on the expected behavior of each transaction, according to the domain that is under consideration.

Checking the model was fast and brought solid confidence in the correctness of the model, by ruling out large classes of invalid models. We can point out that by choosing a size of 3 for the scope, we are considering a dizzying number of possible models[1], and yet the model checker was able to rule out all invalid models in a matter of seconds.

[1]Certainly beyond the coverage provided by unit tests

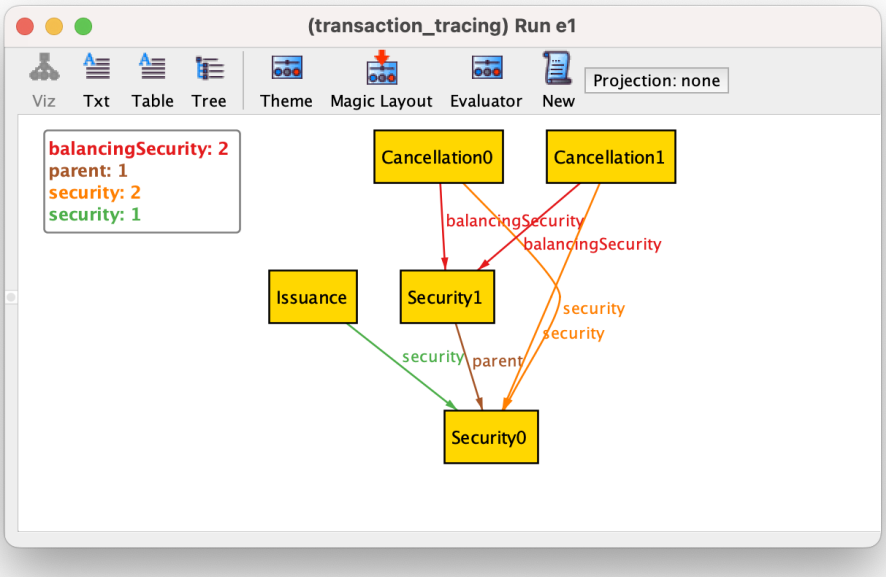


Figure 5.3: An example of an issuance transaction

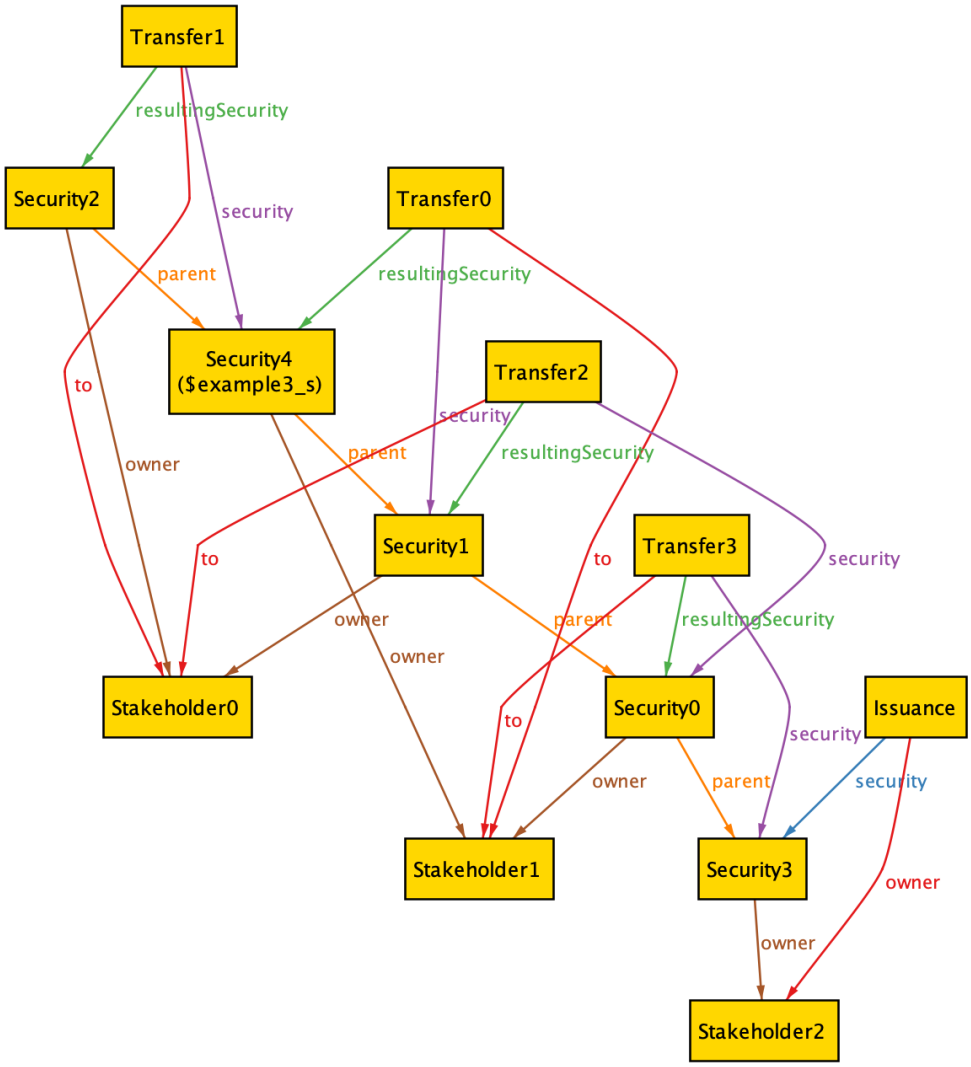


Figure 5.4: An example of a long chain of transactions

Chapter 6

Stock Option Plan Model

6.1 Introduction

Stock option plans are a pool of shares that are reserved to be granted as stock options to employees. Simply put, a stock option gives the right (but not the obligation) to buy a certain amount of stock at a certain price. As the stock appreciates, the stock options become more valuable.

To make it more concrete to the reader, a typical stock option contract might look like the following:

The Stock Option Plan Model is a formal representation of stock option plans used by companies to incentivize employees and other stakeholders.

6.2 Model Design

The model is designed around signatures that represent the stock option plan, the security that represents stock options issued from that stock option plan, the transactions that issue stock options, and the transactions that exercise stock options. In the OCF, an event is registered whenever new shares are vested (become available for exercises).

In our model, we leverage Alloy's higher-arity relations to directly give the number of shares that are vested at a given date.

Stock Option Agreement

You have been awarded an option to acquire stocks in the future from ACME Corporation.

Company	ACME Corporation
Optionee	Alan Turing
Vesting Term	You will receive 1/4th of your options after one year Starting from the first year, you will receive the remaining installments monthly during the next 3 years
Strike Price	The exercise price for the options is US$ 10,00 per share
Acceleration	In case the company is acquired, all shares are immediately considered vested

Optionee

Company Executive

Figure 6.1: Example of a stock option contract

6.3 Signatures

6.3.1 Stock Plan Signature

First and foremost, all stock option plans have a number of reserved shares. Indeed, a stock option plan is sometimes called an option pool[?].

From the stock option plan, stock option grants will be awarded as securities, and, as those

options became available to optionees (vested), they can be exercised[?]. We include relations to exercises and securities in the stock option plan signature to aid in navigating the model.

```
sig StockPlan {
    reservedShares : one Int,
    exercises : disj set Exercise,
    securities : disj set Security
}
```

Listing 20: The StockPlan signature

The disj modifiers are used so that every exercise and security are unique to a stock option plan.

6.3.2 The Security Signature

The security signature represents a security that is issued from a stock option plan. The security has a number of shares, and a reference to the stock option plan from which it was issued, as well as to the vesting term object that define its vesting schedule.

```
sig Security {
    vestingTerm : disj one VestingTerm,
    shares : one Int,
    stockPlan : one StockPlan,
}
```

Listing 21: The Security signature

Since each VestingTerm is exclusive to a Security, we use the disj modifier. Since a StockPlan has many securities, it is not necessary to make the relation disjoint.

6.3.3 Issuance signature

The issuance signature is the transaction that creates a new stock option from a stock option plan. It refers to the newly issued Security. It also assigns a VestingTerm to the Security, which defines the vesting schedule for the security.

```
sig Issuance {
        security : disj one Security,
        shares : one Int,
        vestingTerm : disj one VestingTerm,
        stockPlan : one StockPlan
}
```

Listing 22: The Issuance signature

Perhaps the disj modifier here is superfluous

6.3.4 Exercise signature

As shares become vested according to the vesting term, the optionee is able to exercise those options into actual shares. A resulting security is thus created. Exercises can be many, as long as the total number of exercised shares does not exceed the total number of vested shares.

```
sig Exercise {
        security : one Security,
        resultingSecurity : one Security,
        stockPlan : one StockPlan,
        shares : one Int
        } { pos[shares] }
```

Listing 23: The `Exercise` signature

The `pos[shares]` constraint ensures that the number of exercised shares is always positive. This excludes the uneffectful exercise of zero shares.

6.3.5 Vesting Term signature

As regards the stock option plan model, we adopt a simplified version of the vesting term, without loss of generality. In this case, we model the vesting term as having higher-arity relation mapping numbers of shares to status of vesting.

```
sig VestingTerm {
    security : one Security,
    vestingConditions : set Status -> one Int
}

enum Status { Vested, Pending }
```

Listing 24: The `VestingTerm` signature

6.4 Functions (Queries)

Since our focus here is on respecting accounting principles, we define functions that allow us to calculate the numbers of vested shares, and from that point we can constrain the model to respect the accounting principles.

Even though Alloy is not aimed at arithmetic modeling, because ints are restricted to relatively small bit-width, we can use them to model the vesting term as a function that maps the number of vested shares to the status of vesting.

```
fun vestedShares[v : VestingTerm] : one Int {
    sum[v.vestingConditions[Vested]]
}

fun pendingShares[v : VestingTerm] : one Int {
    sum[v.vestingConditions[Pending]]
}

fun exercisedShares[v : VestingTerm] : one Int {
    let es = { e : Exercise | e.security = v.security } |
    sum[es.shares]
}
```

Listing 25: The `vestedShares`, `pendingShares`, and `exercisedShares` functions

```
fact {
    all s : Security | pos[s.shares]
    all i : Issuance | pos[i.shares]
    all e : Exercise | pos[e.shares]
}
```

Listing 26: The `shares` constraint

6.5 Constraints

6.5.1 All shares must be positive in the signatures

6.5.2 Relational constraints

These constraints are useful to ensure that the bi-directional relations are really inverses. This is a very idiomatic way of constraining the model, and it is very useful to ensure that the model is well-formed.

```
fact {
    stockPlan = ~securities
}

fact {
    ~security = vestingTerm
}
```

Listing 27: The `relational` constraints

6.5.3 Accounting equations

This is at the heart of the stock option plan system, and it is something that JSON Schema can not hope to express. We can constrain the model so that it is not possible to over-vest or over-exercise shares.

```
fact {
    all s : Security |
    gte[s.shares, sum[s.vestingTerm.vestingConditions[Status]]]
}
```

Listing 28: The `vesting` constraint

This makes sure that share numbers are not over-vested.

```
fact {
    all v : VestingTerm | lte[v.exercisedShares, v.vestedShares]
}
```

Listing 29: The `exercise` constraint

This makes sure that share numbers are not over-exercised.

Chapter 7

Vesting System

7.1 Introduction

In this chapter, we present a model for the vesting schedule domain-specific language (DSL). A vesting schedule is a contractual agreement between an employer and an employee that governs the employee's right to receive equity or other forms of compensation over time. The vesting schedule typically consists of a series of vesting periods during which the employee earns the right to receive a portion of their equity or compensation. The vesting periods are governed by a set of conditions and triggers, which determine when the employee is entitled to receive their equity or compensation. The vesting schedule DSL provides a way to express these conditions and triggers in a formal language, allowing us to reason about and analyze vesting schedules more easily. In this chapter, we will present our model for the vesting schedule DSL and discuss its features, including vesting terms, conditions, and triggers, as well as unary and binary operators for composing and manipulating these constructs. We will also present interesting cases and applications of our model.

7.2 Model Overview: Vesting Terms, Conditions, and Triggers

Our model for the vesting schedule DSL is based on three main constructs: vesting terms, conditions, and triggers.

The preliminaries are giving an ordering to dates, and defining a graph for vesting triggers, so we can make them well-formed (i.e., acyclic, since they are like an abstract syntax tree).

```
open util/ordering[Date]
open util/graph[VestingTrigger]

sig Date {}
```

Listing 30: The Date signature

A vesting term represents a period of time during which an employee earns the right to receive a portion of their equity or compensation.

```
abstract sig VestingTerm {
    conditions : disj set VestingCondition,
    shares : one Int
}
```

Listing 31: The VestingTerm signature

A vesting condition represents a condition that must be satisfied for the employee to earn the right to receive their equity or compensation.

```
abstract sig VestingCondition {
    trigger : disj one VestingTrigger,
    shares : one Int,
    term : one VestingTerm
}
```

Listing 32: The VestingCondition signature

Finally, we defined abstract triggers before giving the concrete cases.

```
abstract sig VestingTrigger {}

abstract sig UnaryOperator extends VestingTrigger {
    operand : one VestingTrigger
}

abstract sig BinaryOperator extends VestingTrigger {
    left : disj one VestingTrigger,
    right : disj one VestingTrigger
}
```

Listing 33: The VestingTrigger signature

7.3 Unary and Binary Operators: Until, After, Not, And, and Or

In addition to the basic vesting triggers described previously, our vesting schedule DSL also includes a number of unary and binary operators that can be used to combine and modify these triggers to create more complex vesting schedules.

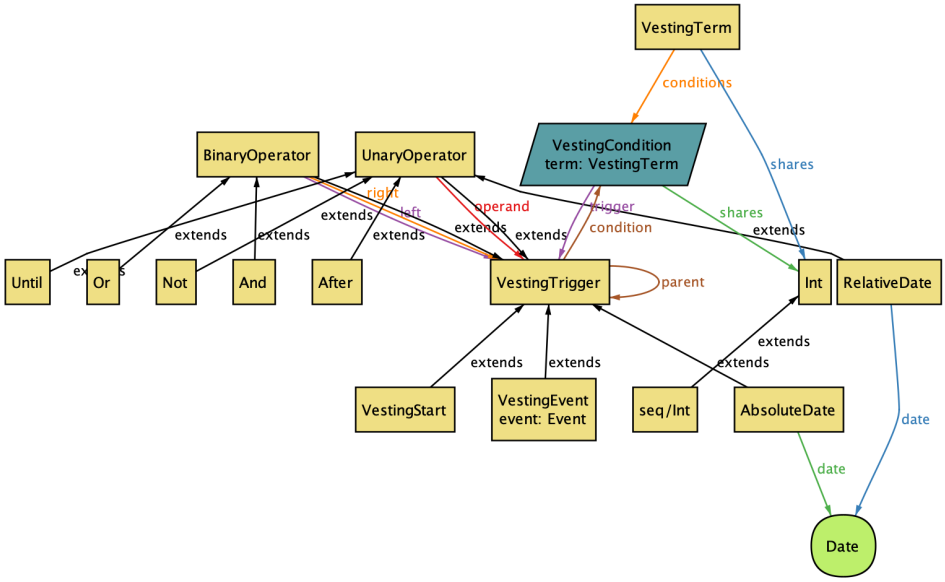


Figure 7.1: Vesting system

7.3.1 Unary Operators

Unary operators are operators that operate on a single trigger. In our DSL, we have defined three unary operators: Until, After, and Not.

Until

The Until operator is a unary operator that modifies a trigger to only trigger until a certain date. In Alloy, we define the Until operator as follows:

```
sig Until extends UnaryOperator {
    date : one Date
}
```

Listing 34: The `Until` signature

This operator can be used to model vesting conditions that expire on a certain date, or that can only be exercised within a certain time period.

After

The After operator is a unary operator that modifies a trigger to only trigger after a certain date. In Alloy, we define the After operator as follows:

```
sig After extends UnaryOperator {
    date : one Date
}
```

Listing 35: The `After` signature

This operator can be used to model vesting conditions that cannot be exercised until a certain date.

Not

The Not operator is a unary operator that negates a trigger. In Alloy, we define the Not operator as follows:

```
sig Not extends UnaryOperator {}
```

Listing 36: The `Not` signature

This operator can be used to model vesting conditions that are contingent on the absence of certain events or conditions.

7.3.2 Binary Operators

Binary operators are operators that operate on two triggers. In our DSL, we have defined two binary operators: And and Or.

And

The And operator is a binary operator that combines two triggers into a new trigger that only triggers if both triggers are true. In Alloy, we define the And operator as follows:

```
sig And extends BinaryOperator {}
```

Listing 37: The `And` signature

This operator can be used to model vesting conditions that are contingent on multiple events or conditions.

Or

The Or operator is a binary operator that combines two triggers into a new trigger that triggers if either trigger is true. In Alloy, we define the Or operator as follows:

```
sig Or extends BinaryOperator {}
```

Listing 38: The `Or` signature

This operator can be used to model vesting conditions that are contingent on either of multiple events or conditions.

By combining these unary and binary operators with the basic vesting triggers described earlier, we can create complex vesting schedules that model a wide range of vesting conditions.

Chapter 8

Conclusion and Future Work

8.1 Summary of Contributions

In this thesis, we showed how Alloy, a lightweight formal language, could be used instead to complement a data model written in JSON Schema, a data validation language.

8.2 Limitations and Open Issues

8.3 Future Directions for Research and Development

8.4 Conclusion

Colocar as contribuições check de transações aritméticos vesting, etc.

I should give that while Alloy is not particularly hard to use after learn-

Bibliography

[1] Carta — equity management solutions. https://carta.com/. (Accessed on 05/23/2023).

[2] Distu - equity value unlocked. https://www.distu.com.br/. (Accessed on 05/23/2023).

[3] Exercise: Definition and How It Works With Options — investopedia.com. https://www.investopedia.com/terms/e/exercise.asp. [Accessed 19-May-2023].

[4] Implementations — json-schema.org. https://json-schema.org/implementations.html. [Accessed 27-May-2023].

[5] JSON Schema Store — schemastore.org. https://www.schemastore.org/json/. [Accessed 27-May-2023].

[6] Ltse — home. https://equity.ltse.com/. (Accessed on 05/23/2023).

[7] Oct format — open cap table coalition. https://www.opencaptablecoalition.com/format. (Accessed on 05/26/2023).

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

[9] Option Pool: Definition, Purpose, How It Works, and Structure — investopedia.com. https://www.investopedia.com/terms/o/option-pool.asp. [Accessed 19-May-2023].

[10] Shareworks - equity compensation solutions. https://www.shareworks.com/. (Accessed on 05/23/2023).

[11] Software and services to organize africa's startups. `https://www.getraise.io/`. (Accessed on 05/23/2023).

[12] Bernardo F. B. Braga, João Paulo Andrade Almeida, Giancarlo Guizzardi, and Alessander B. Benevides. Transforming OntoUML into alloy: towards conceptual model validation using a lightweight formal method. *Innovations in Systems and Software Engineering*, 6(1-2):55–63, February 2010.

[13] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, IETF, December 2017.

[14] Renzo Carpio and Izzat Alsmadi. Websites security policies implementation using alloy analyzer. *SSRN Electronic Journal*, 2021.

[15] Charles Chen, Paul Grisham, Sarfraz Khurshid, and Dewayne Perry. Design and validation of a general security model with the alloy analyzer. 01 2006.

[16] Daniel Jackson. Lightweight formal methods. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, page 1, Berlin, Heidelberg, 2001. Springer-Verlag.

[17] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[18] Jeremy Johnson and Izzat Alsmadi. Formal modeling of banking policies using alloy analyzer. *SSRN Electronic Journal*, 2021.

[19] Eunsuk Kang, Santiago Perez De Rosso, and Daniel Jackson. 500 lines or less - the same-origin policy. `https://aosabook.org/en/500L/the-same-origin-policy.html`. (Accessed on 05/23/2023).

[20] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. Alloy∗: a general-purpose higher-order relational constraint solver. *Formal Methods in System Design*, 55(1):1–32, January 2017.

[21] Ido Milicevic, Aleksandar Erfrati and Daniel Jackson. arby—an embedding of alloy in ruby. In *Abstract State Machines, Alloy, B, VDM, and Z*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014.

[22] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, April 2016.

[23] Rodion Podorozhny, Sarfraz Khurshid, Dewayne Perry, and Xiaoqin Zhang. Verification of multi-agent negotiations using the alloy analyzer. In *Lecture Notes in Computer Science*, pages 501–517. Springer Berlin Heidelberg.