Case Study of Feature-Oriented Requirements Modelling, Applied to an Online Trading System

by

Ana Krulec

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The Feature-Oriented Requirements Modelling (FORM) combines the requirement engineering style structuring of requirements documents with the feature-orientation of the Feature Oriented Software Development, resulting in a feature-oriented model of the functional requirements of a system-under-development (SUD). A feature is a distinguishable unit of added value to the SUD. The objectives of FORM are to model features as independent modules, to allow the addition of new features with minimal changes to the existing features, and to enable automatic generation and checking of properties like correctness, consistency, and non-determinism. FORM structures requirements into three models: a domain model, a collection of behavioural models, and a collection of functional models. A feature is modelled by a distinct behavioural model. This dissertation evaluates FORM by applying it to a new application that can be thought of in terms of features, namely an online trading system (OTS) that receives requests from customers about buying or selling securities on a stock market. The OTS offers variability in terms of the types of orders that customers can request, (e.g. market order, limit order and stop order). The case study revealed six deficiencies of the FORM notation, three of which were easily overcome. The dissertation presents the results of the case study, resolutions to three of the six deficiencies, and an outline of an approach to resolve the other three deficiencies.

Acknowledgements

I would like to thank my supervisor, Dr. Joanne Atlee for her help with the dissertation. She gave me inspiration, has lent me guidance and knowledge for the past two years, and has provided me with much needed ideas. I would also like to thank Pourya Shaker for guidance and help with FORM and Alloy.

I would like to thank Dr. Daniel Berry and Dr. Michael Godfrey who took their time to read my thesis and provide valuable input to improve quality of this research.

Finally, I would like to thank Dr. Andrej Brodnik who opened the possibility for studying abroad for me; and my mom and brother, for much needed encouragement to finish what I started.

Mami, očetu, in Roku

Table of Contents

List of Figures

Chapter 1

Introduction

Software development starts when customers sit down with engineers and discuss the requirements of a proposed software system, hereafter called the system under development (SUD). The customers and engineers have to work together to describe in detail the behaviour and properties of the SUD. After they agree on the requirements, the engineers can start discussing how they will implement the system. The requirements are expressed in terms of the benefits and effects that the SUD will have on its environment. Requirements engineering (RE) strives to organize, structure and describe the detailed requirements in terms of multiple views: a behavioural or functional model of the SUD, a structural or object-oriented (OO) model of the phenomena of the SUD's environment, and constraints on the SUD and its environment. Object Oriented Analysis (OOA) is representative of such RE methods [11].

Feature Oriented Software Development (FOSD) is a software development process that structures an SUD's design and implementation in terms of its features. A feature is a "distinguishable characteristic of the software that is important to individuals in developing the product" [12, 3]. Every feature adds value to the SUD, either in the form of new functionality or new nonfunctional qualities. Because each feature is a distinct enhancement, stakeholders often discuss and reason about an SUD in terms of its constituent features, and express changes to the system by adding, modifying, or deleting individual features.

In most cases of software development, there is no clean mapping between features defined in the early stages of requirements and the features' respective implementations. In contrast, FOSD tries to explicate features at all stages of software development, and

in all artifacts of the development. An SUD is constructed by combining and integrating features, and feature modules can be components of multiple software systems [1].

Shaker et al.'s Feature-Oriented Requirements Modelling (FORM) methodology combines the RE-style structuring of requirements documents with the feature-orientation of FOSD, resulting in a feature-oriented model of the functional requirements of an SUD. The objectives of FORM are to combine *modularity*, so that features can be modelled as independent modules; *modifiability*, so that new features can be added with minimal changes to the existing features; and *precision*, to enable automatic generation and checking of properties like correctness, consistency, and non-determinism. The FORM methodology structures requirements into three models: a *domain model*, a collection of *behavioural models*, and a collection of *functional models*. Features are modelled as distinct behavioural models. The FORM methodology assumes that there are one or more base features that specify the required behaviour of the SUD. Features that override the base functionality are modelled as enhancement features, which in turn may again be base features for some other enhancement features. Unplanned feature interactions are resolved with predefined precedence rules [12].

The purpose of this dissertation is to evaluate FORM by applying it to a new application domain that can be thought of in terms of its features. An evaluation modelled an online trading system (OTS) that receives requests from customers about buying or selling securities on a stock market. The OTS offers variability in terms of the types of orders that customers can request (e.g., market order, limit order, stop order, etc.). In the OTS, each type of order is modelled as a different feature. Market orders are modelled as base features. Other types of trade orders are modelled as enhancement features that extend or override the market-order feature. The OTS is smart enough to process orders until they are ready to be entered into the market. As such, the provider trading system (PTS) will not be aware of a requested limit order until the price of the stock on the market has reached the required limit price.

The case study revealed six deficiencies of the FORM notation: three that we were able to resolve through small extensions to FORM and three deficiencies that require more substantial consideration. In the latter cases, we provide sketches of how FORM might be extended to address the new behaviour.

In the rest of the dissertation, Chapter 2 describes a trading system and required ter-

minology to understand the system we modelled. Chapter 3 describes Feature Oriented Requirements Modelling (FORM). Sections 3.1 and 3.2 describe requirements modelling. Section 3.3 then describes Feature Oriented Software Development which breaks down a software system into distinguishable characteristics. After that, Section 3.5 describes FORM that addresses a particular part of FOSD. Section 3.4 describes the Alloy modelling language and how it is employed in FORM. Chapter 4 describes our application of FORM to the modelling an online trading system. Section 4.1 describes the domain model and sections 4.2 and 4.3 describe behavioural models. Chapter 5 describes our experiences conducting the case study. Section 5.2 discuses minor deficiencies in FORM that we encountered and for which we successfully devised new notation constructs or modelling guidelines and section 5.3 discusses major deficiencies for which we devised workarounds, but which need future investigation. Chapter 6 is a conclusion. The complete FORM models are given in the appendices.

Chapter 2

An Example of an Online Trading System

In order to understand trading systems, the dissertation first presents the basic terminology and a short history of how trading evolved, that is derived from the description of the Stock Market in Wikipedia, [4]. The rest of the terminology required to understand this dissertation is given in Section 2.1. Section 2.2 describes different types of orders for buying and selling stocks.

A *trade* is "a voluntary, often asymmetric, exchange of goods, services, or money," [4] and a mechanism that allows trading is called a *market*. This dissertation concentrates on a *stock market*, which is a "public market for the trading of company's stock and derivatives at agreed price." [4] The first origins of the stock market are believed to have been in the 12th century in France, when "courratiers de change" were managing and regulating trade with agricultural communities on behalf of the banks. In the 13th century, the family Van der Beutze had a building in Antwerp where most of the merchants were performing trades. New places soon opened across Europe, and by the middle of the 13th century Venetian bankers began to trade government securities. The Dutch later started joint stock companies, so shareholders could invest in business ventures and get a share of their profits or losses. In 1602, the Dutch East India Company issued the first share on the Amsterdam Stock Exchange and became the first company to issue stocks and bonds. In the 17th century, the Amsterdam Stock Exchange introduced continuous trade [4].

There are several ways to exchange stocks on the stock market. Some exchanges happen

at physical locations, where traders meet face-to-face, and some are virtual, in which computers find matching buy and sell orders. Probably the most common association when mentioning a stock exchange is the pit of a trading floor where traders are standing in a circle around computer screens shouting and hand-signalling when placing bids, i.e., offers, and asks, i.e., requests for stocks. When the bidding and asking prices match, sales take place on a first-come-first-serve basis [4].

For example, the New York Stock Exchange is an example of a physical exchange. Members of the exchange enter orders with a floor broker, a person located in the exchange building who walks to the trading pit of that stock and trades the order. The specialist at the pit matches buy and sell orders. Once the trade has been made, the details are reported and sent back to the brokerage firm, which notifies investors. Today most of the processes, like order matching and data publishing, are starting to be managed electronically. NASDAQ is a completely virtual exchange market in which parties use a computer network for communication. It allows dealers to electronically enter their trades, which are matched automatically by a matching algorithm [4]. NASDAQ allows users to specify orders that are to be executed based on market prices or at a specific time; these options make trading easier for users who are not able to follow the market continuously.

More and more stock exchanges are turning into electronically managed trading systems, where investors or their brokers can enter orders electronically and have them matched automatically. Trading Exchange Expert defines a *Trading System* as "a set of rules, parameters, and indicators that determine entry and exit points during trading" [5]. It is composed of two parts: an online trading server and a trading engine. The online trading server is responsible for accepting orders from clients and sending ACK or NACK responses back to the client to communicate the receipt or rejection of the order, respectively. It also keeps a history, called an *order book*, of chronologically sorted buy and sell orders. At the back-end, the trading engine is responsible for executing the trades, i.e., performing matching algorithms on orders, and for related activities, like transferring shares and funds between accounts. If the matching function results in an actual match of a buyer and a seller, the trading server will communicate a success message back to the parties involved in the transaction. Whenever the trading engine makes a material change to the state of the order book, the trading server is charged with publishing the results of the trade, which are seen by the public.

Figure 2.1 shows an example message flow between a customer, an online trading server,

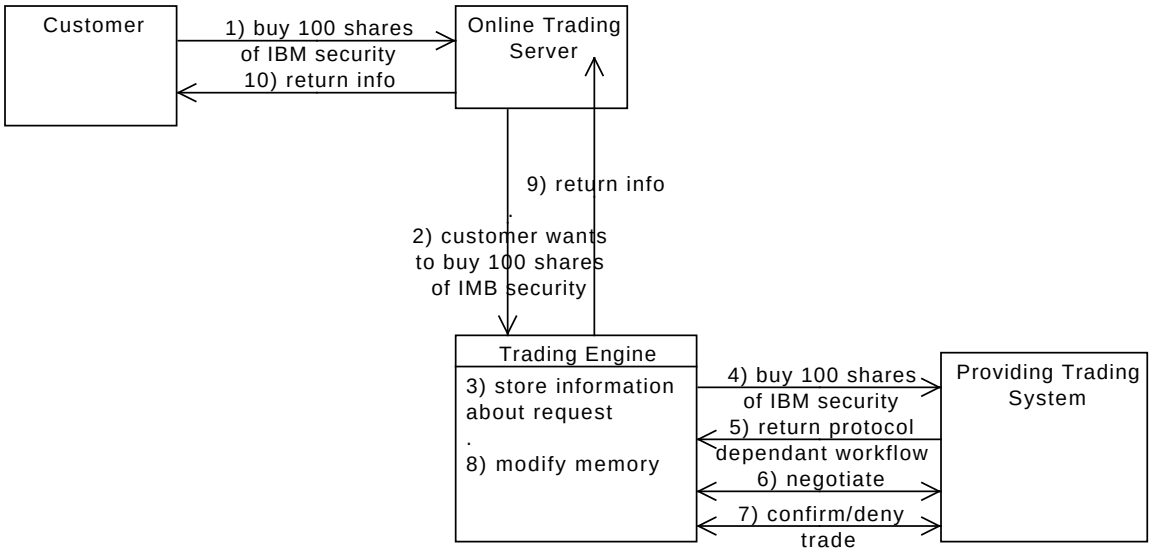

Figure 2.1: An example message flow between customer, online trading server, trading engine and providing trading system

a trading engine, and a providing trading system. The online trading server receives a request from the customer to process a quote or trade request according to a particular protocol. The server sends a request to the trading engine, where the details are stored and the request is forwarded to the providing trading system (PTS). The provider application selects the appropriate protocol-dependent work-flow for the request and responds with a series of messages. The trading engine updates its records with the new state for the quote or trade. The online server then communicates the appropriate confirmations, notices, ACKs and other information back to the customer-trading application or the PTS based on the expected work-flow.

In this scenario, the online-trading server's main task is to pass messages between the customer and the trading engine, and to keep a book of orders. We have made our trading server smarter in order to model the variability between different order protocols. We treat the trading engine and PTS as one entity and call it an online trading system (OTS). The OTS accepts all of the orders and forwards them to the PTS only when they are eligible to be executed, i.e., when they become market orders. We still depend on the trading engine to keep track of whether an order is filled completely and how much there is still left to be filled. This design decision simplifies the OTS, so that it does not have to continually check the current availability of securities and their prices on the market, and it does not match buy and sell orders.

2.1 Terminology

Before we describe the different types of orders, we define the most commonly used terms in trading. The definitions are taken from Finance Library [7], U.S. Securities and Exchange Commission [6], and Wikipedia [4]:

- A **security** is a negotiable instrument that proves ownership of stocks, bonds, or other investments.
- A **share** is the smallest unit of a security that can be bought or sold. For example, IBM is the security, and one can buy 100 shares of IBM security.
- A **ticker** provides information about a particular stock: its last trade price, its last trade time, the change in price, the stock exchange name, the total volume, the last

open price, the last close price, and its **ticker symbol**, which is an abbreviation for shares of a that stock on a particular stock market.

- A **quote** is information about a particular stock that includes the last trade price, the last trade time, the change in price, the stock exchange name, the total volume, the last open price, the last close price, etc.
- **Exchange trading** is the trading of securities via facilities that are constructed for the purpose of trading, for example a stock exchange.
- **Over-the-counter** trading is the trading of securities directly between two parties.

2.2 Types of Orders

An investor has several options when it comes to placing an order to buy or sell securities. He or she can choose for his or hers orders to be executed on condition of the prices of securities, and can choose the time period for which his or hers orders are valid.

- **Market Order** A market order is a basic request to buy or sell some number of shares of some security. When the order is executed, the securities are bought or sold at the current market price. Unless a user specifies otherwise, an order is entered as a market order.

  As long as there are willing buyers and sellers, the user is almost always guaranteed that his or her order will be executed. However, the price that the user pays may not always be the price obtained from a real-time quote service, especially in fast-moving markets where security prices are more volatile. The larger the the number of shares to be bought or sold, the greater the chance that the user will receive different prices for parts of the order. The other types of orders vary depending on price manipulation and time to market.

- **Limit Order** A limit order is usually placed to secure the selling (or purchase) of a security at a higher (or lower) price than the current market price in order to limit losses. A buy-limit order can be executed only at the limit price or lower, and a sell-limit order can be executed only at the limit price or higher.

However, there are some points a customer should keep in mind before placing an order. First, if the security market price never falls (or rises), then a buy-limit (or a sell-limit) order may never be executed. Second, limit orders are executed in the order in which they are received. Therefore, it is possible that the order will not be filled if the price fluctuates above or below the limit before the order rises to the front of the waiting line.

For example, suppose that a security has a current value of $12.00/share on the market. A customer wants to make sure that he or she does not pay more than $10.00/share for the security, therefore the customer enters a limit-buy order at any limit-price up to $10.00. Doing so, the customer avoids buying a security at $20.00/share and suffering immediate losses if the security drops later in the day or weeks ahead.

- **Stop Order** A stop order is placed to protect or ensure profits. A buy-stop order specifies for a security a price above the current market price, and a sell-stop order specifies a price below the current market price. Once the market price of the requested security reaches the stop-price, the order becomes a market order and gets executed. Note that the order might not be executed at the preferred price, since once it turns into market order, the order is filled at the current market price. A stop order is particularly advantageous to an investor who is unable to monitor his or hers securities for a period of time. However, a stop order may be activated by a short-term fluctuation in a security's price.

  For example, let's say a customer purchases 100 shares of XYZ security for $30.00/share. The customer wants to limit a loss, so he or she places a sell-stop order at $28.00/share. If the price falls to $28.00 or below, the stop order is activated and sell-market order for 100 shares of XYZ security is initiated.

  The use of a stop order is common for a security that trades on an exchange market rather than in an over-the-counter market. Sometimes broker-dealers may not allow stop orders on some securities.

- **Stop-Limit Order** A stop-limit order combines the features of a stop order and a limit order. Once the stop-price is reached, the order becomes a limit order at a specified limit-price. Once the limit price is reached, the limit order becomes a market order.

In order to avoid the risk of a stop order being executed in a fast-moving market, the customer can place a buy-stop-limit order. Suppose that the current market price of XYZ security is $20.00. A customer decides to place a buy-stop-limit order with stop price $30.00 and limit price $31.00. If the market price of XYZ passes $30.00, the order will be placed as a buy-limit order. If the market price rises rapidly to $35.00, the limit order will not be executed until it falls which may be never. However, the customer avoids getting a bad fill.

- **Bracket Order** A bracket order combines market, limit and stop orders to help the customer limit losses and guarantee profits by bracketing an order with two opposite side orders. A limit-buy order is bracketed by a high-side sell-limit order and low-side sell-stop order, and a limit-sell order is bracketed by a high-side buy-stop order and low-side buy-limit order.

  The order quantities of the high- and low-side bracket orders match the quantity of the original order. The bracketed orders have a default $1.00 offset from the original price. The offset can be changed for a specific order. Once a price rises or falls and one of the bracket orders is executed, the order on the other side is cancelled.

  For example, suppose a customer submits a buy-bracket order for 100 shares of XYZ security at a limit-price of $83.87/share and offset $1.00. When the price drops below the limit-price, the limit order is executed. Immediately, two new orders are placed. A sell-limit order for $84.87/share and a sell-stop order for $82.87/share, both for 100 shares of XYZ security. If the price of the security on the market falls and the sell-stop order is executed, the high-side sell-limit order is cancelled. If instead the price of the security rises and the sell-limit order is executed, the low-side sell-stop order is cancelled.

- **Market-to-Limit Order** A market-to-limit order is first executed as a market order at the current best market price. If the order is only partially filled, the rest is cancelled and submitted as a limit order at the price with which the already-filled market order was executed.

  For example, a customer wants to buy 500 shares of XYZ security at the current market price. When the order gets filled, only 400 shares are bought at $20.00/share. The rest of the order is cancelled and a new buy-limit order is placed for 100 shares

of XYZ security at $20.00/share. The new order will only be executed if the market price of XYZ security falls under $20.00/share.

- **Limit-on-Close/Open Order** A limit-on-close order will execute at the closing of the market, if the closing price of the security is at or better than the limit-price. Similarly, a limit-on-open order will execute at the opening of the market, if the opening price of the security is at or better than the limit-price. Otherwise the order is cancelled.

- **Immediate-or-Cancel Order** An immediate-or-cancel order is executed as a market order. If it is executed only partially, the rest of the order is immediately cancelled.

- **Good-till-Cancelled/Date/Time Order** An order cannot be cancelled by a customer, unless it is specified as a good-till-cancelled order. A good-till-cancelled order is executed as a market order and stays active until it is filled completely or the customer cancels it. Similarly, a good-till-date/time order is executed as a market order and stays active until it is filled completely or until the specified ending date and time have passed.

- **Fill-or-Kill Order** A fill-or-kill order requires a complete order to be executed immediately as a market order. If the order is not filled completely when it is accepted by the market, the entire order is automatically cancelled.

- **All-or-None Order** An all-or-none order is executed as a market order, but only if a complete order can be filled all at once. The order remains active until a complete order is filled or it is cancelled at the close of the market.

2.2.1 Summary

This chapter describes an example of an online trading system that forms a case study that we are evaluating. In the course of this dissertation, all orders described in the previous section are modelled. The next chapter describes Feature Oriented Modelling Language that is used to model the example of the described online trading system.

Chapter 3

Feature-Oriented Requirements Modelling

The purpose of this dissertation is to evaluate Shaker's language for modelling the requirements of a software system to be developed, with special emphasis given to the features to be supported [12]. Before the reader understands how we approached modelling the online trading system (OTS), he or she should understand Shaker's approach to organizing the requirements of a system under development (SUD). The approach extends already established requirements engineering (RE) approaches, like the Object Oriented Analysis (OOA) methodology for specifying an SUD. Section 3.1 first defines requirements modelling for an SUD. Section 3.2 introduces RE modelling, using Larman's OOA methodology as a representative approach. Section 3.3 describes the Feature-Oriented Software Development (FOSD) paradigm, which extends the UML language, but with respect to features. Section 3.4 describes the basics of the Alloy language , and its notation which are adapted by FORM to describe its notation. Section 3.5 upgrades FOSD with Feature-Oriented Requirements Modelling (FORM) developed by Shaker et. al, and Section 3.6 shows how the notation is used in FORM.

3.1 Requirements

When developing a software system, it is common for software engineers to first create a description of the proposed software system together with clients, because "the description is the clay in which software developers fashion their works" [9]. The most important part of creating the description of the software is distinguishing between the *requirements*, which describe what the software should do, and the *design*, which describes how the software should be built. The requirements description includes the system-under-development (SUD); its *domain* which describes environment in which SUD will run; and use and purpose of the SUD. The purpose of the software should be looked for outside of the software: in the real world, where the effects and benefits of the SUD will be observed, interpreted, assessed and enjoyed. On the other hand, the design describes the internal components, interconnections, data structures and algorithms of the SUD [9].

3.2 Requirements Modelling

There are many languages for expressing requirements [10, 2, 11, 8]. At a high level they all advocate a common collection of views of the requirements. We describe RE-style modelling using Larman's OOA methodology, because it does a throughout job describing the Unified Modelling Language (UML) that is known to most readers [11].

The UML is "a visual language for specifying, constructing, and documenting the artifacts of systems" [11]. In general, it describes diagrams that can be used either to represent concepts in the real world or software elements in an object-oriented design.

The first product of analyzing and designing a software-under-development (SUD) in an OOA manner is the *domain model*, which describes all noteworthy concepts, vocabulary and information content in the SUD's environment. The domain model can be considered a visual representation of the real-world objects and concepts in the domain. In the UML, a domain model is illustrated as a set of class diagrams that show domain objects or conceptual classes and their attributes, and the associations between them. A conceptual class can be an idea, a thing, or an object [11].

To create a domain model, one must identify the conceptual classes, draw them in a UML class diagram, and add associations and attributes. For example, in an online trading

system (OTS), it makes sense to consider objects like a transaction, a user and a providing trading system (see Figure 3.1). One should keep in mind that a domain model is not a data model, which shows persistent data to be stored. Therefore, one should not exclude a class just because there is no need to remember the information, or because a class has no attributes [11].

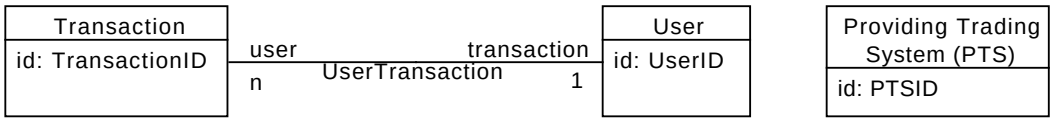

Figure 3.1: A simple UML class diagram of the online-trading system

If a concept carries information that needs to be remembered, the information is represented as *attributes*. An attribute is a "logical data value of an object" [11]. An attribute should not be a complex domain concept. When deciding whether concept data should be expressed as an attribute or a component class, one keeps in mind that attributes are usually primitive data types, e.g. strings or numbers. For example, each of the three classes in the OTS domain model in Figure 3.1 carries information about a class's own unique ID.

If concept data are of a complex type, they are modelled as a component class, and a line connects the compound class to the component class. An *association* is "a relationship between instances of classes that indicates some meaningful and interesting connection" [11]. An association has a name and is represented as a line between the related classes. Each end of an association is called a *role* and includes a name, and a multiplicity expression. For example, in the OTS domain model in Figure 3.1 there is an association between a user and transaction. The association name is `UserTransactions` and role names are `user` and `transaction`, respectively.

The second product of OOA is a behavioural model of the SUD, which defines the SUD's behaviour in terms of externally visible inputs from the environment and the SUD's outputs, which affect the state of the environment. The behavioural model describes the SUD's reaction "to three different events: external events from actors, timer events, and faults or exceptions" [11].

Larman uses *System Sequence Diagrams* (SSD) as the notation to express an SUD's

behaviour. An SUD illustrates related input and output events. An input to the SUD is a *system event*, which is generated by an actor, e.g. `User`, and handled by a *system operation* [11]. An SSD has a graphical representation that shows events created by external actors, their order, and the SUD's corresponding responses.

System operations are modelled using *operation contracts* that specify pre- and post-conditions. Each contract consists of four sections: the name of the operation and its signature; cross references to related operations; the precondition, which lists noteworthy assumptions about the state of the domain before the operation executes; and the post-condition, which describes the state of the domain after the operation is completed [11].

A contract is a way to describe an SUD's functionality in a way that one can focus on *what* the final effects of the operation on the domain are, rather than on *how* the operation is implemented. The effects an SUD can have on the domain state include the creation or deletion of objects, the creation and deletion of associations, and changes to attribute values.

3.3 Feature-Oriented Software Development

Breaking down the SUD into distinguishable characteristics creates a collection of features that can be composed according to needs of a user. From a collection of features, different software systems can be generated that have well-defined commonalities as well as differences. The "set of software systems generated from a set of characteristics" is also called a *software product line* [1].

Feature-Oriented Software Development (FOSD) is "a paradigm for the construction, customization, and synthesis of large-scale software systems" [1]. The central concept of FOSD is a *feature*, which is a "distinguishable characteristic of the software that is important to individuals involved in developing the product" [12, 3]. It can be considered as a unit of functionality of an SUD that corresponds to a distinct set of requirements and may also correspond to one or more design decisions and potential configure options [1].

In FOSD, a family of SUDs is described in terms of their commonalities and variabilities in analysis, design, and implementation. FOSD favours the use of the feature concept in all phases of the software cycle, so that features specified during the analysis phase can be

traced through the design and implementation. These features can be reused in multiple products, and product instances of the product line vary in the features they provide [1].

FOSD focuses on three major research problems [1]:

Feature Modelling: Feature-Oriented Domain Analysis (FODA) was the first approach in this area. It introduced the concept of a feature, and uses features to describe common and variable properties of a software system. It introduced the notion of a *Feature Model* (FM), which describes the relationships and dependencies among features in a product line. However, features are explicit only in analysis and not in later stages of development, e.g., not in design and code.

Feature Interaction: This research focuses mainly on run-time interactions among features. A *feature interaction* is defined as a "situation in which two or more features exhibit unexpected behaviour that does not occur when the features are used in isolation" [1]. Given any two features, they might run independently, i.e., not interact at all, they might be designed to interact, or they might interact unexpectedly.

Feature Implementation: This research makes feature explicit at the source-code level (e.g., features are separated from the base code). This separation allows developers to trace features from the problem space to their realization in the solution space.

The *domain analysis* part of FOSD includes defining features that are part of an SUD or part of a software product line. Developers learn about the scope of the product line, the features it includes, and how features cooperate. The goal is to define common and distinguishable properties in the product line, and to express them with FMs. Some FOSD approaches strive to enrich feature models with additional information about feature cardinalities, constraints, and non-functional properties of features, but they risk making the FM less understandable to users who are not domain experts. The FOSD methodologies do not offer a complete solution for requirements modelling (RM) in terms of features. In particular, they are not complete, detailed, or precise enough to reason about the behaviours of collections of features in a specific product. Therefore, Shaker et. al decided to develop Feature Oriented Requirements Modelling (FORM), which is RE-style requirements modelling combined with FOSD, resulting in a feature-oriented model of the functional requirements of an SUD.

3.4 Alloy

Feature Oriented Requirements Modelling (FORM) uses Alloy notation to describe the domain model and constraints on it. Therefore, we first present Alloy in this section, and its notation in the following subsection (3.4.1). FORM is presented in section 3.5 and FORM's application of the language is described in section 3.6.

Alloy is a *declarative* language. Its *logic* describes a problem in terms of sets of objects (entities), relationships between objects, and constraints on allowable values of object sets and relations. Alloy represents a complex object as a relation, and its properties as *constraints*. An *operation* specifies allowable changes, in terms of a pre- and a post-condition, on each objects and relations [8].

The Alloy *language* offers a flexible type system with subtypes and unions of types. It also supports modular specifications, which enable reuse of generic declarations and constraints in different contexts [8].

Alloy *analysis* supports automatic simulation and checking. Given a logic formula, the analyzer attempts to find model states or executions that satisfy, via an example model state, or violate, via a counterexample model state, the Alloy specification. The search space is defined by the user and is limited by the number of objects of each type, called the *scope*. Defining the scope gives the user control over the examined space, and makes it feasible for the analyzer to exhaustively examine all instances within it. Some examples may require a smaller scope for some entities, and a higher scope for others [8].

Following Dijkstra's dictum that "program testing can be used to show the presence of bugs, but never to show their absence" [8], Alloy assumes a *small scope hypothesis*. The hypothesis states that "all flaws in models can be illustrated by small instances, since they arise from some shape being handled incorrectly, and whether the shape belongs to a small or large instance makes no difference" [8]. The advantage of this hypothesis is that one can use the Alloy analyzer to check properties about an infinite number of model instances on a finite number of small model instances.

3.4.1 Alloy Notation

The following subsection is a summary of the Alloy language notation [8]. The Alloy logic uses *relational logic*, which "combines the quantifiers of first-order logic with the operators of the relational calculus" [8].

An Alloy model is an entity-relationship model, like a UML class diagram, which describes the set of possible states. Each model state represents a specific domain state (DS) (i.e., a particular valuation of object sets and relationships), a pair of states, or an execution trace of states. The Alloy analyzer searches for instances that satisfy the set of given definitions and constraints. For example, Figure 3.2 shows an example of an Alloy trace. In the first state, Ticker1 has a symbol and a price and it is observed by User1 with UserId0. In the second state, User0 becomes an observer of the same Ticker1.

An object structure in an Alloy model is built from *atoms*, which are basic entities, and *relations*, which are relationships between atoms. An atom is a primitive entity that cannot be divided into smaller parts; its properties do not change over time, and it does not have any built-in properties. For example, a user's ID can be represented as an atom of type `UserID`.

Each object in Alloy is declared as a signature, which is a type declaration. For example, a simplified version of the domain model of an OTS presented in Chapter 3, shown in Figure 4.1, has the following set of signatures:

```
sig User {}
sig PTS {}
sig Transaction {}
```

Every entity, message, and relationship in a FORM domain model has its own signature. In order to build objects that are more complex, Alloy uses relations to capture internal structure. A *relation* consists of a set of tuples, where each tuple represents a sequence of atoms. For example, given the following sets:

```
Transaction = {(t0), (t1)}
TransactionID = {(0), (2)}
```

then the following relation, `transaction_id`, between structures `Transaction` and `TransactionID`

```
transaction_id: Transaction -> TransactionID
```

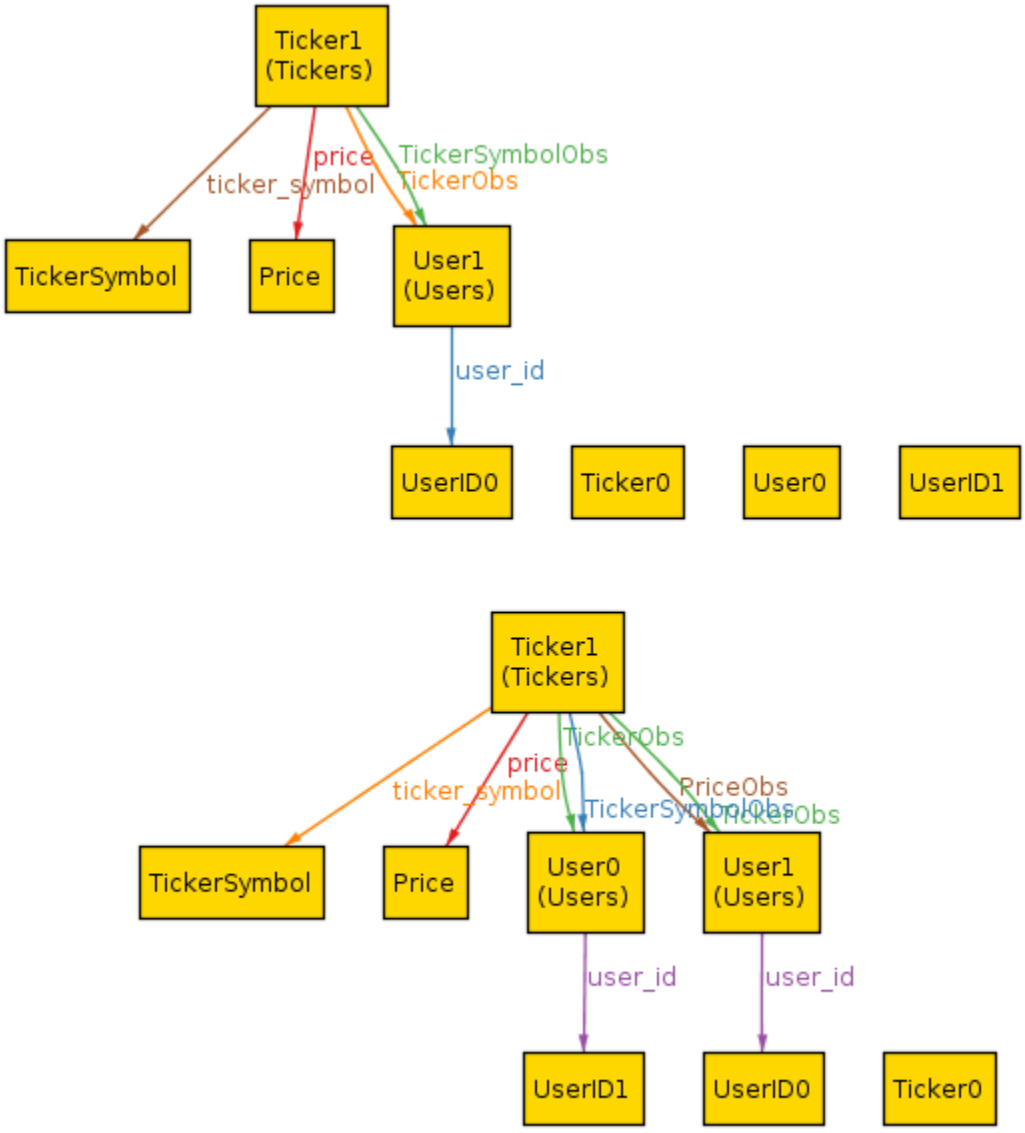

Figure 3.2: An example of an Alloy instance trace

is a type declaration for relationship `transaction_id`, whose value set is the cross product of the domain and the range:

```
transaction_id = {(t0, 0), (t0, 1), (t1, 0), (t1, 1)}
```

A relationship can also be represented as a three-way mapping between concepts. Every complex object is represented as a signature and its attributes are represented as three-way relationships between the object, its attribute, and attribute type. For example, the attribute `transaction_id` of a `Transaction` object represents a three-way mapping between the Domain State (DS) of the OTS, a `Transaction`, and its type `TransactionID`. It has the following signature:

```
sig DS {
  transaction_id: Transaction -> TransactionID
}
```

Alloy offers two different categories of operators to manipulate objects, sets of objects, and relations: *set operators* and *relational operators*. In a set operator, the order of tuples is irrelevant, but their arity and types matter. Whereas in a relational operator, the order of tuples is essential, but their arity does not play an important role. In Alloy, a set is represented as a relation, and a *scalar* is represented as singleton set. This representation allows the any operators, including the relational join operator, to be applied to scalars, sets, and relations.

The following operators are set operators and each can be applied to any pair of relations as long as they have the same arity. Since a scalar is interpreted as a singleton set, no additional braces ({}) are needed when operating over scalars [8]:

+	union
&	intersection
-	difference
=	equality

Alloy provides a constant literal **none** to represent an empty set. Alloy also supports a range of relational operators, but FORM extensively uses only one, so we will omit

descriptions of the others (for descriptions of other relational operators, please refer to Jackson [8]).

The operator **dot join** (**.**) is a relational operator for composing tuples. In order to join two tuples, $s_1 -> ... -> s_m$ and $t_1 -> ... -> t_n$, the last atom of the first tuple, s_m, and the first atom of the second tuple, t_1, must be the same atom. The result is the following tuple, in which the matching atoms, s_m and t_1, are omitted: $s_1 -> ... -> s_{m-1} -> t_2 -> ... -> t_n$. If there are no matching atoms s_m and t_1, the result of the join is empty. The tuples can have any arity, as long as they are not both unary. If tuples represent functions, then the composition is also a function, and the dot operator represents functional composition. For example, let `UserTransactions` be the relationship between users, `User`, and their transactions, `Transaction`, defined in the following way:

```
user: UserTransactions -> User
transaction: UserTransactions -> Transaction
```

Let us further assume that every `Transaction` has an ID, and every `User` has an ID, specified in the following way:

```
transaction_id: Transaction -> TransactionID
user_id: User -> UserID
```

If one wants to find the user for a specific transaction given a transaction ID, `tID`, one applies the dot join operation between `user`, `transaction`, `user_id`, and `transaction_id`. (1) To retrieve a specific transaction from the set of all transactions whose ID is value `tID`, one first applies the dot join operator to `tID` and the relation `transaction_id`:

```
transaction_id.tID: Transaction
```

(2) To retrieve the associated element of `UserTransactions`, one applies the dot join operation to the relation `transaction` and the transaction from step (1):

```
transaction.(transaction_id.tID): UserTransactions
```

(3) To find the user who is associated with the transaction, one applies the dot join operator to the element of `UserTransactions` from step (2) and the relation `user`:

```
(transaction.(transaction_id.tID)).user: User
```

(4) Finally, to find the `user_id` of the user identified in step (3), one applies the dot join operation to the user and the relation `user_id`s:

```
((transaction.(transaction_id.tID)).user).user_id: uID
```

The result is the `user_id`, `uID`, that is the ID of the user who is associated with the transaction whose ID is value `tID`.

FORM uses Alloy syntax in two ways: as an *expression language* over model elements, and as an *action language* for writing assignments to model variables. A relational operator is used to navigate a FORM model and to isolate the objects, object sets, attributes, and relations that are to be evaluated and manipulated. For example, when an OTS is processing a limit order, it constantly receives updates for ticker prices. When it receives an update, it has to check if the new price has reached the required price. Suppose that the FORM feature machine that processes the order `self` stores information about the required price in a local variable `LOBFMLimitPrice` in the behavioural state (`bs`). `LOBFMLimitPrice` is a relation between the `LOBFM` feature machine and the `Price` stored in the behavioural state, `bs`:

```
LOBFMLimitPrice: LOBFM -> Price
```

In order to access the variable value, `Price`, one applies the dot join operator to the behavioural state (BS) and the variable. FORM defines the literal `self`, which is a pointer to the feature machine's own instance, and `bs` which is a complex Alloy structure that represents the behavioural state of the behavioural model. To check whether the newly received `price` has reached the limit price, `LOBFMLimitPrice`, one applies less-or-equals operator between variables:

```
self.bs.LOBFMLimitPrice <= price
```

FORM also uses relational operators for *expressing constraints* on model elements in order to isolate objects, sets of objects, attributes, or relationships that are being constrained. For example, to restrict the number of `user_id` attributes that a `User` entity can have, our FORM model specifies the following cardinality constraint:

```
all user: Users | #user.user_id = 1
```

This constraint includes a *quantifier*, stating that every (`all`) object in the set `Users` must have only one ID. Alloy logic supports the following quantifiers for expressing constraints:

- **all** x: e | F formula F holds for every x in e;

- **some** x: e | F formula F holds for some x in e;
- **no** x: e | F formula F holds for no x in e;
- **lone** x: e | F formula F holds for at most one x in e;
- **one** x: e | F formula F holds for exactly one x in e.

As mentioned earlier in this section, Alloy also offers analysis, which can be used to either find examples that satisfy or counterexamples that do not satisfy given assertions. An assertion is expressed as an expected constraint on allowable model instances. For example, the following set of assertions specifies constraints on the numbers of the roles `status` and `transaction` in the relationship `TransactionStatus`:

```
all transaction_status: TransactionStatuss | #transaction_Status.status = 1
all transaction_status: TransactionStatuss | #transaction_Status.transaction = 1
```

Quantifiers can be applied to Alloy expressions as well:

- **some** e e has some tuples;
- **no** e e has no tuples;
- **lone** e e has at most one tuple;
- **one** e e has exactly one tuple.

For example, `user_id` relates each `UserID` to exactly one `User`, which means that no two users can have the same id. This constraint is specified as part of the definition of a `user_id`:

```
user_id: User one -> UserID
```

Constraints are used to express *facts* and *assertions*. Facts are constraints that are assumed to always hold. They can be written as *signature facts*, which means that they apply to every member of a signature; such facts are expressed in the second ellipsis of the signature. For example, a FORM domain model always model always has a set of signature facts. Let us assume that a domain model has users each of whom has a unique ID. In the Alloy declaration of the domain state (DS) the connection between a user and his or her

ID is given in the first ellipsis, and the constraint on the number of IDs each user can have is given in the second ellipsis:

```
sig User {}
sig UserID {}

sig DS {
  user_id: User -> UserID
}{
  all user: Users | #user.user_id = 1
}
```

Assertions are also intended to be valid in a model instances, but the analyzer does not assume that they always hold. The analyzer checks the assertions against the rest of the model. If it is possible for a model instance to violate an assertion, the analyzer returns a counterexample. For example, no two users in an OTS are allowed to have the same ID. In order to check if the constraint is satisfied, one declares an assertion which states that for all instances of the domain state DS, and for all users in each instance of the DS, `user_id` maps each user `u` to exactly one ID:

```
assert unique_user_id {
   all ds: DS | all u: ds.Users | #u.(ds.user_id) = 1
}
```

Alloy is a declarative language, so operations are expressed by *specifying the state of objects before and after the operation*. For example, in the OTS model, we assume that the set of `Tickers` stays the same throughout the execution. In order to express this constraint, two different names for two different domain states are needed: `ds`, referring to the domain state before the operation, and `ds'`, referring to the domain state after the operation. We can then write expressions and constraints on the values of model elements in the two domain states:

```
pred DSexec(ds: DS, ds': DS) {
  ds.Tickers = ds'.Tickers
}
```

All expressions that can be reused are packaged as *functions* in Alloy. For example, the enhanced feature machine for limit order (shown in Figure 3.6) uses functions

`CreateTransaction` and `ForwardOrder`. The function `CreateFunction` is defined as a predicate. A predicate in Alloy has the same functionality as a function but it evaluates to true or false, whereas functions evaluate to a set. The complete Alloy function with signature is given in Figure 3.3. (1) A function adds a new limit-order object to the set of `Orders`. The object's attribute values are assigned by adding mappings (`object -> new value`) to the attributes' relations, `time_stamp` and `status_volume` (see lines 6-8 in Figure 3.3). (2) The function adds a `Transaction` object with new links to its attributes (see lines 10-15 in Figure 3.3). (3) At the end, the function adds the new relationships `transaction_status` and `user_transactions` and their associated roles: one role for each entity involved in the relationship (see lines 17-23 in Figure 3.3).

3.5 FORM

Shaker et al.'s Feature-Oriented Requirements Modelling (FORM) methodology combines Requirements Engineering (RE) style requirements modelling with FOSD resulting in a feature-oriented model of the functional requirements of an SUD. It takes into account how detailed software requirements should be modelled, and further structures those models with respect to features. The goals of the methodology are *modularity*, such that features are modelled as independent modules; *modifiability*, which allows new features to be added with minimal changes to the existing features; and *precision*, which enables automatic checking of properties like correctness, consistency, and non-determinism. This section is a summary of their work [12].

The methodology uses the notion of requirements as described by Zave and Jackson [9, 13]. The *domain* of a system under development (SUD) is the "environment or context in which the SUD will operate" [12]. The *requirements* of an SUD "are desired effects of the SUD on its domain" [12]. Requirements are expressed solely in terms of observations and manipulations of the domain phenomena — that is, in terms of what the system does. System phenomena — that is, descriptions of how the system works, such as the internal components, communications, and system interface phenomena, like sensors and actuators and their exact inputs and outputs — are not considered in this methodology.

The methodology structures requirements into three models: a *domain model*, a *behavioural model*, and a *functional model*. This decomposition takes into account already

```
pred CreateTransaction[ ds, ds': DS, t: Transaction, order: Ordered,
                        transaction_status: TransactionStatus,
                        user_transactions: UserTransactions, ttype: TransactionType,
                        taction: TransactionAction, tticker_symbol: TickerSymbol,
                        tvolume: Volume, tuser: User ] {
   ds'.Ordereds = ds.Ordereds + order
   ds'.time_stamp = ds.time_stamp + order->currTime
   ds'.status_volume = ds.status_volume + order->tvolume

   ds'.Transactions = ds.Transactions + t
   ds'.type = ds.type + t->ttype
   ds'.action = ds.action + t->taction
   ds'.ticker_symbol = ds.ticker_symbol + t->tticker_symbol
   ds'.volume = ds.volume + t->tvolume
   ds'.phase = ds.phase + t->initiated

   ds'.TransactionStatuss = ds.TransactionStatuss + transaction_status
   ds'.status = ds.status + transaction_status->order
   ds'.status_transaction = ds.status_transaction + transaction_status->t

   ds'.UserTransactionss = ds.UserTransactionss + user_transactions
   ds'.user = ds.user + user_transactions->tuser
   ds'.transaction_user = ds.transaction_user + user_transactions->t
}
```

Figure 3.3: An example of an Alloy function CreateTransaction

established views of requirements modelling and further structures the views according to features. Each of the models is described in detail in the following subsections, 3.5.1, 3.5.2, and 3.5.3, respectively. Small examples will be drawn from the online trading system (OTS) example described in Chapter 2. Chapter 4 presents a more complete description of our application of FORM to the OTS case study.

The methodology first considers each feature in isolation, and then offers methods to address feature interactions to create a complete working product. The methodology assumes there is one or more base features in the SUD that can be enhanced by one or more enhancement machines which in turn can be again enhanced. If a feature is designed to override the behaviour of another feature, we say this is a planned interaction, and the new feature is modelled as an enhancement feature. Other features that add functionality to the SUD without interacting with the base system may be additive features that operate in isolation. Such features may have unplanned interactions, such as conflicting actions or inconsistent assumptions about the domain or the base feature. Unplanned interactions are resolved with precedence rules.

3.5.1 Domain Model

A *domain model* is a structured description of the domain of an SUD expressed in a notation similar to a UML class diagram. For example, Figure 3.4 shows a simple domain model of an online trading system (OTS). A domain model consists of a set of uniquely identifiable *objects* that can have *attributes*, a set of *relationships* that have *roles*, a set of *constraints*, and an *initialization predicate*. An object can either be an entity (e.g. `User`), a relationship (e.g. `UserTransactions`), or a message (e.g. `RequestOrder`).

An *entity* is a persistent object that can be either *active* (e.g. influence other objects) or *passive*. Our OTS example has two actor types: `PTS` and `User`. Passive objects can be controlled by the SUD, the domain or both. For example, a `Ticker` can be controlled by the OTS or the domain, while a `Transaction` is a record produced by the OTS and is purely controlled by the SUD.

A *relationship* is a UML association between other objects in the domain. Each relationship end point represents the *role* that that object plays in the relationship (note, an n-ary relationship has n roles). For example, our simple OTS example has the follow-

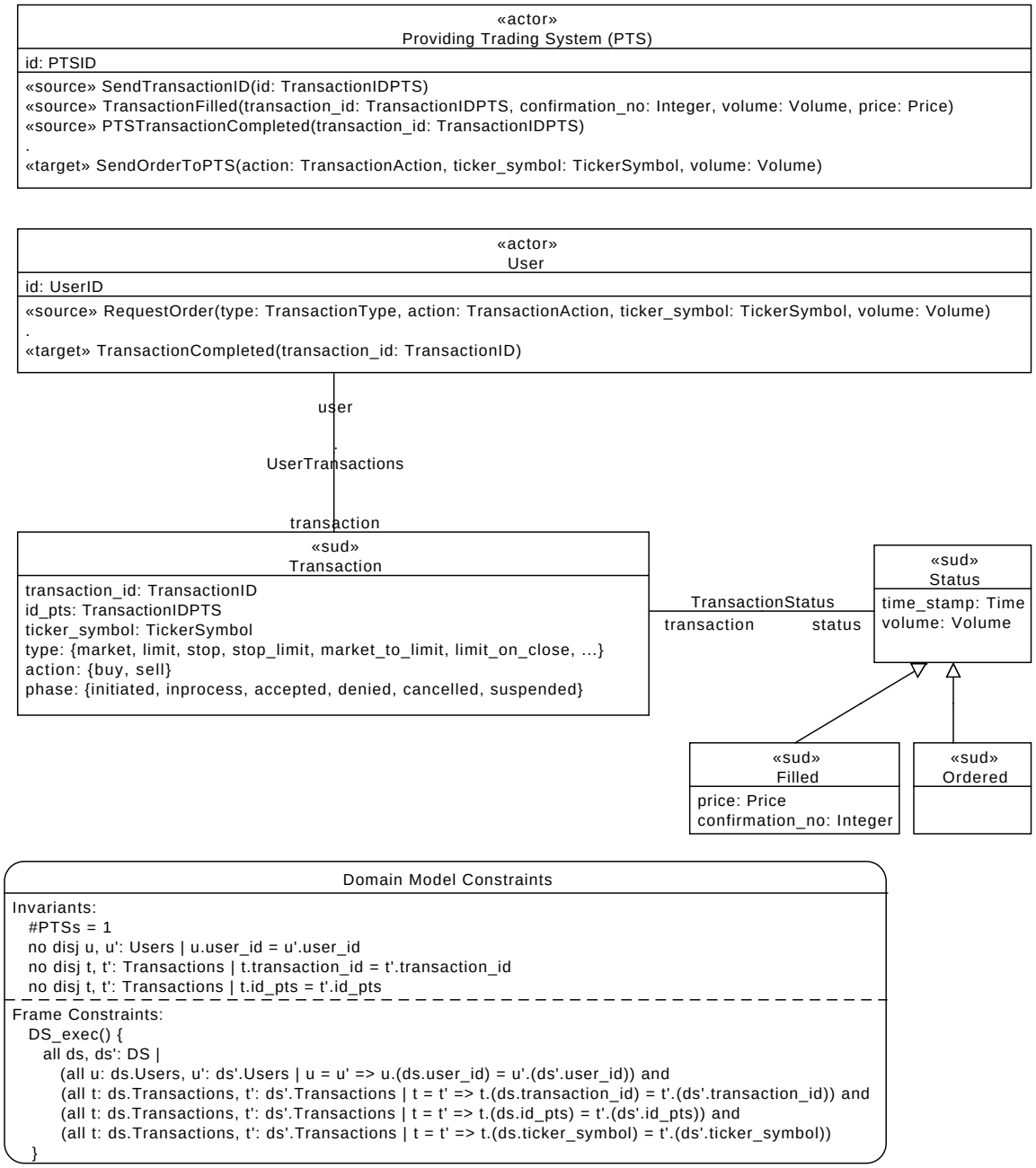

Figure 3.4: A simple FORM domain model of an online trading system

ing relationships: `UserTransaction`, which connects `User` and `Transaction` objects, and `TransactionStatus`, which connects `Transaction` and `Status` objects.

A *message* is an object that represents an interaction between the SUD and an actor in the domain. A message can have one or more *parameters* with corresponding types, and an *end point* which is either the source or target of the message. For example, our OTS example has several messages that are sent and received by `User`s and the `PTS`. A `User` initiates a transaction by sending a `RequestOrder` message to the OTS. This message contains additional information in the form of parameters: the type of transaction (`type`), an indication of whether the request is to buy or sell shares (`action`), the ticker symbol of the share (`ticker_symbol`), and the number of shares to be traded (`volume`).

A domain model describes the set of allowable *domain states*. A domain state represents an "instance of the domain model with particular set of objects with a particular attribute values" [12]. A domain state is *transient* if it includes messages, and *persistent* otherwise. The domain model can be augmented with constraints that place additional restrictions on what constitutes an allowable domain state. Specially, domain-model constraints restrict the values and instances of domain entities and relationships and are expressed in the Alloy modelling language. For example, the constraint that every user has a unique id in the domain model of an OTS is expressed in the following way:

```
no disj u, u': Users | u.user_id = u'.user_id
```

Constraints on the domain that are imposed by the SUD are specified in the behavioural model.

As is described in the next section, a FORM behavioural model monitors and modifies the domain model of an SUD. A *domain change event* (DCE) indicates a change to the domain state of the domain model that may trigger the behavioural model to impose a set of *domain change actions* (DCAs). For example, a user transmits a request for a trade, via a `RequestOrder` message. The resulting `RequestOrder+` DCE triggers a set of DCAs that creates a transaction object, `+CreateTransaction`, and sends a request for trading to the ProvidingTradingSystem (PTS), `+SendOrderToPTS`.

FORM makes use of UML stereotypes to extend the notation for describing the domain model of an SUD. It uses stereotypes to define whether a concept in the SUD is an actor (`<<actor>>`), is domain controlled (`<<domain>>`), or is SUD controlled (`<<sud>>`). It

also uses stereotypes to indicate whether an actor is a target, `<<target>>`, or a source, `<<source>>`, of a message (see Figure 4.1).

3.5.2 Behavioural Model

A *behavioural model* consists of a set of feature machines (FMs), one for each feature. Every FM describes the requirements of a feature, including its name, parameters, and constraints. The feature is modelled as a state machine with a set of states and a set of transitions. Each FM monitors the domain state and issues actions that change it. The FM of different features run concurrently and use precedence rules to solve feature interactions.

A feature that is designed to interact with another feature f is expressed as an *enhancement feature machine* (EFM) that overrides feature f. In this case, f is the base feature but f may itself be an enhancement feature over another feature. For example, suppose a customer decides to trade some stocks. If the customer does not have any special requests, the trade request is transmitted as a market order. If the customer wants to limit losses, he or she can request a limit order. We model a limit order as an enhancement of a basic market order.

The rest of this section describes general FMs and then describes how new features that enhance existing features are modelled as EFMs.

General Feature Machine

A *feature machine* (FM) is a hierarchical state machine that is constructed from a set of *local variables*, a set of *control states*, a set of *transitions* between them, and a *constructor*. Transitions in a FM represent changes in the behaviour of the modelled SUD. Each transition is labelled with the event that triggers the transition, a guard condition that limits the situations in which the transition fires, and a set of actions that are performed whenever the transition fires. A state can either be *basic* with no internal components, or *composite*, if it is composed of several component states. The set of control states that an FM is currently in is the FM's *state configuration*. A *valid* state configuration contains one basic state and all of its ancestor states.

A *local variable* has a name and a type, which may be a concept in the domain model, and stores information that is needed for the duration of the FM. A transition label may

refer to objects, attributes, states, or variables in the domain or behavioural model. FORM introduces *macros* to help simplify transition labels and make the model easier to read. A macro is defined with a `let` key word and is referenced with a letter $. For example, in order to reference the user in MOFM who initiated the order, the machine MOFM, `self`, needs to look for local variable `MOFMUser` in the behavioural state, `bs`. Since the notation to access the local variable is a bit cumbersome, a macro `$user` is defined:

```
let user = self.(bs.MOFMUser)
```

Each transition can either be a *constructing*, *regular*, *overriding*, or *terminating* transition. A *regular transition* has a source state, a label, a priority, and a destination state. A *constructing transition* fires when the FM is initialized by its constructor. The source state of a constructing transition is the machine's initial pseudo-state (depicted as a black circle). A *terminating transition* has a destination state that is depicted as an X to indicate the end of the machine's execution. *Overriding transitions* only appear in enhancement feature machines (EFM); they are described in the section on EFMs.

For example, Figure 3.5 shows the FM for market orders. It has two states: `MOWaitingForAck` and `MOWaitingForExecution`. The market order FM keeps local variables that refer to the market-order transaction (`MOFMT`) and the requesting user (`MOFMUser`). The local variables in the behavioural model are used to store information that the FM needs to refer to during execution. Local variables persist only for the duration of the FM or can be aliases to elements in the domain model. The variable `MOFMT` is an alias for a transaction stored in the set of transactions in the domain model of the OTS.

The FM for the market-order has four transitions: the constructing transition `t1`, which fires when the FM is invoked; transition `t2`, in which the transaction ID is received from the PTS; transition `t3`, in which the PTS sends information about partial transactions that have been filled; and transition `t4`, which terminates the FM when the transaction is completed.

A *transition label* consists of a *trigger*, which is a domain change event: a *guard*, which is an Alloy constraint over domain elements and local variables, and a set of *actions*. Looking at the transitions more closely: Once an order is sent to the PTS (`t1`), and the transaction ID has been assigned (`t2`), the OTS waits in state `MOWAitingForExecution` for messages from the PTS. Whenever the order is partially filled, the PTS sends a message `TransactionFilled` to the OTS with information about the filled transaction. This

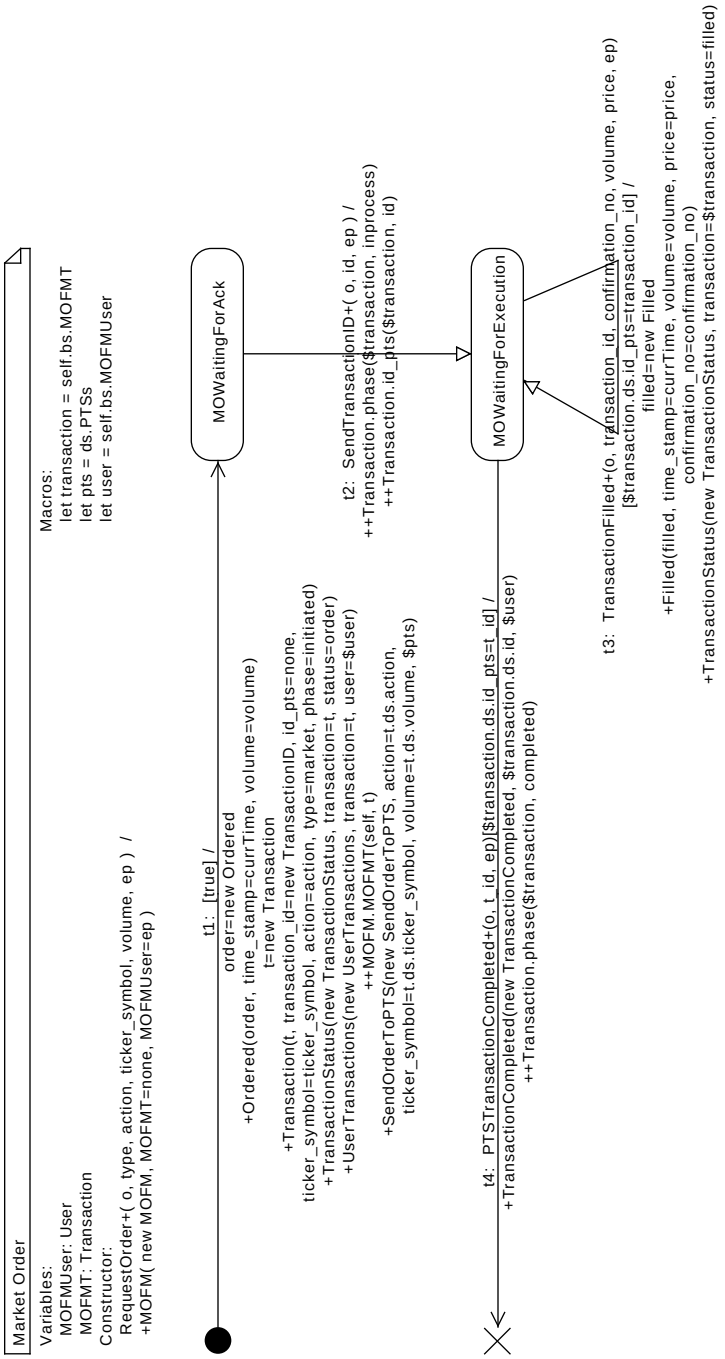

Figure 3.5: A FORM feature machine for a market order

transition has a guard that ensures that the transition fires only if the transaction ID in the message matches the machine's market order, `MOFMT`. When a matching message is received, the OTS performs two events: (1) it creates a new object, a `Filled` transaction, that records information about the completed trade:

```
filled = new Filled
+Filled( filled, time_stamp, volume, price, confirmation_no )
```

(2) It creates a new relationship between the matching market order `MOFMT` and the new `filled` object. This action makes use of a *macro* to more easily refer to the FM's associated market order object in the domain model.

```
+TransactionStatus( new TransactionStatus, transaction=$transaction,
                    status=filled )
```

Lastly, every FM has a *constructor*. When an FM is invoked, its constructor is called. The constructor contains a trigger and a guard, and generates a fresh instance of the FM. For example, an instance of an market-order FM in Figure 3.5 is constructed whenever a `User` sends a message `RequestOrder` of any `type`.

```
RequestOrder+( type, MOFMAction, MOFMTickerSymbol, MOFMVolume )
```

The constructor's actions create a new instance of the FM, `MOFM`, and initialize the local variables. For example, the user variable is linked to the user who requested the order (i.e., the end point of the message):

```
+MOFM( new MOFM, MOFMT=none, MOFMUser=ep )
```

The FM's associated transaction, `MOFMT`, is initially set to `none` — the value is changed by the set of actions performed by the constructing transition. Upon instantiation of the FM, the constructing transition fires. In that transition, (1) a new `transaction` object is created and linked to the local variable:

```
t=new Transaction
+Transaction( t, transaction_id=new TransactionID, id_pts=none,
              ticker_symbol=MOFMTickerSymbol, action=MOFMAction, type=market,
              phase=initiated )
+MOFM.MOFMT( self, t )
```

(2) The newly created transaction is linked to the user:

```
+UserTransactions( new UserTransactions, transaction=t, user=$user )
```

(3) A new `status` relationship is created and linked to the newly created transaction:

```
order=new Order
+Order( order, time_stamp=currTime, volume=MOFMVolume )
+TransactionStatus( new TransactionStatus, transaction=t, status=order )
```

(4) A message is sent to the PTS containing information about the order:

```
+SendOrderToPTS( new SendOrderToPTS, action=t.ds.action,
                 ticker_symbol=t.ds.ticker_symbol, volume=t.ds.volume,
                 $pts )
```

Enhancement Feature Machine

A feature that is designed to interact with another feature is called an *enhancement feature* and is modelled in an *enhancement feature machine* (EFM). It is designed to add new behaviour (i.e., new states, transitions, guards, and actions) to the behaviour of another feature, modelled either as a general feature machine or an EFM of yet another feature. An EFM executes concurrently with its base feature's FM, and can monitor and control actions of the base machine [12]:

- an EFM transition guard can refer to the execution state of its base FM,
- an EFM can asset an invariant (i.e., a constraint) over the state configuration of its base FM,
- an EFM can change the execution state of the base FM, and
- an EFM transition can override a transition in the base FM. In this case, the EFM transition is labelled with an `<<override query>>` stereotype, which means that, if enabled, the EFM transition takes precedence over and prohibits the concurrent execution of any enabled transition of the base FM.

For example, suppose that a user submits a specific kind of trade order that is to be executed only if the market price of a security reaches a certain price. In that case, the

order is transmitted as a limit order. Since a limit order transforms into a market order as soon as the market price reaches the limit price, a limit-order FM can be modelled as an EFM shown in Figure 3.6, whose base feature is the market-order FM (shown in Figure 3.5).

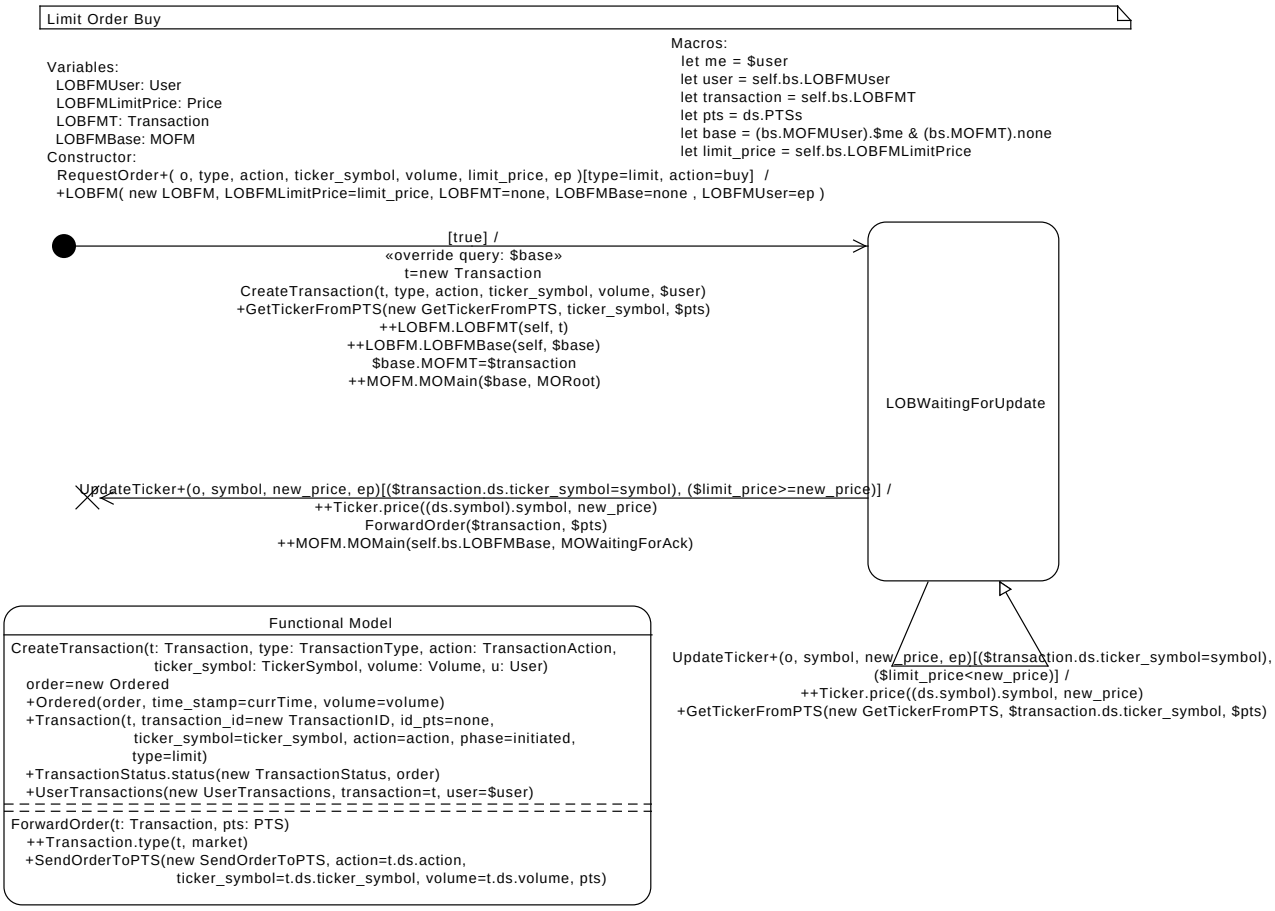

Figure 3.6: A FORM enhancement feature machine for a limit order

A new instance of a limit order FM is created whenever a `User` sends a message `RequestOrder` of `type` limit. Since a buy-limit order differs from a sell-limit order, two different FMs are designed. A buy-limit order FM (from now on addressed as limit-order-buy (LOB) FM) is invoked by the following constructor that has a guard over the `type` and `action`:

```
RequestOrder( type, action, ticker_symbol, volume, limit_price ) [type=limit, action=buy]
```

As a result, a new instance of the LOB FM is created and all local variables are initialized. Note, this FM carries two local variables: `LOBFMLimitPrice`, which stores the user-specified limit price, and `LOBFMBase`, which is a reference to the base FM:

```
+LOBFM( new LOBFM, LOBFMLimitPrice=limit_price, LOBFMT=none, LOBFMBase=none,
        LOBFMUser=ep )
```

Recall that this `RequestOrder` will also activate a new instance of a market-order FM, because the constructor of that FM reacts to `RequestOrder` messages of all types. The constructor of the limit-order EFM overrides the constructing transition of the base machine to create a limit-order `Transaction` in the domain rather than a market-order `Transaction`.

```
<<override query: ( $base )>>
order=new Ordered
+Ordered( order, time_stamp=currTime, volume=volume )
t=new Transaction
+Transaction( t, transaction_id=new TransactionID, id_pts=none,
              ticker_symbol=ticker_symbol, action=action, phase=initiated,
              type=limit )
+TransactionStatus.status( new TransactionStatus, order )
+UserTransactions( new UserTransactions, transaction=t, user=$user )
++LOBFM.LOBFMT( self, t )
```

The new transaction is also linked to the base machine's transaction:

```
$base.MOFMT=$transaction
```

However, the order is not sent to the PTS, until the market price is at or below user's specified limit price. Instead, a request for information about the security is sent to the PTS:

```
+GetTickerFromPTS( new GetTickerFromPTS, ticker_symbol, $pts )
```

An override query can change the state of the base machine. If the EFM is to execute in parallel with the base machine, then the base machine is put into some valid state configuration. If the base machine is to be suspended, then the base machine's execution state is set to the most appropriate composite state. In that case, the base machine can execute only transitions whose source state is the composite state or any ancestor state; but it cannot respond to any event that triggers a transition from a descendant state. In

this example, the only composite state in the base feature MOFM is the root state, `MORoot`. Therefore, MOFM is sent to the root state with the following command:

```
++MOFM.MOMain( $base, MORoot )
```

When a user transmits a trade request, both FMs, base and enhancement FM, are constructed. In order for the EFM to monitor and control actions of the base FM, it needs to have a reference to it. This reference is established by finding the base FM that is active and that has the same user as the EFM in the behavioural state `bs`:

```
(bs.MOFMUser).LOBFMUser
```

Since a user can have more than one transaction, this constraint can return more than one base FM. When the system initially receives a trade request, both newly instantiated machines have same user and both have an empty transaction. Therefore, we specify a second constraint that returns all base FMs that have an empty transaction in the behavioural state `bs`:

```
(bs.MOFMT).none
```

We intersect both constraints to reference the base machine that has the same user as `LOBFMUser` and has an empty transaction. Thus, `LOBFM`'s reference to the base FM in LOBFM is set in the following way:

```
++LOBFM.LOBFMBase( self, (bs.MOFMUser).LOBFMUser & (bs.MOFMT).none )
```

After construction, the LOB FM waits for updates about the security's market price. If the price reaches the specified limit price, the LOB FM is terminated and the market-order base feature resumes execution in state `MOWaitingForAck`:

```
UpdateTicker+( symbol, new_price )[ $transaction.ds.ticker_symbol=symbol,
                                    LOBFMLimitPrice>=limit_price ]
++MOFM.MOMain( $base, MOWaitingForAck )
```

In addition, the LOB FM's terminating transaction updates the ticker price in the domain with the new price:

```
++Ticker.price( (ds.symbol).symbol, new_price )
```

The type of the transaction order changes to a market order:

```
++Transaction.type( $transaction, market )
```

Finally, the market-order with corresponding information about the transaction is sent to the PTS:

```
+SendOrderToPTS( new SendOrderToPTS, action, ticker_symbol, volume, $pts )
```

All FMs, general and enhancement, must follow certain well-formedness rules that forbid two actions in the same step to conflict with each other. For example, if one action adds an object or relationship to the domain state, the other cannot remove it in the same step. Also, two actions can also not set in the same transition different values to the same attribute of an object, the same local variable or the state configuration of the base FM. In addition, FMs must be deterministic. If more than one transition is enabled simultaneously, one must have a higher priority than the other.

3.5.3 Functional Model

Behavioural models can become complicated overtime when some transitions require long sequences of actions. Moreover, some actions are performed repeatedly by multiple transitions. Thus, FORM includes a *functional model*, which is a "collection of functions that describe named groupings of changes to the domain state" [12]. The functional model can be used to simplify transition labels in behavioural models and improve modifiability, since a function is written once but can be reused multiple times.

A function has a name, a list of parameters, a precondition, and a set of atomic domain changes. For example, when a customer submits an order to the OTS, a new `Transaction` object is created. When a new transaction is created, two new relationships are also always created: `TransactionStatus` and `User` object. Since this is a common step in most of FMs, we create a function `CreateTransaction`, that takes a reference to a `Transaction` object, information to initialize the transaction (type, action, ticker_symbol, volume) and a reference to `User` object:

```
CreateTransaction( t: Transaction, type: TransactionType, action: TransactionAction,
                   ticker_symbol: TickerSymbol, volume: Volume, u: User )
```

This function does not have any preconditions, so the function looks like the following:

```
CreateTransaction( t: Transaction, ttype: TransactionType, taction: TransactionAction,
                   tticker_symbol: TickerSymbol, tvolume: Volume, u: User )
   Pre-condition: [true]
   Post-conditions:
      order=new Ordered
      +Ordered( order, time_stamp=currTime, volume=tvolume )
      +Transaction( t, transaction_id=new TransactionID, id_pts=none,
                    ticker_symbol=tticker_symbol, action=taction, type=ttype,
                    phase=initiated )
      +TransactionStatus( new TransactionStaus, transaction=t, status=order )
      +UserTransactions( new UserTransactions, transaction=t, user=u )
```

3.6 Feature Oriented Requirements Modelling in Alloy

Section 3.4 introduces Alloy syntax using examples from the online trading system (OTS) model. This section describes how to systematically represent a Feature Oriented Requirements Modelling (FORM) domain model in Alloy. Every domain model (DM) is an Alloy model that is composed of signatures for all types defined in the domain model (feature models, concepts, messages, relationships, ...).

Recall that an Alloy model represents a collection of model *instances*. Each instance represents a distinct model state, a pair of states, or an execution trace. Similarly, a *domain state* (DS) of a FORM domain model is an instance of the domain-model space, representing a particular valuation of sets of objects and relationships. When running the Alloy analyzer on a domain model, it returns all valid DSs that adhere to the model's definitions and constraints.

The rest of this section describes guidelines for constructing an Alloy model for a FORM DM. `sig DS{...}{...}` represents the DS space. The first list of definitions are elements of the DS that appear in the first ellipsis of the definition of the DS:

1. Every concept in the DM has its own signature that is a type definition, and a set of objects of that type in the DS. For example, concept `User` has corresponding type definition:

```
sig User {}
```

and set of `User` objects in the `DS { }`:

```
Users: set User
```

2. Every attribute is defined as a relation in the DS that maps the parent concept to the attribute type. For example, the attributes of the `Transaction` concept are defined as follows:

```
transaction_id: Transaction -> TransactionID
id_pts: Transaction -> TransactionIDPTS
ticker_symbol: Transaction -> TickerSymbol
action: Transaction -> TransactionAction
phase: Transaction -> TransactionPhase
type: Transaction -> TransactionType
status: Transaction -> TransactionStatus
```

3. Similarly, every role is defined as a relation between its association and the role type (i.e., concepts). Similarly, every message parameter is defined as a relation between its message and the parameter type, and every message end point is defined as a relation that maps the message to the active entity that sends or receives the message. For example, `GetTickerInfo` has the following definition, which relates the message to exactly one `User`:

```
ep_GTI: GetTickerInfo -> one User
```

4. Every SUD-controlled concept has an associated relation in the DS that maps the concept to all of the DM actors that can *observe* the object. There is a similar relation for each of the SUD-controlled concept's attributes, roles, and parameters. For example, `Transaction` is a passive concept in the OTS example. It can only be observed by the OTS, the user who is involved in the transaction, and the PTS. This constraint is expressed as the following Alloy type definition:

```
TransactionObs: Transaction -> (User + PTS)
```

Since all of a transaction's attributes are also observable by the user and the PTS, all of `Transaction`'s attributes have similar relations:

```
transaction_id_obs: Transaction -> (User + PTS)
id_pts_obs: Transaction -> (User + PTS)
ticker_symbol_obs: Transaction -> (User + PTS)
action_obs: Transaction -> (User + PTS)
phase_obs: Transaction -> (User + PTS)
type_obs: Transaction -> (User + PTS)
status_obs: Transaction -> (User + PTS)
```

As mentioned earlier, a FORM domain model comes with a set of constraints that represent well-formedness conditions on the model and domain constraints on the requirements. These constraints are listed in the second ellipsis of the DS signature.

1. Every DS lists domain-specific constraints. For example, the OTS has a constraint on the number of PTSs:

   ```
   #PTSs=1
   ```

2. For each role of each relationship (and each parameter of message) whose type is a concept, there is a constraint that makes sure that an instance of that role (or attribute) is given in the DS if an only if the associated link (or concept) is also in the DS. For example, the `UserTransactions` relationship has two roles, `user` and `transaction`. `user`'s type is concept `User`. Therefore, if the `user` is in the DS, then it has a corresponding member in the set of `Users`. The following constraint expresses these relationships:

   ```
   (UserTransactions.user in Users) and (user.User in UserTransactionss)
   ```

3. For each end-point of a message, there is a constraint that makes sure that an end-point is in the DS if and only if the message and the end-point concept are also in the DS. For example, a relationship `ep_GTI` is in the DS, if and only if its corresponding message, `GetTickerInfo`, and its end-point, `User`, are in the DS:

   ```
   (GetTickerInfo.ep_GTI in Users) and (ep_GTI.User in GetTickerInfos)
   ```

4. The same is true for all SUD-controlled concepts. For example, a relation that connects a concept `Transaction` to its observer, `TransactionObs`, is in the DS, if

and only if both the `Transaction` object and the corresponding observing active entities, `PTS + User`, are in the DS:

```
(Transaction.TransactionObs in (PTSs + Users)) and
  ((PTS + User).TransactionObs in Transactions)
```

5. Each passive object has a constraint that states that if the passive object cannot be observed by some active entity, then neither can its attributes, roles, and parameters. For example, `Transaction` together with its attributes are only observable by its user, the OTS, and the PTS. The following operation uses state subset notation if an entity can observe an attribute of a `Transaction`, it must be able to observe the `Transaction` itself:

```
all transaction: Transactions |
    { transaction.TransactionIDObs in transaction.TransactionObs
      transaction.IDPTSObs in transaction.TransactionObs
      transaction.TickerSymbolObs in transaction.TransactionObs
      transaction.TypeObs in transaction.TransactionObs
      transaction.ActionObs in transaction.TransactionObs
      transaction.PhaseObs in transaction.TransactionObs }
```

6. For every attribute, parameter, or role instance in the DS, the corresponding container instance (object, message, association, respectively) is also in the DS. For example, object `User` has attribute `user_id`. The model ensures that if an instance of `user_id` is in the DS, then its object `User` is in the DS, as well:

```
user_id.UserID in Users
```

7. Every attribute, role, or parameter has a constraint about its cardinality. For example, the relationship `TransactionStatus` has two roles, each having cardinality 1:

```
all transaction_status: TransactionStatuss | #transaction_Status.status = 1
all transaction_status: TransactionStatuss | #transaction_Status.transaction = 1
```

8. Every message has a constraint that links it to at most one active entity. For example, each `GetTickerInfo` message has at most one (`User`) sender:

```
#ep_GTI.GetTickerInfos =< 1
```

9. Finally, no two elements of the same relationship can relate the same objects in the DS. For example, there exists no two distinct (`no disj`) elements of the `TransactionStatus` relationship that would relate the same instances of roles, `status` and `transaction`:

```
no disj s, t: TransactionStatuss |
    { s.status = t.status, s.transaction = t.transaction}
```

3.6.1 Summary

This chapter describes requirements engineering and its approach to feature oriented modelling. It places FORM withing FOSD approach and describes FORM's notation. The next chapter describes modelling an online trading system with FORM.

Chapter 4

Modelling the Online Trading System in FORM

Shaker et al. developed the FORM on a motivating example, a telephony system, that had already been thought out and clearly defined in terms of features. The services and features of a telephony system can be freely combined to form different configurations; each telephone call is configured with the features to which the caller and callee subscribe. The features can interact in different ways, including expected and unexpected interactions. A caller is not aware of the features a callee has, and a callee is not aware of the caller's features. For example, suppose that a user has Automatic Call Back enabled, and he or she calls a user who has Call Waiting enabled. In this case, Call Waiting takes precedence over Automatic Call Back, so that if the callee ignores the call waiting signal, the Automatic Call Back feature takes effect and records information to reestablish the second call once the first call ends.

On the other hand, an online trading system (OTS) is not primarily thought of as a set of distinct features. It offers variability in terms of different types of trade orders, but these orders do not interact with each other. A user can create unlimited number of orders, each of which invokes its own feature machine (FM) that runs independently. Some orders create and block other FMs, but the interactions are all planned and are included in the FM models. That is, the features themselves interfere only when they are designed to do so. Furthermore, different trading systems never interact with each other. The OTS is our system-under-development (SUD) and the domain describes the OTS's environment,

and the behaviour model describes the SUD's behaviour in terms of the SUD's effect on its environment. We assume that there is only one Providing Trading System (PTS) that provides data about securities and their prices.

The OTS case study is an evaluation of only the expressiveness of the FORM notations. Specifically, the case study evaluates the ability to which FORM's notations are able to model all aspects of the requirements for an OTS including planned interactions between features. The case study does not evaluate FORM's approach to resolving unplanned interactions between features.

Section 4.1 first describes the domain model of the OTS. Sections 4.2 and 4.3 describe the behavioural model. We chose to describe three FMs describing the behaviour of the OTS in detail. Examples of other FMs can be found in Appendix B. First, the simplest, base FM (market order) is described. Then an example of EFM that overrides the base FM, limit order, is given. Finally, section 4.3.1 gives an example of an EFM that spawns and kills instances of other EFMs, bracket order, is given.

4.1 Domain Model

A simple version of the OTS domain model was presented in Section 3.5.1 as means to show key aspects of a FORM domain model. The full version includes an additional entity, `Ticker`, and additional messages between the OTS and two actors, `User`s and the `PTS` (see Figure 4.1). A `Ticker` is an entity that is controlled by both the domain and the system under development (SUD).

Each `User` has a unique ID. It has a set of trade transactions that are being processed or have been processed. Ownership is shown by way of an association, `UserTransaction`, that has two roles: `user` and `transaction`. Users are the source of several messages for the OTS, like `GetTickerInfo()`, `RequestOrder()`, and `Cancel()`; and are the target of several messages from the OTS, like `ForwardTIToCustomer()`, `TimeExpired()`, `OrderCancelled`, and `TransactionCompleted()`.

The `PTS` also has a unique ID. It communicates only with the OTS, therefore, it is the source and target of multiple messages from the OTS.

Each `Transaction` is an entity that holds information about the status and results of some trade request. Each time an order is placed, a `Transaction` object is created that

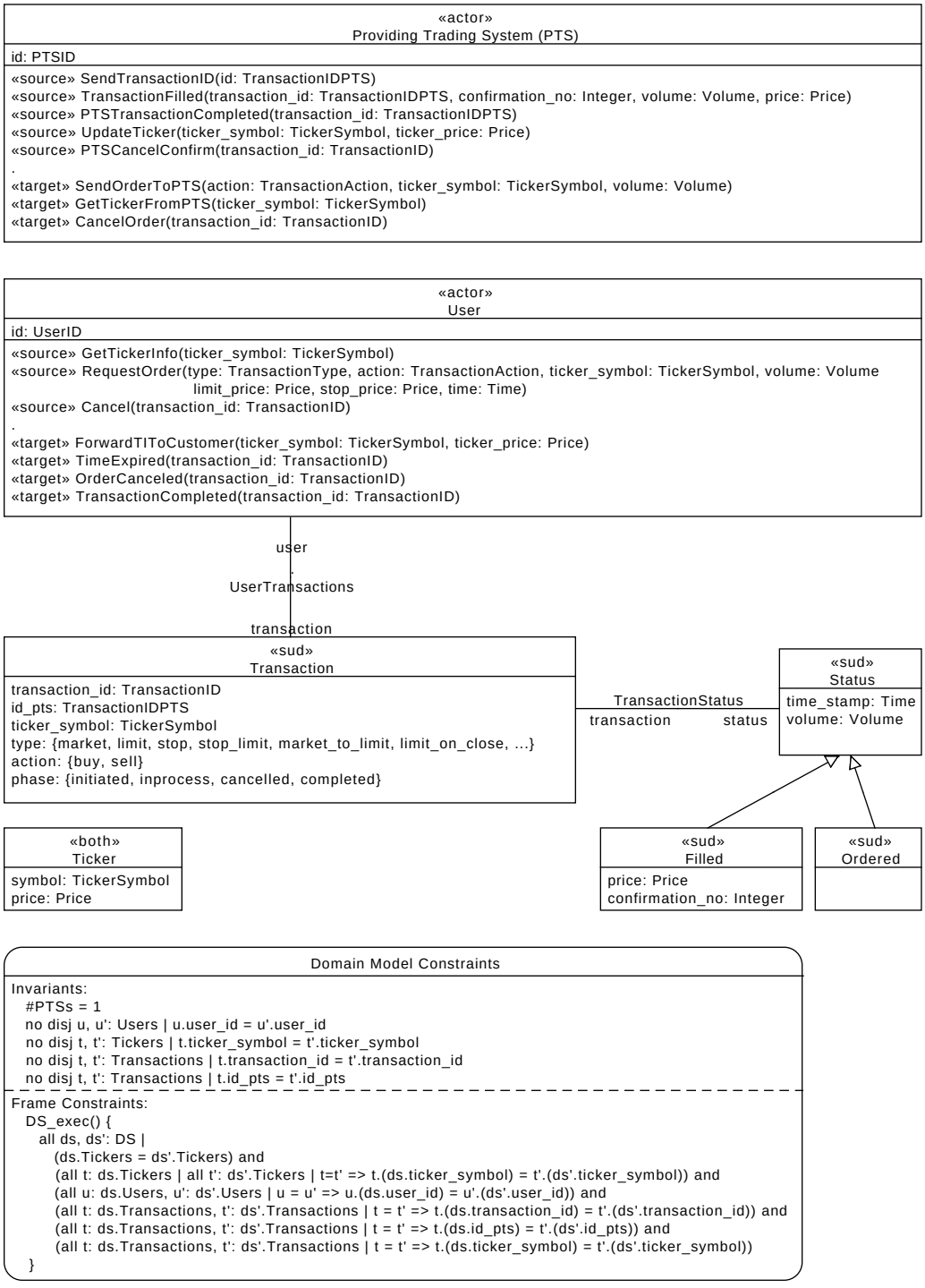

Figure 4.1: Domain model of an online trading system

holds of all the corresponding data. Whenever an order is (partially) filled, that same transaction is updated with information about the filled order. The `Transaction` object has a unique ID that is used by the OTS. It also has an ID assigned by the PTS that the PTS uses to distinguish it from other transactions in the PTS. We require both IDs because transactions may exist in the OTS for a long time before they become known to the PTS and are assigned a PTS ID. For example, in a limit-on-open order, the user transmits a request for trade, but the request does not get processed before the next morning. After the stock market opens in the morning, the OTS checks with PTS the market price of the stock. If the price does not match the limit price, the order is never transmitted and the OTS sends the user a notification about the cancelled order. In this case, the only ID that the corresponding `Transaction` object has is the OTS ID (see Figure B.6).

Each `Transaction` object also holds information about the stock being traded (`tickerSymbol`), the `action` to be performed (whether it is a buy or a sell order), the current `phase` of the order (initiated, in process, accepted, denied, cancelled, or suspended), the `type` of the order (whether it is market, limit, stop, bracket, etc.), and the `status` of the order (whether the order has been placed, and all information about partial fills of the order). A `Transaction` is completely controlled by the SUD, since it holds information generated only by the OTS. It is created when the order is placed, and is updated when the order is processed. On the other hand, a `Ticker` object is controlled both by the environment and the SUD. It holds information about securities, including the name (`symbol`) and the market `price` of the security. The domain controls the set of tickers that can be traded by the SUD, and the SUD updates ticker information (i.e., price) each time it receives information from the PTS.

A complete Alloy representation of the domain model is given in Appendix A. It was modelled with guidelines described in section 3.5.1 and includes the domain model of the OTS together with constraints.

4.2 Base Feature Model

The most basic type of trade order is a market order: a request to immediately buy or sell some number of shares of a stock, at the current market price. All other types of trade orders turn into market orders once a specified set of rules are satisfied. Therefore, we

chose to model market orders as a general feature machine (FM) and model all other types of trade orders as enhancement FMs (EFMs) that augment and override a market order or some other order. Our model for market order was presented in its entirety in Section 3.5.2.

4.3 Enhancement Feature Models

All other types of trade orders can be considered as enhancements of a market order. They either suspend the market order FM to perform some preprocessing, or they allow the market order to proceed and they modify the results. For example, a buy-limit order is modelled as an EFM that suspends the execution of an associated market order FM, waits until the market price of a particular stock has reached a set limit price, and then allows the market order FM to execute the trade.

We modelled nine different EFM models for different types of orders. They are all presented in Appendix B: a limit order (see Figure B.3), a stop order (see Figure B.4), a stop-limit order (see Figure B.5), a limit-on-open order (see Figure B.6), a limit-on-close order (see Figure B.7), a market-to-limit order (see Figure B.8), good-till-cancelled order (see Figure B.9), a good-till-date/time order (see Figure B.10), and a bracket order (see Figure B.11).

4.3.1 Buy Bracket Order

A bracket order combines market, limit, and stop orders so that a customer can limit losses and guarantee profits by limiting an order with two opposite-side orders. A bracket order FM is an example of a more complex EFM. It not only adds actions to the base FM behaviour but also creates other EFMs. When the EFM completes, its completion transition creates two more EFMs. When one of the EFMs sends an order to the PTS, the other one is cancelled.

A buy bracket-order FM (BOBFM) is constructed whenever a user sends a request for an order, whose `type` is equal to `bracket` and whose `action` is equal to `buy` (see Figure 4.2):

```
RequestOrder( type, action, ticker_symbol, volume, offset )[ type=bracket, action=buy ]
```

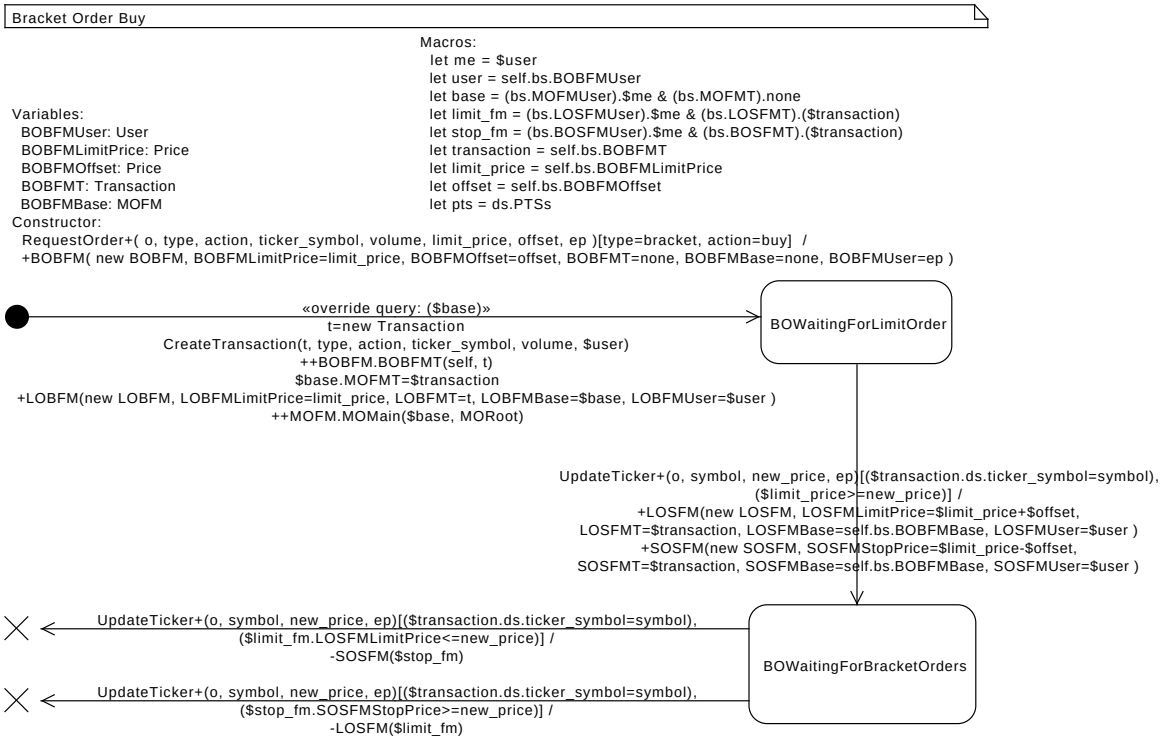

Figure 4.2: A FORM enhancement feature machine for a bracket order

In the constructing transition, a new `Transaction` object is created, with corresponding `type`, `action`, `ticker_symbol`, and `volume`. The transition also constructs a buy-limit order FM (LOBFM) and associates it with the same transaction. At the same time, a market order FM is created (in response to the same `RequestOrder` message). The BOBFM's constructing transition disables MOFM, and enters state `BOBWaitingForLimitOrder`.

While in state `BOBWaitingForLimitOrder`, the BOBFM keeps track of ticker updates from the PTS (which are issued in response to requests from the LOBFM). When the ticker price reaches the limit price, the terminating transition of the LOBFM transforms the order into a market order and sends the order to the PTS. At the same time, the BOBFM creates two new orders, a sell-limit order and a sell-stop order. The corresponding FMs are created, LOSFM and SOSFM, whose prices are updated with the BOBFM offset: `$limit_price+$offset` and `$limit_price-$offset`. BOBFM transitions to state `BOBWaitingForBracketOrders`.

The BOBFM again keeps track of ticker updates from the PTS system. If the ticker price rises on the market:

```
UpdateTicker+( symbol, new_price ) [ $limit_fm.LOSFMLimitPrice<=new_price ]
```

the LOSFM reacts by turning the order into a market order (e.g. modifying the domain state) and activating the base feature machine (changes the state of an MOFM, see Figure 3.6). At the same time, BOBFM terminates the SOSFM:

```
-SOSFM()
```

Alternatively, if the ticker price falls on the market, the SOSFM reacts:

```
UpdateTicker+( symbol, new_price ) [ $stop_fm.SOSFMStopPrice>=new_price ]
```

the SOSFM reacts by turning the order into a market order and activating the base feature machine, MOFM. At the same time, BOBFM terminates the LOSFM:

```
-LOSFM()
```

In both scenarios, the BOBFM exits its waiting state and terminates. At this point, the base FM, MOFM, continues and executes the desired order.

4.4 Summary

This chapter describes modelling the online trading system with FORM. It describes domain, behavioural, and functional model in detail. The next chapter describes the analysis of modelling the online trading system with FORM.

Chapter 5

Results and Evaluation

This chapter relates our experience in performing the case study and presents our evaluation of the FORM notation. Section 5.1 first presents our results of using the Alloy analyzer to check our FORM domain model. Section 5.2 gives evaluation of the FORM modelling language and introduces changes made to the methodology. Section 5.3 gives examples of workarounds and deficiencies of FORM encountered while modelling an online trading system (OTS).

5.1 Alloy Analysis of a FORM Domain Model

We used the Alloy analyzer to perform simple checks of our Alloy model for the FORM domain model of the OTS example. We were interested in whether the model was over-constrained (i.e., whether the analyzer was unable to find example of instances), and whether any of the generated model instances looked inappropriate. Recall that the analyzer looks for instances that either satisfy or violate constraints. If the analyzer does not return any instances, then the model is overconstrained. When limiting the scope of the analysis to two domain states, two `User`s and two `Ticker`s, the analyzer returned several instances of the model, which showed that we did not over-constrain the model. After manually inspecting the instances, we found two flaws in FORM: (1) objects had attributes that are not in the domain state, and (2) constraints were true for all instances in the universe, so the analyzer generated the model with instances that we are not interested in. After modifying the constraints on the domain system, we reran the analyzer

and concluded that all instances of domain state represented the model in the correct way. Figures 5.1 and 5.2 show an example of an instance trace returned by the analyzer. The trace shows only `User`, `Ticker` and `GetTickerInfo` objects with their attributes. In the first domain state (see Figure 5.1), `User1` with the `UserID` initiates a `GetTickerInfo` request for a `Ticker0` with `TickerSymbol1` and `Price1`. The next domain state (see Figure 5.2) is the same, except that `User1` becomes an observer of the `Price` of objects `Ticker0` and `Ticker1`. We can see that tickers do not change their symbols, and users keep their user ID. We notice that the price of `Ticker0` changes (in the first domain state it has a link to `Price0` and in the second domain state it has a link to `Price1`), which is not a problem since ticker prices do change over time. We also notice that all constraints on the cardinality of attributes hold: message `GetTickerInfo` has only one link to a ticker symbol and one link to an end point; user has only one Id; tickers have only one ticker symbol; and tickers have only one price value at a time although their values can change.

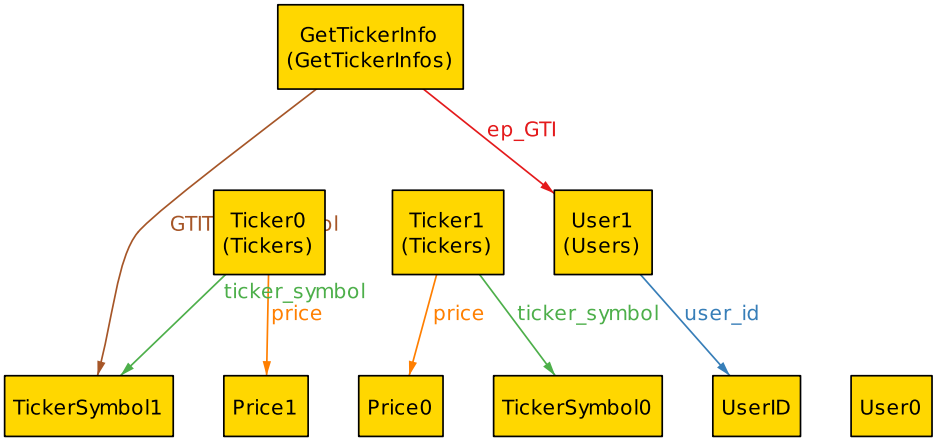

Figure 5.1: The first domain-state instance of an Alloy-generated trace of the OTS domain

5.2 Evaluation of the FORM Notation

This dissertation evaluates the expressiveness of the Feature-Oriented Requirements Modelling (FORM) notation by modelling the requirements for a system-under-development (SUD) that can be thought of in terms of its features. We modelled an online trading system (OTS) that receives requests from customers about buying or selling stocks and

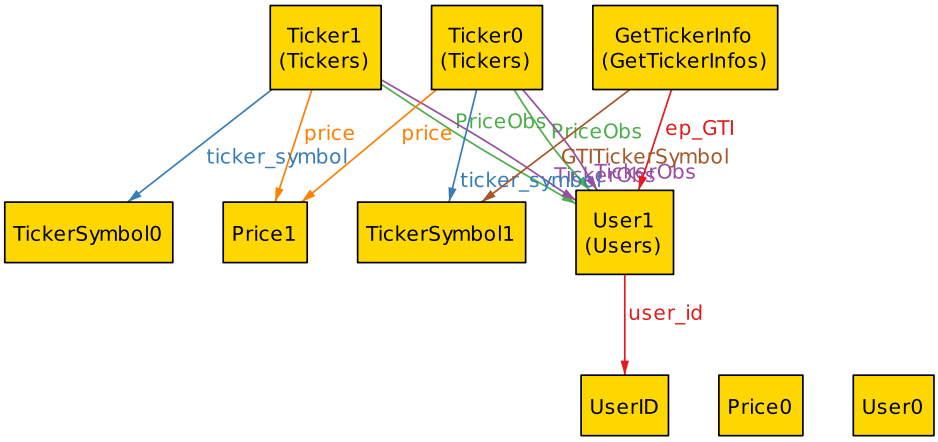

Figure 5.2: The second domain-state instance of an Alloy-generated trace of the OTS domain

forwards requests to a providing trading system (PTS). The OTS offers variability in terms of the types of orders that customers can request (e.g., market order, limit order, stop order, etc.). In our model, each type of order is represented as a different feature. A market order is modelled as a base feature machine (FM), and other order types are modelled as enhancement features (EFMs) that extend or override the market-order feature. The OTS is smart enough to process orders until they are ready to enter into the market. For example, the PTS will not be aware of a limit order until the market price of the stock to be traded has reached a required limit price.

In order to successfully model the OTS example using FORM, we required six changes to the FORM modelling language, three of which were successfully adopted into the FORM definition. First, the scope of the triggering event of the constructor of a feature machine needed to be extended to include the constructing transition (5.2.1). Second, we identified some deficiencies in the set of well-formedness constraints on a domain model. Existing constraints were modified to operate on only elements in a domain-state (DS) instance rather than on types, and an additional constraint was added (5.2.2). Third, frame predicates had to be added to specify that some variables don't change value during execution (5.2.3). The changes are described in detail in the following subsections. Dissertation uses terms "pre-dissertation FORM" and "post-dissertation FORM" to refer to how the FORM methodology worked before and after the case study, respectively. All examples presented in the dissertation so far and in the appendices use the post-dissertation FORM notation.

5.2.1 Extended the Scope of the Trigger Event of Feature-Machine Constructors

As described in Section 3.5.2, every FM in a FORM model has a constructor. The constructor is invoked when a particular *domain change event* (DCE) happens that enables the constructor's guards. For example, a new instance of a Market-Order Feature Machine (MOFM) is created when a user sends a message `RequestOrder`, which carries information about the order type, the type of action (buy or sell), the ticker symbol, and the amount the customer wants to buy or sell. Figure 3.5 shows that the constructor of the MOFM invokes the machine and initializes the local variables:

```
+MOFM( new MOFM, MOFMTransaction=none, MOFMUser=ep )
```

The MOFM keeps information about the user who initiated the request, `MOFMUser`, and the new transaction to be processed, `MOFMTransaction`. The constructor of an FM does not make any changes to the domain state, because there is no notion of overriding the constructor of an FM and all feature actions should be override-able. Therefore, the local variable `MOFMTransaction` is initialized to `none` in the constructor and is reassigned when the new `Transaction` is created.

In pre-dissertation FORM, the scope of a constructor's trigger event was limited to just the constructor, so in order to use parameters of the `RequestOrder` in creating the `Transaction` object in the constructing transition, the MOFM would need to create additional local variables to hold the parameter values. This is wasteful because it forces the declaration of a number of additional local variables that are used only in the initializing transition. The following lines show how the MOFM constructor and constructing transition would be modelled in pre-dissertation FORM:

```
Local variables:
   MOFMTransaction: Transaction
   MOFMType: TransactionType
   MOFMAction: TransactionAction
   MOFMTickerSymbol: TickerSymbol
   MOFMVolume: Volume
   MOFMUser: User

Constructor:
```

```
   RequestOrder+( o, type, action, ticker_symbol, volume, ep )
   +MOFM( new MOFM, MOFMTransaction=none, MOFMType=type, MOFMAction=action,
          MOFMTickerSymbol=ticker_symbol, MOFMVolume=volume, MOFMUser=ep )


Constructing transition:
   order=new Ordered
   +Ordered( order, time_stamp=currTime, volume=self.bs.MOFMVolume )
   t=new Transaction
   +Transaction( t, transaction_id=new TransactionID, id_pts=none,
                 ticker_symbol=self.bs.MOFMTickerSymbol, action=self.bs.MOFMAction,
                 type=market, phase=initiated )
   +TransactionStatus( new TransactionStatus, transaction=t, status=order )
   +UserTransactions( new UserTransactions, transaction=t, user=self.bs.MOFMUser )
   ++MOFM.MOFMTransaction( self, t )
```

In consultation with Shaker et al., we modified the semantics of FORM and extended the scope of the constructor to include the first transition that executes, called the *constructing transition*. With this change the domain change actions (DCA) in the constructing transition in MOFM can access directly the parameters of the `RequestOrder` message. Thus, MOFM's constructor and the constructing transition modelled in post-dissertation FORM is as follows:

```
Local variables:
   MOFMTransaction: Transaction
   MOFMUser: User


Constructor:
   RequestOrder+( o, ttype, taction, tticker_symbol, tvolume, ep )
   +MOFM( new MOFM, MOFMTransaction=none, MOFMUser=ep )


Constructing transition:
   order=new Ordered
   +Ordered( order, time_stamp=currTime, volume=tvolume )
   t=new Transaction
   +Transaction( t, transaction_id=new TransactionID, id_pts=none,
```

```
                ticker_symbol=tticker_symbol, action=taction, type=market,
                phase=initiated )
+TransactionStatus( new TransactionStatus, transaction=t, status=order )
+UserTransactions( new UserTransactions, transaction=t, user=self.bs.MOFMUser )
++MOFM.MOFMTransaction( self, t )
```

5.2.2 Fixed Domain-Model Constraints

The Alloy analyzer tries to find an example or a counterexample model instance that satisfies or violates the given specification. Since a FORM domain model (DM) can be completely translated into the Alloy notation, we are able to perform type checks and check well-formedness constraints. When running tests, we discovered some minor flaws in the original FORM constraints for a DM.

First, we discovered that we need a constraint that is comparable to constraint number 2 in section 3.6, but in case where attributes, parameters, or roles are of non-concept types. The constraint states if a concept's attributes, parameters, or roles of some non-concept type are in domain state (DS), then the concept itself must also be in the DS: *For every attribute, parameter, or role "A" of each concept "C" where "A" has non-concept type "T": A.T in Cs* For example, the object `User` has attribute `user_id` that is of non-concept type `UserID`. Our model has to make sure that if an attribute (e.g. `user_id`) of `User` object is in DS, then its object, `User` must be in the DS, as well:

```
user_id.UserID in Users
```

Second, recall that each concept has a signature in the DM, and a set of elements in the DS space. For example, the set of users in the OTS example is first defined with a signature in the DM, and then with its set in the DS:

```
sig User {}

sig DS {
  Users: set User
}
```

In pre-dissertation FORM, constraints were defined over types. For example, the following constraint states that each user has only one user ID:

```
all user: User | #user.user_id = 1
```

However, we are interested only in constraining instances in the DS. Therefore, such a constraint should refer instead to elements of the concept set in the DS. In post-dissertation FORM, the above constraint is correctly modelled as:

```
all user: Users | #user.user_id = 1
```

The same change applies to constraints on the number of a message's end-points. Thus

```
#ep_GTI.GetTickerInfo =< 1
```

is corrected to be

```
#ep_GTI.GetTickerInfos =< 1
```

This update also applies to constraints on the number of same-relationship links between two objects in the DS. Thus

```
no disj r1, r2: TransactionStatus |
  {r1.status = r2.status, r1.transaction = r2.transaction}
```

is corrected to be

```
no disj r1, r2: TransactionStatuss |
  {r1.status = r2.status, r1.transaction = r2.transaction}
```

5.2.3 Added Frame Predicates

Every DM in a FORM model comprises of set of signatures, an initialization predicate, and a definition of the DS space. When the Alloy model is constructed, one can analyze it using the Alloy analyzer and explore different instances of the model. In particular, we consider execution traces of domain-state instances. In Alloy, operations specify pre- and post-conditions on the domain state. However, Alloy expects the post-conditions to cover all elements of the domain state, not just the elements whose values are changed by the operation. Frame predicates are used to invariably constrain values that never change, so that the invariants need not be explicitly included in the definition of every operation. For example, the set of tickers our OTS offers is always the same. Therefore, we added frame predicate, `DS_exec`, to the domain model:

```
pred DS_exec() {
  all ds, ds': DS | ds.Tickers = ds'.Tickers
}
```

5.3 Workarounds and Deficiencies

This section describes workarounds that we had to devise in order to model our system correctly. Section 5.3.1 describes the adding of the notion of time and timeouts to the feature machine and to the domain model. Section 5.3.2 describes the problem of modelling compliance rules in out OTS example, where trade orders are processed only if they satisfy compliance rules. Section 5.3.3 describes the workaround for enhancing feature machines to reference their base feature machine(s) that had to be developed. Sections 5.3.4 and 5.3.5 describe suggestions for incorporating inheritance and polymorphism, which would make FORM modelling easier.

5.3.1 Added Time and Timeouts

Modelling an OTS requires notion of time and timeouts. All transactions include a time-stamp attribute that records the date and time when the transaction was created. Moreover, some types of transaction orders enable customers to specify deadlines in which an order is cancelled, if the order is not executed by a certain date and time (e.g., a Good-till-Date/Time Order). To be able to model these types of orders, we added a primitive type `Time` to our FORM domain model (DM). We also created a new keyword `currTime` that evaluates to the current time, and a new keyword `after(given_time)` that defines a domain change event that occurs at the time `given_time`.

For example, the feature model for Good-till-Date/Time Order expressed in post-dissertation has a transition with a DCE `after(end_time)` that triggers the cancellation of the order, where `end_time` denotes some date or time in the future:

```
after( self.bs.end_time ) /
+CancelOrder( new CancelOrder, self.bs.GTDTOFMTransaction.id_pts, ds.PTSs )
```

5.3.2 Policy Language

Customers using an OTS must abide by *compliance rules*, which are regulations and laws that govern trading. Pre-trade compliance rules are checked before an order can be placed. For example, when a user decides to buy IBM shares, the system has to check whether the order is within the trade regulations: whether the security is currently being traded, whether there is a limit on how many shares of the security can be bought or sold, at what price, etc. These rules offer a second example of the variability in an OTS.

In general, there is a question about how to model *policies*, which are high-level variable constraints on allowable behaviour. Policies have not been incorporated into FORM so we did not include compliance rules officially in our case study. However, we did attempt to model them, and we sketch here a possible approach that still needs to be investigated.

We suggest modelling compliance rules using functional models and invoking the functions as guards on any transition, in which a new order is requested. Rules and their parameters are represented as entities in the domain model (see Figure 5.3). We attempted to model two simple rules: a trade-block rule and a trade-quantity-limit rule. A *Trade-Block Rule* suspends order placements and trade for a particular ticker. The rule carries information about the ticker symbol, the start and end time of the suspension, the entry time of the rule, and whether it is the buying or selling of shares that is forbidden. A *Trade-Quantity-Limit Rule* restricts the volume on order placements for particular ticker. The rule carries information about the ticker symbol, lower or upper limits on the volume traded, whether the limit applies to the buying or selling of shares, and the entry date of the rule.

We added a new concept to the domain model of the OTS, `Rule`, that can either be `TradeQuantityLimitRule` or `TradeBlockRule`. A rule can apply to a particular ticker or a particular user, so we added two relationships to the domain model, `UserRule` for restrictions on a user, and `TickerRule` for restrictions on a Ticker. We also modified the Market-Order FM (MOFM) (see Figure 5.4) to include a new step that checks all relevant compliance rules. When the MOFM is instantiated, it creates a transaction and transfers into intermediate state `MOcheckingComplianceRules`. There are two transitions that exit this state: one that is triggered by the function `CheckComplianceRules` evaluating to true, and that leads to the rest of the FM:

```
[ ! CheckComplianceRules( $transaction, $user ) ] /
```

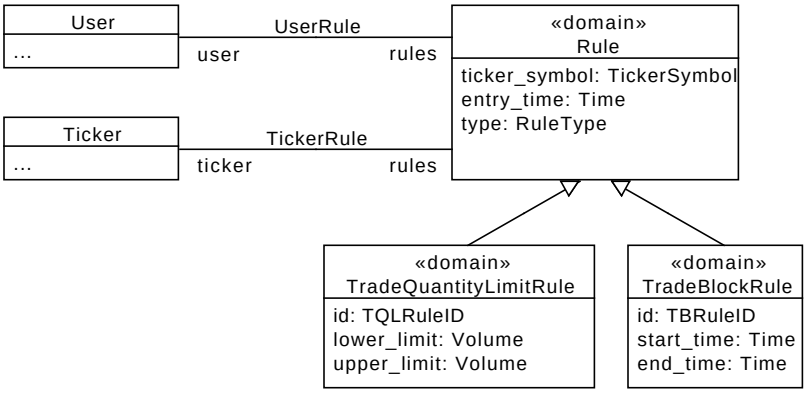

Figure 5.3: OTS domain model augmented with compliance rules

```
+OrderNotCompliant( new OrderNotCompliant, $transaction.ds.transaction_id, $user )
++Transaction.phase( $transaction, non_compliant )
```

and one transition that is triggered by the function `CheckComplianceRules` evaluating to false, and that terminates the FM:

```
[ CheckComplianceRules( $transaction ) ] /
     +SendOrderToPTS( new SendOrderToPTS, action=$transaction.ds.action,
                      ticker_symbol=$transaction.ds.ticker_symbol,
                      volume=$transaction.ds.volume, $pts )
```

`CheckComplianceRules` is a functional model that navigates the domain state and invokes the functional model of every rule that is associated with the trade order's use and the ticker symbol of the shares being traded. In this way, the functional model for `CheckingComplianceRules` can be context independent and can be reused in the guard of any FM:

```
pred CheckComplianceRules(t: Transaction, u: User) {
   all ds: DS |
      TradeQuantityLimit[ ds, u, t,
           ((ds.transaction_status_transaction).t).(ds.transaction_status_status)] and
      TradeBlock[ ds, u, t,
           ((ds.transaction_status_transaction).t).(ds.transaction_status_status)]
```

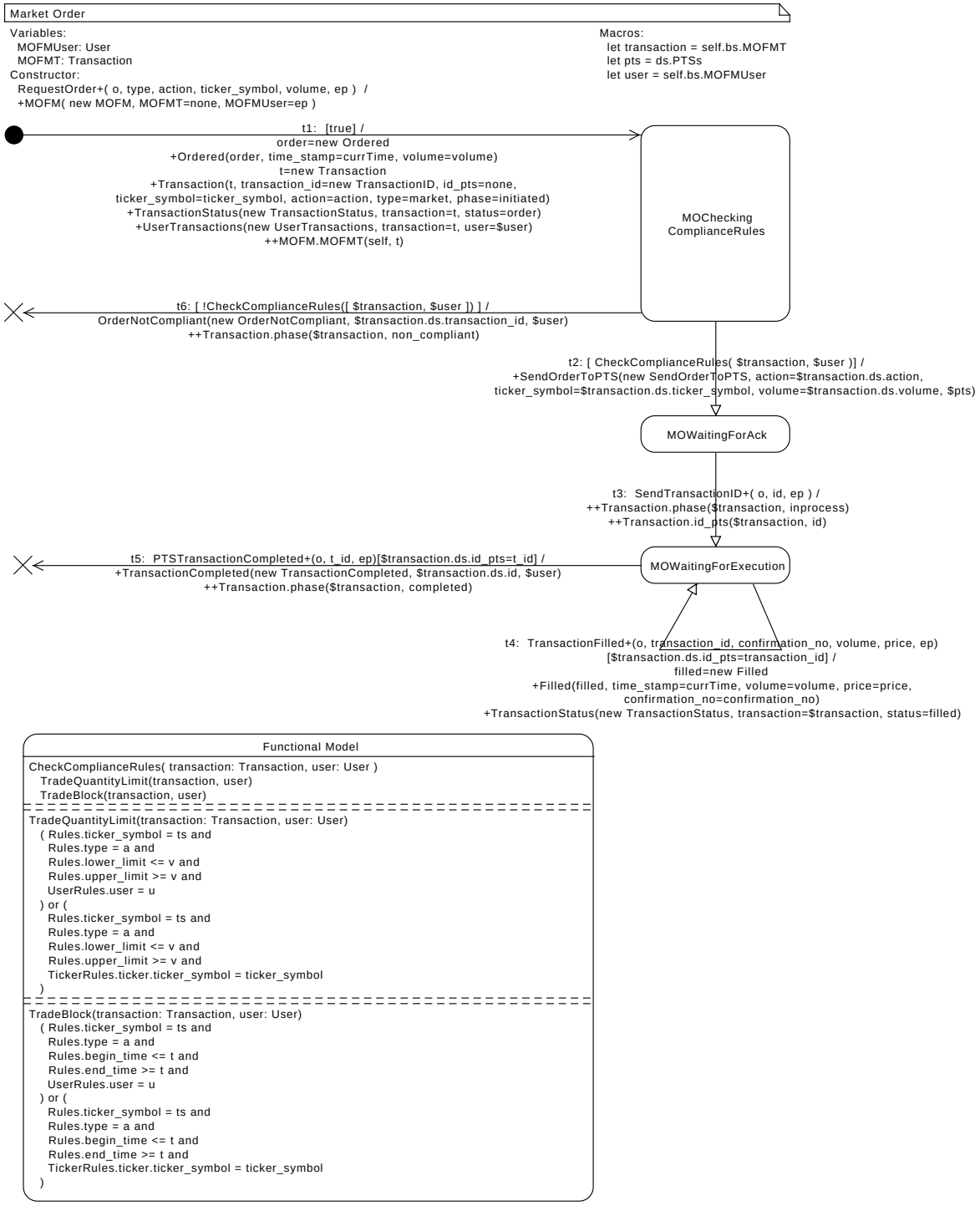

Figure 5.4: OTS market order feature machine augmented with compliance rule checking

```
}
```

In addition, we created for each specific type of compliance rule a functional model that checks adherence to that rule. For example, the `TradeQuantityLimit` function checks that the volume on an order of a restricted security within allowed bounds:

```
pred TradeQuantityLimit[ds: DS, u: User, t: Transaction, s: Status] {
   ( some r: ds.UserRules |
      (ds.Rules).(ds.rule_ticker_symbol) = t.(ds.ticker_symbol) and
      (ds.Rules).(ds.rule_type) = t.(ds.action) and
      (ds.Rules).(ds.lower_limit) >= s.(ds.volume) and
      (ds.Rules).(ds.upper_limit) <= s.(ds.volume) and
      (r.(ds.user_rule_user) = u)
   ) or ( some r: ds.TickerRules |
      (ds.Rules).(ds.rule_ticker_symbol) = t.(ds.ticker_symbol) and
      (ds.Rules).(ds.rule_type) = t.(ds.action) and
      (ds.Rules).(ds.lower_limit) >= s.(ds.volume) and
      (ds.Rules).(ds.upper_limit) <= s.(ds.volume) and
      (r.(ds.ticker_rule_ticker).(ds.symbol) = t.(ds.ticker_symbol))
   )
}
```

Similarly, the `TradeBlock` functional model checks whether a requested order is for a security whose trade is currently being blocked:

```
pred TradeBlock(ds: DS, u: User, t: Transaction, s: Status ) {
   ( some r: ds.UserRules |
      (ds.Rules).(ds.rule_ticker_symbol) = t.(ds.ticker_symbol) and
      (ds.Rules).(ds.rule_type) = t.(ds.action) and
      (ds.Rules).(ds.start_time) >= s.(ds.time_stamp) and
      (ds.Rules).(ds.end_time) <= s.(ds.time_stamp) and
      (r.(ds.user_rule_user) = u)
    ) or ( some r: ds.TickerRules |
      (ds.Rules).(ds.rule_ticker_symbol) = t.(ds.ticker_symbol) and
     (ds.Rules).(ds.rule_type) = t.(ds.action) and
     (ds.Rules).(ds.start_time) >= s.(ds.time_stamp) and
      (ds.Rules).(ds.end_time) <= s.(ds.time_stamp) and
```

```
      (r.(ds.ticker_rule_ticker).(ds.symbol) = t.(ds.ticker_symbol))
    )
}
```

Such an approach seems promising as long as the compliance rules are simple and stateless. If a compliance rule is stateful, then it would need to be modelled using a behavioural model, and the coordination between feature machines and compliance-rules machines would need to be worked out.

5.3.3 Referencing the Base Feature Machine

FORM is a *context-dependent* modelling language, in that each enhancement feature must know about the feature(s) it is overriding. When we first modelled enhancement FMs (EFM), each EFM referred to the base FM associated with the user who requested the order. However, in practise, each user may have multiple orders pending at the same time. Fortunately, a user may submit only one order at a time, i.e., no one can submit multiple orders in parallel. When a user submits a new `RequestOrder`, the message triggers the construction of both the base and enhanced FM. At this point, there are only two machines in the system that have both the same user and an empty transaction object. The constructing transition of the EFM is modified to find the newly constructed base FM in the behavioural state and that has the same user as the EFM as is processing an empty transaction object; the EFM associates this FM with a local variable in the EFM.

The FM is described in the behavioural model. Each model state represents a specific *behavioural state* (BS) of the FM (i.e., a particular valuation of object sets and transitions). Since FORM only has a formal description of terms of traces, we introduced an Alloy syntax for referring to BS variables. For example, every FM has a reference to the user that initiated the request for transaction:

```
MOFMUser
```

In order to find a reference to that variable, the variable needs to be searched for in the behavioural state (BS):

```
bs.MOFMUser
```

When a user transmits a trade request, both FMs, base and enhancement FM, are constructed. In order for the EFM to monitor and control actions of the base FM, it needs to have a reference to it. This reference is established by finding the base FM that is active and that has the same user as the EFM in the behavioural state `bs`:

```
(bs.MOFMUser).LOBFMUser
```

Since a user can have more than one transaction, this constraint can return more than one base FM. When the system initially receives a trade request, both newly instantiated machines have same user and both have an empty transaction. Therefore, we specify a second constraint that returns all base FMs that have an empty transaction in the behavioural state `bs`:

```
(bs.MOFMT).none
```

We intersect both constraints to reference the base machine that has the same user as `LOBFMUser` and has an empty transaction. Thus, `LOBFM`'s reference to the base FM in LOBFM is set in the following way:

```
++LOBFM.LOBFMBase( self, (bs.MOFMUser).LOBFMUser & (bs.MOFMT).none )
```

5.3.4 Inheritance of Feature Machines

FORM uses functional models to capture common actions that are performed multiple times. For example, an OTS uses a functional model to represent all of the actions that are performed whenever a new trade transaction is recorded. When a transaction is created, it is stored in the DS, and the FM has a local variable that is an alias to the object. Ideally, the signature of the function would include a reference to the feature machine (FM):

```
CreateTransaction(fm: FM, type: TransactionType, action:TransactionAction,
                  ts: TickerSymbol, volume: Volume, u: User)
```

so that a new transaction can be added to the domain state and the alias to the local variable can be set inside the function:

```
transaction=new Transaction
++fm.FMTransaction(self, transaction)
```

However, the type of the FM is different for each feature, and FORM is context dependent, the function cannot take a reference to a derived type of a FM. Instead, the transaction is created outside of the function, before the function is called:

```
t=new Transaction
```

the function takes the reference to a newly created `Transaction` object:

```
CreateTransaction(t: Transaction, type: TransactionType, action:TransactionAction,
                  ts: TickerSymbol, volume: Volume, u: User)
```

and the linking of newly updated object to associated local variable is performed after the `CreateTransaction` function is called:

```
++MOFM.MOFMTransaction(self, t)
```

5.3.5 Polymorphism for Messages

A `RequestOrder` message carries information about a trade order that a user wishes to be performed, including the type of the order, whether the user wants to sell or buy shares, the ticker symbol of the security to be traded, and the volume of the trade. The message may additionally carry information about the requested limit price, stop price, date to cancel transaction, etc. The extra information is not always needed, so it would be useful to allow messages to be polymorphic. Currently, `RequestOrder` always has the following signature:

```
RequestOrder( type: TransactionType, action: TransactionAction,
              ticker_symbol: TickerSymbol, volume: Volume, limit_price: Price,
              stop_price: Price, time: Time )
```

although parameters `limit_price`, `stop_price`, and `time` are often set to `none`.

Chapter 6

Conclusion

This dissertation evaluates the expressiveness of the FORM notations. It evaluates the extent to which we were able to model all aspects of the requirements for an OTS, including planned interactions between features. The dissertation does not evaluate FORM's approach to resolving unplanned interactions between features.

We modelled an OTS that receives requests from customers about buying or selling securities on a stock market. The OTS offers variability in terms of the types of orders that customers can request (e.g. market order, limit order, stop order, etc.). In the OTS, each type of order is modelled as a different feature. The market order was chosen as a base feature that other features enhance. The OTS is smart enough to process orders until they are ready to be entered into the market.

The case study revealed two classes of deficiencies of the FORM notation. We were able to resolve the first class of deficiencies through simple fixes and extensions to FORM. First, the scope of the triggering event of an FM's constructor needed to be extended to the FM's constructing transition. Second, we identified some deficiencies in the set of well-formedness constraints. The constraints were modified to operate only over elements in the domain-state instance, and an additional constraint was added. Third, frame predicates were added to specify which variables do not change value during execution. We also successfully translated the domain model into the Alloy notation and checked the well-formedness constraints.

However, there was a second class of deficiencies of the FORM notation that was not as amenable to resolution through small changes to the FORM notation. These deficiencies, of

which there were three, require more substantial changes, along the lines that we sketched in this dissertation. First, we added language constructs for modelling time and timeouts. Second, introduced an approach to express compliance rules or policies using the FORM functional model, and invoking the functions as guards on transitions in behavioural model. Third, we devised a method by which an enhancement FM could be linked to its base FM(s). However, these solutions need to be further formalized and evaluated.

References

[1] Apel, S., Kastner, C.: An Overview of Feature-Oriented Software Development. Journal of Object Technology 8 (2009) 2, 15, 16

[2] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P.: Object-Oriented Development: The Fusion Method. Pretence Hall (1994) 13

[3] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional (2000). 1, 15

[4] Wikipedia: Stock Market. http://en.wikipedia.org/wiki/Stock_market 4, 5, 7

[5] Trading Systems Expert. http://www.tradingsystemsexpert.com/ 5

[6] U.S. Securities and Exchange Commission: www.sec.gov 7

[7] Finance Library: www.finance-lib.com 7

[8] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006) 13, 17, 18, 20, 21

[9] Jackson, M.: Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices. ACM Press/Addison-Wesley Publishing Co. (1995) 13, 25

[10] van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley (2009) 13

[11] Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (3rd Edition). Prentice Hall (1994) 1, 13, 14, 15

[12] Shaker, P., Atlee, J. M.: Feature-Oriented Requirements Modelling. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2. (2010) 1, 2, 12, 15, 25, 29, 34, 38

[13] Zave, P., Jackson, M,: A component-based approach to telecommunication software. IEEE Software (1998) 25

APPENDICES

Appendix A

The Domain Model of the Online Trading System in Alloy Notation

```
sig User {}
sig UserID {}

sig PTS {}
sig PTSID {}

sig Ticker {}
sig TickerSymbol {}
sig Price {}
sig Volume {}

sig Transaction {}
sig TransactionID {}
sig TransactionIDPTS {}
enum TransactionAction {buy, sell}
enum TransactionPhase {initiated, inprocess, cancelled, completed }
enum TransactionType {market, stop, limit, stop_limit, market_to_limit, limit_on_open,
  limit_on_close, immediate_or_cancel, good_till_canceled, good_till_datetime, bracket_order}

sig TransactionStatus {}
abstract sig Status {}
sig Filled extends Status {}
sig Ordered extends Status {}
sig Time {}
sig ConfirmationNumber {}

sig UserTransactions {}

// messages PTS
sig SendTransactionID {}
sig TransactionFilled {}
sig PTSTransactionCompleted {}
sig UpdateTicker {}
sig PTSCancelConfirm {}
sig SendOrderToPTS {}
sig GetTickerInfoFromPTS {}
sig CancelOrder {}

// messages User
sig GetTickerInfo {}
sig RequestOrder {}
sig Cancel {}
sig ForwardTIToCustomer {}
sig TimeExpired {}
sig OrderCanceled {}
sig TransactionCompleted {}

sig DS {
```

```
// <<actor>>
Users: set User,
user_id: User -> UserID,

PTSs: set PTS,
pts_id: PTS -> PTSID,

// <<sud>>
Transactions: set Transaction,
transaction_id: Transaction -> TransactionID,
id_pts: Transaction -> TransactionIDPTS,
ticker_symbol: Transaction -> TickerSymbol,
action: Transaction -> TransactionAction,
phase: Transaction -> TransactionPhase,
type: Transaction -> TransactionType,
status: Transaction -> TransactionStatus,

TransactionObs: Transaction -> (User + PTS),
transaction_id_obs: Transaction -> (User + PTS),
id_pts_obs: Transaction -> (User + PTS),
ticker_symbol_obs: Transaction -> (User + PTS),
action_obs: Transaction -> (User + PTS),
phase_obs: Transaction -> (User + PTS),
type_obs: Transaction -> (User + PTS),
status_obs: Transaction -> (User + PTS),

UserTransactionss: set UserTransactions,
user_transactions_user: UserTransactions -> User,
user_transactions_transaction: UserTransactions -> Transaction,

UserTransactionsObs: UserTransactions -> (User + PTS),
user_transactions_user_obs: UserTransactions -> (User + PTS),
user_transactions_transaction_obs: UserTransactions -> (User + PTS),

TransactionStatuss: set TransactionStatus,
transaction_status_status: TransactionStatus -> Status,
transaction_status_transaction: TransactionStatus -> Transaction,

TransactionStatusObs: TransactionStatus -> (User + PTS),
transaction_status_status_obs: TransactionStatus -> (User + PTS),
transaction_status_transaction_obs: TransactionStatus -> (User + PTS),

Statuss: set Status,
time_stamp: Status -> Time,
volume: Status -> Volume,

StatusObs: Status -> (User + PTS),
time_stamp_obs: Status -> (User + PTS),
```

```
volume_obs: Status -> (User + PTS),

Filleds: set Filled,
transaction_price: Filled -> Price,
confirmation_number: Filled -> ConfirmationNumber,

FilledObs: Filled -> (User + PTS),
transaction_price_obs: Filled -> (User + PTS),
confirmation_number_obs: Filled -> (User + PTS),

Orderds: set Ordered,
OrderedObs: Ordered -> (User + PTS),

// <<both>>
Tickers: set Ticker,
symbol: Ticker -> TickerSymbol,
price: Ticker -> Price,

// messages User
GetTickerInfos: set GetTickerInfo,
GTITickerSymbol: GetTickerInfo -> TickerSymbol,
ep_GTI: GetTickerInfo -> one User,

RequestOrders: set RequestOrder,
ROType: RequestOrder -> TransactionType,
ROAction: RequestOrder -> TransactionAction,
ROTickerSymbol: RequestOrder -> TickerSymbol,
ROVolume: RequestOrder -> Volume,
ep_RO: RequestOrder -> one User,

Cancels: set Cancel,
CTransactionID: Cancel -> TransactionID,
ep_C: Cancel -> one User,

ForwardTIToCustomers: set ForwardTIToCustomer,
FTITCTickerSymbol: ForwardTIToCustomer -> TickerSymbol,
FTITCTickerPrice: ForwardTIToCustomer -> Price,
ep_FTITC: ForwardTIToCustomer -> one User,

TimeExpireds: set TimeExpired,
TETransactionID: TimeExpired -> TransactionID,
ep_TE: TimeExpired -> one User,

OrderCanceleds: set OrderCanceled,
OCTransactionID: OrderCanceled -> TransactionID,
ep_OC: OrderCanceled -> one User,

TransactionCompleteds: set TransactionCompleted,
```

```
  TCTransactionID: TransactionCompleted -> TransactionID,
  ep_TC: TransactionCompleted -> one User,

  // messages PTS
  SendTransactionIDs: set SendTransactionID,
  STIDTransactionID: SendTransactionID -> TransactionIDPTS,
  ep_STID: SendTransactionID  -> one PTS,

  TransactionFilleds: set TransactionFilled,
  TFTransactionID: TransactionFilled -> TransactionIDPTS,
  TFConfirmationNumber: TransactionFilled -> ConfirmationNumber,
  TFVolume: TransactionFilled -> Volume,
  TFPrice:  TransactionFilled -> Price,
  ep_TF: TransactionFilled -> one PTS,

  PTSTransactionCompleteds: set PTSTransactionCompleted,
  PTSTCTransactionID: PTSTransactionCompleted -> TransactionIDPTS,
  ep_PTSTC: PTSTransactionCompleted -> one PTS,

  UpdateTickers: set UpdateTicker,
  UTTickerSymbol: UpdateTicker -> TickerSymbol,
  UTPrice: UpdateTicker -> Price,
  ep_UT: UpdateTicker -> one PTS,

  PTSCancelConfirms: set PTSCancelConfirm,
  PTSCCTransactionID: PTSCancelConfirm -> TransactionIDPTS,
  ep_PTSCC: PTSCancelConfirm -> one PTS,

  SendOrderToPTSs: set SendOrderToPTS,
  SOTPTSAction: SendOrderToPTS -> TransactionAction,
  SOTPTSTickerSymbol: SendOrderToPTS -> TickerSymbol,
  SOTPTSVolume: SendOrderToPTS -> Price,
  ep_SOTPTS: SendOrderToPTS -> one PTS,

  GetTickerInfoFromPTSs: set GetTickerInfoFromPTS,
  GTIFPTSTickerSymbol: GetTickerInfoFromPTS -> TickerSymbol,
  ep_GTIFPTS: GetTickerInfoFromPTS -> one PTS,

  CancelOrders: set CancelOrder,
  COTransactionID: CancelOrder -> TransactionIDPTS,
  ep_CO: CancelOrder -> one PTS
}{
   // domain constraints
   #PTSs=1
   // every user has unique id
   no disj u, u': Users | u.user_id = u'.user_id
   // every ticker has unique ticker symbol
   no disj t, t': Tickers | t.symbol = t'.symbol
```

```
// every transaction has unique internal id and unique id from PTS
no disj t, t': Transactions | t.transaction_id = t'.transaction_id
no disj t, t': Transactions | t.id_pts = t'.id_pts

// non-concept attributes constraints
user_id.UserID in Users
pts_id.PTSID in PTSs

transaction_id.TransactionID in Transactions
id_pts.TransactionIDPTS in Transactions
ticker_symbol.TickerSymbol in Transactions
action.TransactionAction in Transactions
phase.TransactionPhase in Transactions
type.TransactionType in Transactions
status.TransactionStatus in Transactions

time_stamp.Time in Statuss
volume.Volume in Statuss

transaction_price.Price in Filleds
confirmation_number.ConfirmationNumber in Filleds

symbol.TickerSymbol in Tickers
price.Price in Tickers

ROType.TransactionType in RequestOrders
ROAction.TransactionAction in RequestOrders
ROTickerSymbol.TickerSymbol in RequestOrders
ROVolume.Volume in RequestOrders

// concept attributes constraints
(UserTransactions.user_transactions_user in Users) and (user_transactions_user.User in
    UserTransactionss)
(UserTransactions.user_transactions_transaction in Transactions) and (
    user_transactions_transaction.Transaction in UserTransactionss)

(TransactionStatus.transaction_status_status in Statuss) and (transaction_status_status.Status
     in TransactionStatuss)
(TransactionStatus.transaction_status_transaction in Transactions) and (
    transaction_status_transaction.Transaction in TransactionStatuss)

// message attributes and endpoint are in domain state
CTransactionID.TransactionID in Cancels
Cancel.ep_C in User and ep_C.Users in Cancels
#ep_C.Users =< 1

FTITCTickerSymbol.TickerSymbol in ForwardTIToCustomers
FTITCTickerPrice.Price in ForwardTIToCustomers
```

```
ForwardTIToCustomer.ep_FTITC in User and ep_FTITC.Users in ForwardTIToCustomers
#ep_FTITC.Users =< 1

TETransactionID.TransactionID in TimeExpireds
TimeExpired.ep_TE in User and ep_TE.Users in TimeExpireds
#ep_TE.Users =< 1

OCTransactionID.TransactionID in OrderCanceleds
OrderCanceled.ep_OC in User and ep_OC.Users in OrderCanceleds
#ep_OC.Users =< 1

TCTransactionID.TransactionID in TransactionCompleteds
TransactionCompleted.ep_TC in User and ep_TC.Users in TransactionCompleteds
#ep_TC.Users =< 1

STIDTransactionID.TransactionIDPTS in SendTransactionIDs
SendTransactionID.ep_STID in PTS and ep_STID.PTSs in SendTransactionIDs
#ep_STID.PTSs =< 1

TFTransactionID.TransactionIDPTS in TransactionFilleds
TFConfirmationNumber.ConfirmationNumber in TransactionFilleds
TFVolume.Volume in TransactionFilleds
TFPrice.Price in TransactionFilleds
TransactionFilled.ep_TF in PTS and ep_TF.PTSs in TransactionFilleds
#ep_TF.PTSs =< 1

PTSTCTransactionID.TransactionIDPTS in PTSTransactionCompleteds
PTSTransactionCompleted.ep_PTSTC in PTS and ep_PTSTC.PTSs in PTSTransactionCompleted
#ep_PTSTC.PTSs =< 1

UTTickerSymbol.TickerSymbol in UpdateTickers
UTPrice.Price in UpdateTickers
UpdateTicker.ep_UT in PTS and ep_UT.PTSs in UpdateTickers
#ep_UT.PTSs =< 1

PTSCCTransactionID.TransactionIDPTS in PTSCancelConfirms
PTSCancelConfirm.ep_PTSCC in PTS and ep_PTSCC.PTSs in PTSCancelConfirms
#ep_PTSCC.PTSs =< 1

SOTPTSAction.TransactionAction in SendOrderToPTSs
SOTPTSTickerSymbol.TickerSymbol in SendOrderToPTSs
SOTPTSVolume.Price in SendOrderToPTSs
SendOrderToPTS.ep_SOTPTS in PTS and ep_SOTPTS.PTSs in SendOrderToPTSs
#ep_SOTPTS.PTSs =< 1

GTIFPTSTickerSymbol.TickerSymbol in GetTickerInfoFromPTSs
GetTickerInfoFromPTS.ep_GTIFPTS in PTS and ep_GTIFPTS.PTSs in GetTickerInfoFromPTSs
#ep_GTIFPTS.PTSs =< 1
```

```
COTransactionID.TransactionIDPTS in CancelOrders
CancelOrder.ep_CO in PTS and ep_CO.PTSs in CancelOrders
#ep_CO.PTSs =< 1

// cardinality constraints
all user: Users | { #user.user_id = 1 }
all pts: PTSs | { #pts.pts_id = 1 }

all transaction: Transactions | { #transaction.transaction_id = 1 }
all transaction: Transactions | { #transaction.id_pts = 1 }
all transaction: Transactions | { #transaction.ticker_symbol = 1 }
all transaction: Transactions | { #transaction.action = 1 }
all transaction: Transactions | { #transaction.phase = 1 }
all transaction: Transactions | { #transaction.type = 1 }
all transaction: Transactions | { #transaction.status = 1 }

all user_transactions: UserTransactionss | { #user_transactions.user_transactions_user = 1 }
all user_transactions: UserTransactionss | { #user_transactions.user_transactions_transaction
    = 1 }

all transaction_status: TransactionStatuss | { #transaction_status.transaction_status_status =
     1 }
all transaction_status: TransactionStatuss | { #transaction_status.
    transaction_status_transaction = 1 }

all status: Statuss | { #status.time_stamp = 1 }
all status: Statuss | { #status.volume = 1 }

all filled: Filleds | { #filled.transaction_price = 1 }
all filled: Filleds | { #filled.confirmation_number = 1 }

all ticker: Tickers | { #ticker.symbol = 1 }
all ticker: Tickers | { #ticker.price = 1 }

all request_order: RequestOrders | { #request_order.ROType = 1 }
all request_order: RequestOrders | { #request_order.ROAction = 1 }
all request_order: RequestOrders | { #request_order.ROTickerSymbol = 1 }
all request_order: RequestOrders | { #request_order.ROVolume = 1 }

all cancel: Cancels | { #cancel.CTransactionID = 1 }

all forward_ti_to_customer: ForwardTIToCustomers | { #forward_ti_to_customer.FTITCTickerSymbol
     = 1 }
all forward_ti_to_customer: ForwardTIToCustomers | { #forward_ti_to_customer.FTITCTickerPrice
    = 1 }

all time_expired: TimeExpireds | { #time_expired.TETransactionID = 1 }
```

```
    all order_canceled: OrderCanceleds | { #order_canceled.OCTransactionID = 1 }

    all transaction_completed: TransactionCompleteds | { #transaction_completed.TCTransactionID =
        1 }

    all send_transaction_id: SendTransactionIDs | { #send_transaction_id.STIDTransactionID = 1 }

    all transaction_filled: TransactionFilleds | { #transaction_filled.TFTransactionID = 1 }
    all transaction_filled: TransactionFilleds | { #transaction_filled.TFConfirmationNumber = 1 }
    all transaction_filled: TransactionFilleds | { #transaction_filled.TFVolume = 1 }
    all transaction_filled: TransactionFilleds | { #transaction_filled.TFPrice = 1 }

    all pts_transaction_completed: PTSTransactionCompleted | { #pts_transaction_completed.
        PTSTCTransactionID = 1 }
    all update_ticker: UpdateTickers | { #update_ticker.UTTickerSymbol = 1 }
    all update_ticker: UpdateTickers | { #update_ticker.UTPrice = 1 }

    all pts_cancel_confirm: PTSCancelConfirms | { #pts_cancel_confirm.PTSCCTransactionID = 1 }

    all send_order_to_pts: SendOrderToPTSs | { #send_order_to_pts.SOTPTSAction = 1 }
    all send_order_to_pts: SendOrderToPTSs | { #send_order_to_pts.SOTPTSTickerSymbol = 1 }
    all send_order_to_pts: SendOrderToPTSs | { #send_order_to_pts.SOTPTSVolume = 1 }

    all get_ticker_info_from_pts: GetTickerInfoFromPTSs | { #get_ticker_info_from_pts.
        GTIFPTSTickerSymbol = 1 }

    all cancel_order: CancelOrders | { #cancel_order.COTransactionID = 1 }

    // two elements in the same relationship cannot relate same objects
    no disj s, t: TransactionStatuss | { s.transaction_status_status = t.transaction_status_status
         and s.transaction_status_transaction = t.transaction_status_transaction }
    no disj s, t: UserTransactions | { s.user_transactions_user = t.user_transactions_user and s.
        user_transactions_transaction = t.user_transactions_transaction }
}
```

Appendix B

The Behavioural Model of the Online Trading System in Feature Oriented Requriements Modelling

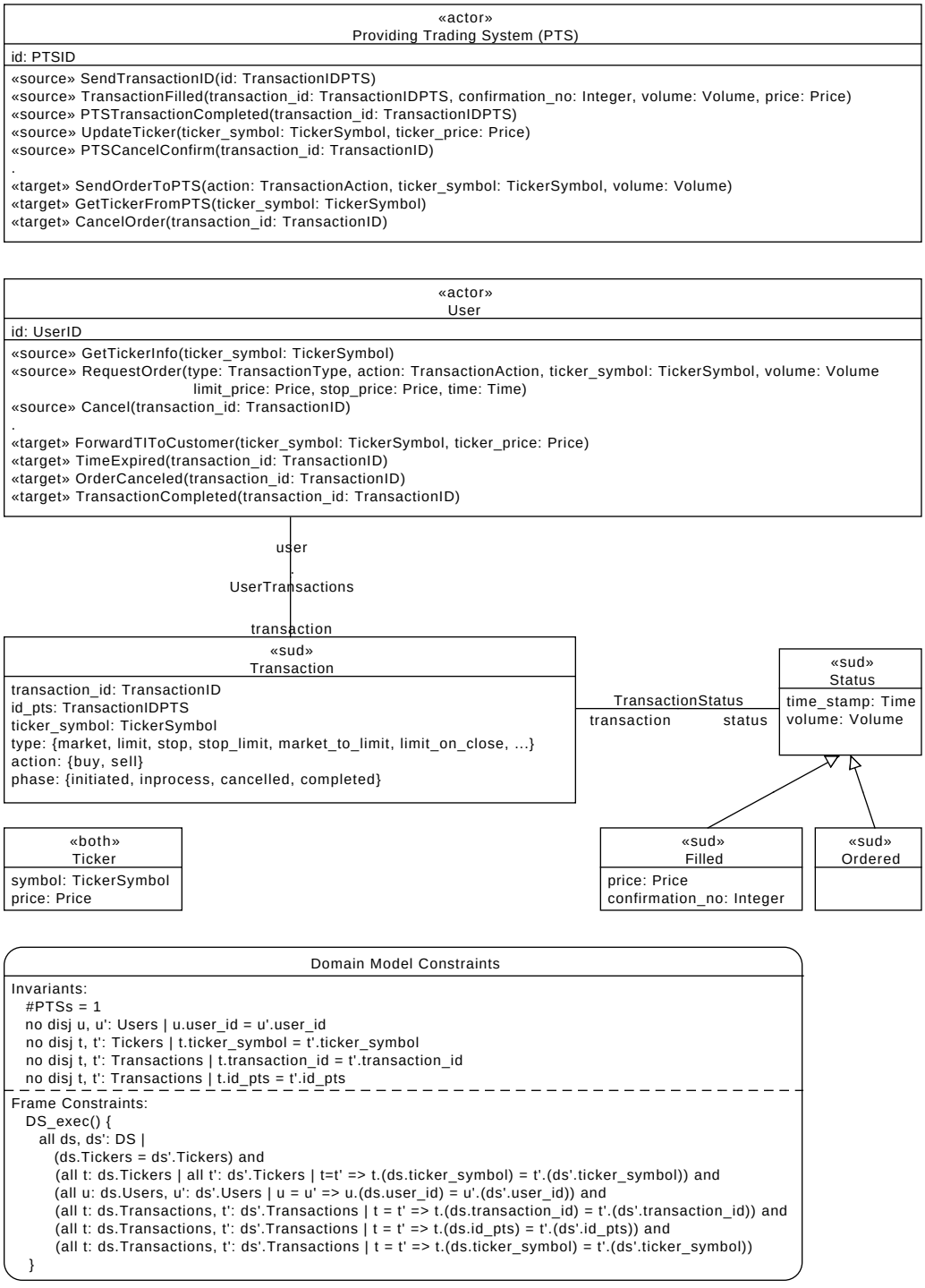

Figure B.1: The FORM domain model of an online trading system

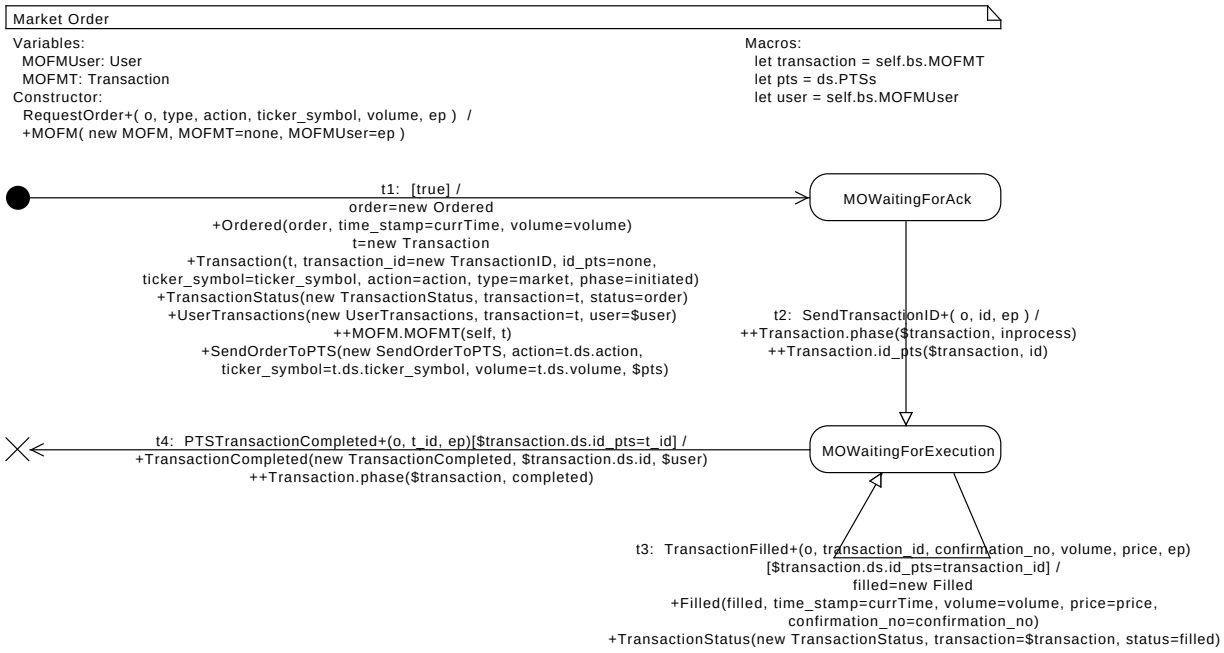

Figure B.2: Market order modelled as a FORM feature model

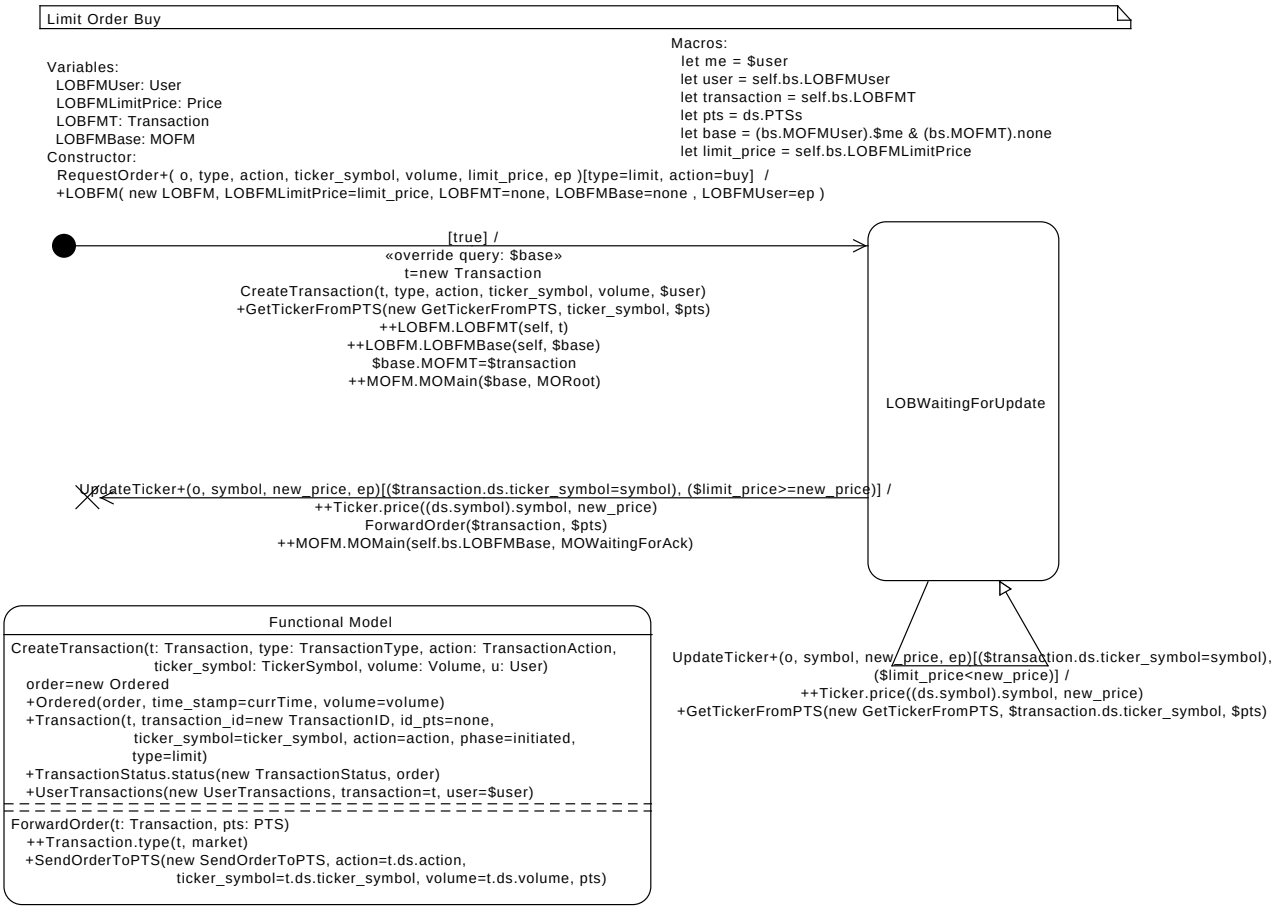

Figure B.3: Buy-limit order modelled as a FORM feature model

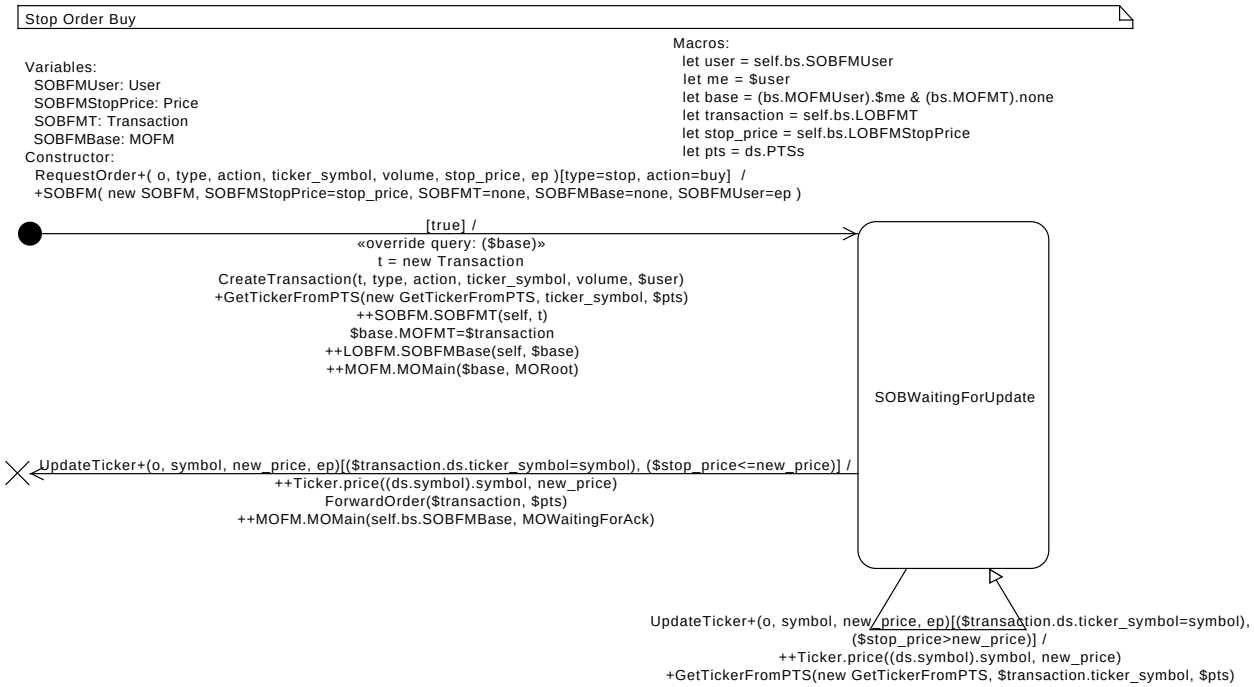

Figure B.4: Buy-stop modelled as a FORM feature model

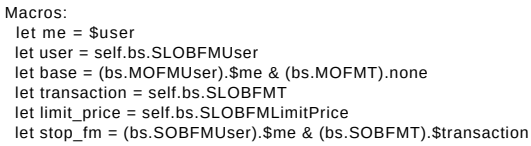

Figure B.5: Buy-stop-limit order modelled as a FORM feature model

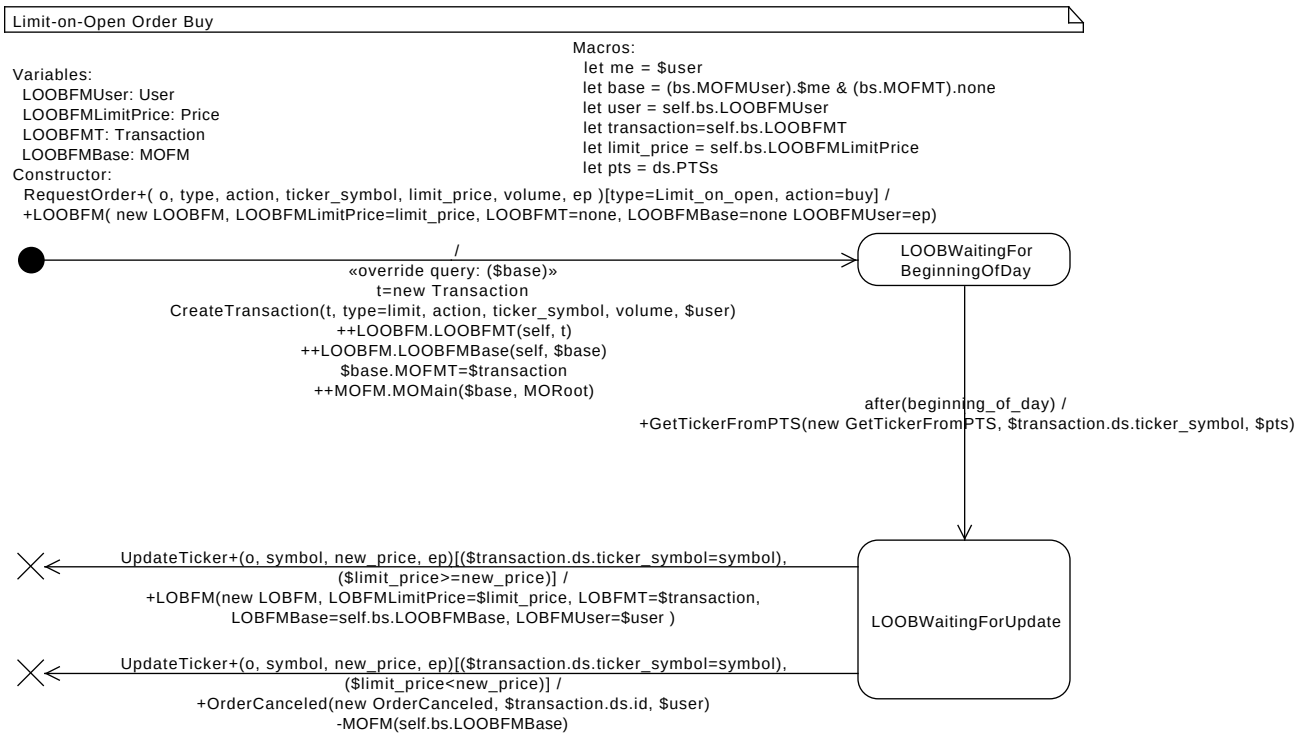

Figure B.6: Limit-on-open order modelled as a FORM feature model

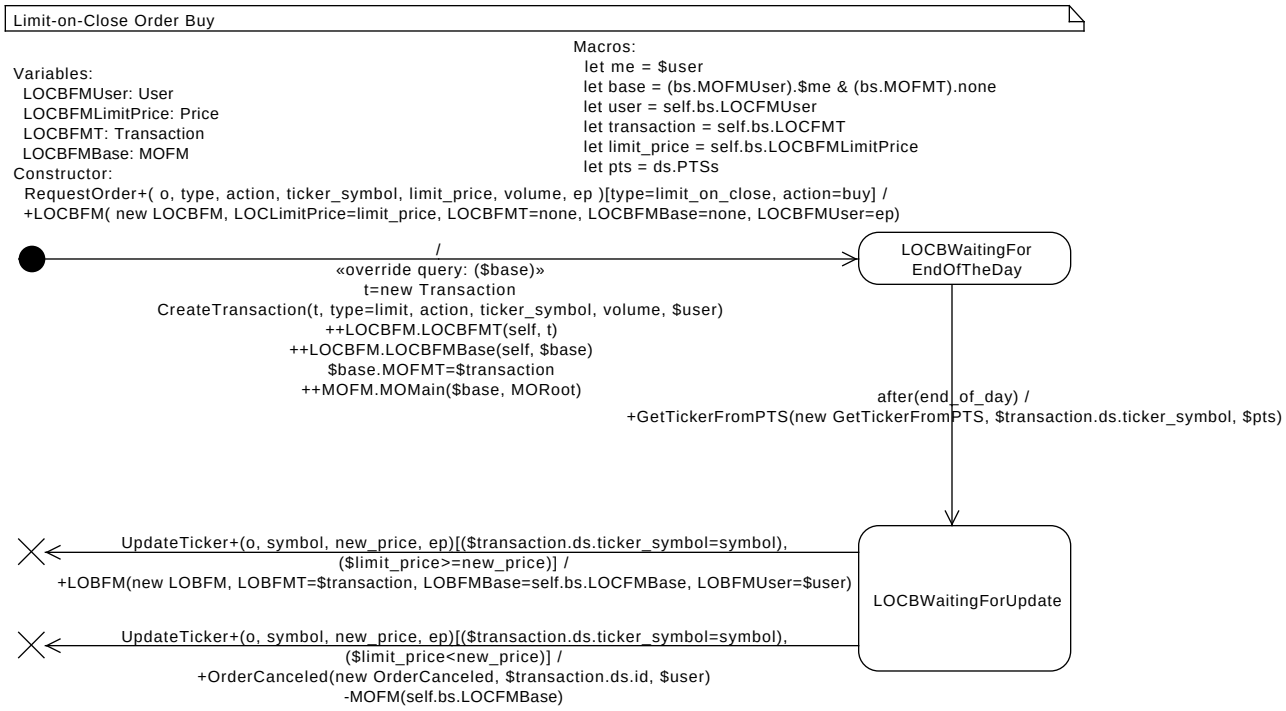

Figure B.7: Limit-on-close order modelled as a FORM feature model

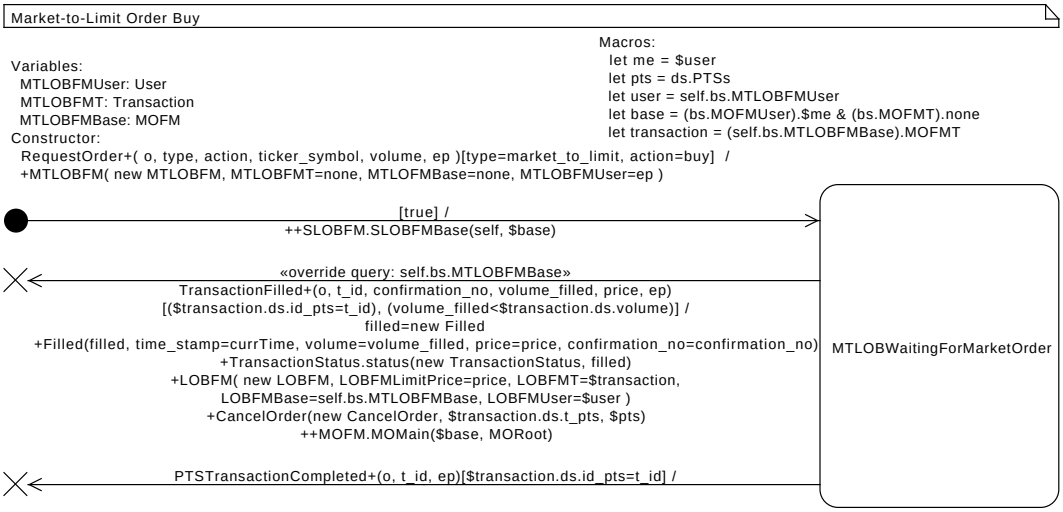

Figure B.8: Market-to-limit order modelled as a FORM feature model

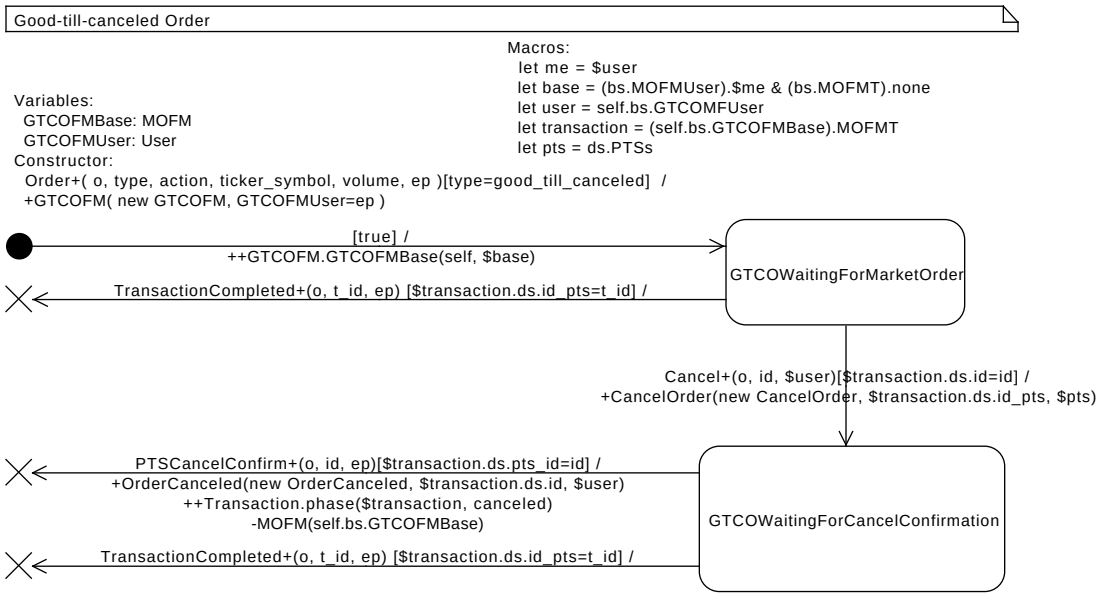

Figure B.9: Good-till-cancelled order modelled as a FORM feature model

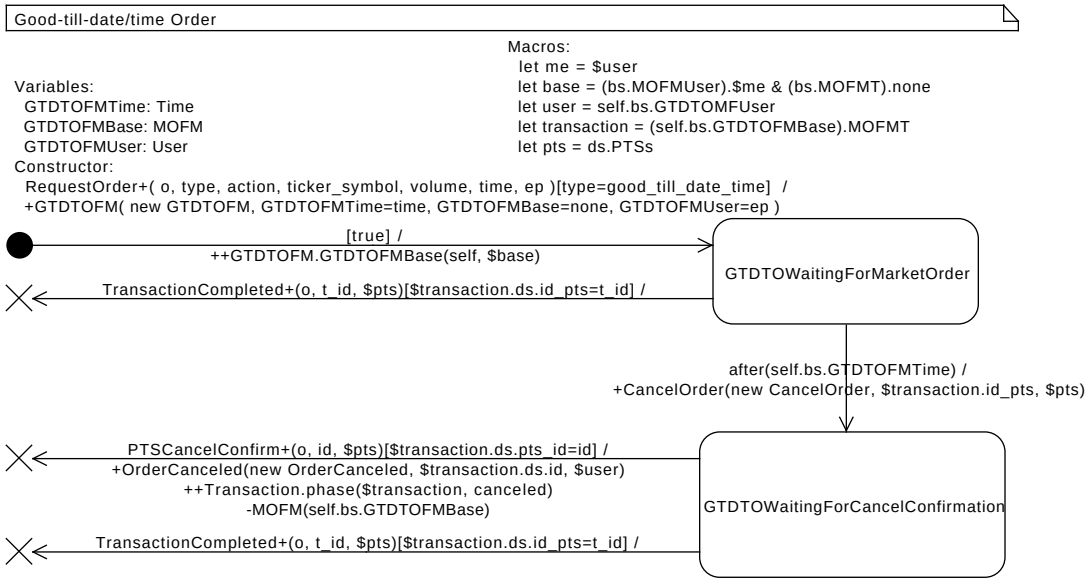

Figure B.10: Good-till-date/time order modelled as a FORM feature model

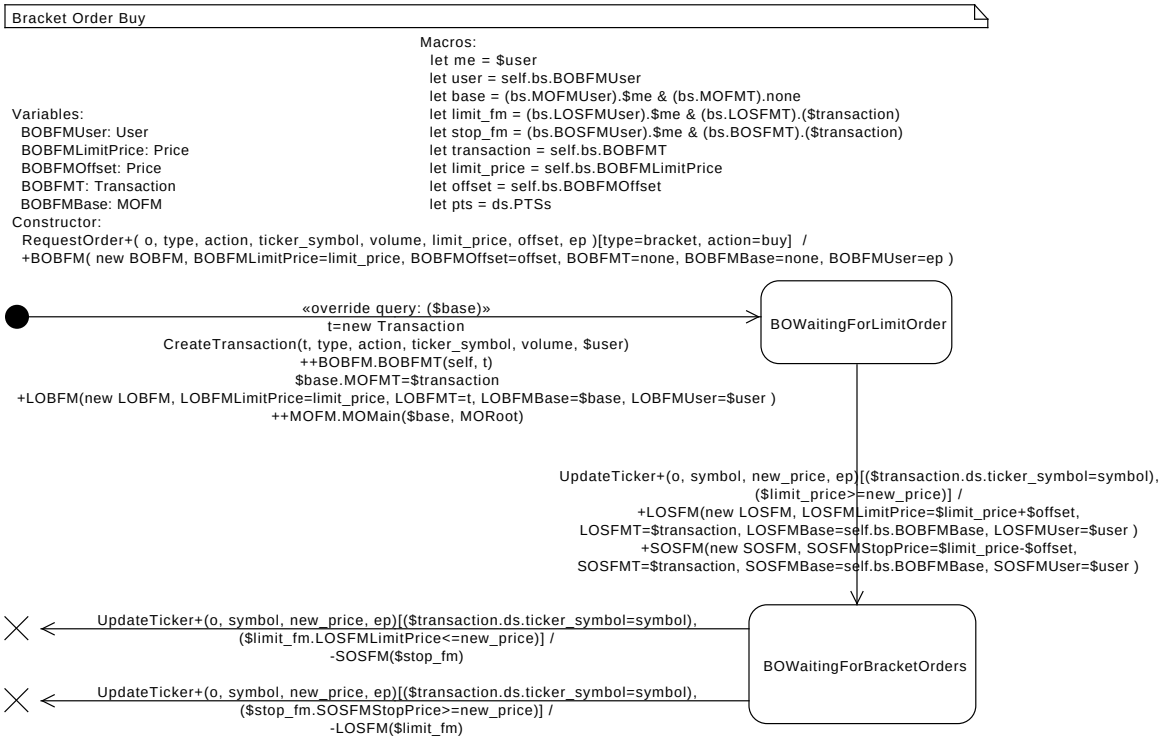

Figure B.11: Bracket order modelled as a FORM feature model