Chapter 3
Modeling and System Specification

Contents

"The beginning is the most important part of the work." —Plato

The term "system specification" is used both for the *process* of deriving the properties of the software system as well as for the *document* that describes those properties. As the system is developed, its properties will change during different stages of its lifecycle, and so it may be unclear which specification is being referred to. To avoid ambiguity we adopt a common meaning: The system specification states what should be valid (true) about the system at the time when the system is delivered to the customer. Specifying system means stating *what* we desire to achieve, *not how* we plan to accomplish it or what has been achieved at an intermediate stage. The focus of this chapter is on describing the system *function*, not its *form*. Chapter 5 will focus on the form, how to build the system.

There are several aspects of specifying the system under development, including:

- Understanding the problem and determining what needs to be specified
- Selecting notation(s) to use for the specification
- Verifying that the specification meets the requirements

Of course, this is not a linear sequence of activities. Rather, as we achieve better understanding of the problem, we may wish to switch to a different notation; also, the verification activity may uncover some weaknesses in understanding the problem and trigger an additional study of the problem at hand.

We have already encountered one popular notation for specification, that is, the UML standard. We will continue using UML and learn some more about it as well as about some other notations. Most developers agree that a single type of system model is not enough to specify any non-trivial system. You usually need several different models, told in different "languages" for different stakeholders. The end user has certain requirements about the system, such as that the system allows him to do his job easier. The business manager may be more concerned about the policies,

rules, and processes supported by the system. Some stakeholders will care about engineering design's details, and others will not. Therefore, it is advisable to develop the system specification as viewed from several different angles, using different notations.

My primary concern here is the developer's perspective. We need to specify what are the resting/equilibrium states and the anticipated perturbations. How does the system appear in an equilibrium state? How does it react to a perturbation and what sequence of steps it goes through to reach a new equilibrium? We already saw that use cases deal with such issues, to a certain extent, although informally. Here, I will review some more precise approaches. This does not necessarily imply formal methods. Some notations are better suited for particular types of problems. Our goal is to work with a certain degree of precision that is amenable to some form of analysis.

The system specification should be derived from the requirements. The specification should accurately describe the system behavior necessary to satisfy the requirements. Most developers would argue that the hardest part of software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging its specification—figuring out what exactly needs to be done. The developer might have misunderstood the customer's needs. The customer may be unsure, and the initial requirements will often be fuzzy or incomplete. I should emphasize again and again that writing the requirements and deriving the specification is not a strictly sequential process. Rather, we must explore the requirements and system specification iteratively, until a satisfactory solution is found. Even then, we may need to revisit and reexamine both if questions arise during the design and implementation.

Although the system requirements are ultimately decided by the customer, the developer needs to know how to ask the right questions and how to systemize the information gathered from the customer. But, what questions to ask? A useful approach would be to be start with a catalogue of *simple representative problems* that tend to occur in every real-world problem. These elementary-building-block problems are called "problem frames." Each can be described in a well-defined format, each has a well-known solution, and each has a well-known set of associated issues. We already made initial steps in Section 2.3.1. In Section 3.3 we will see how complex problems can be made manageable by applying problem frames. In this way, problem frames can help us bridge the gap between system requirements and system specification.

3.1 What is a System?

"All models are wrong, but some are useful." —George E. P. Box

"There is no property absolutely essential to one thing. The same property, which figures as the essence of a thing on one occasion, becomes a very inessential feature upon another." —William James

In Chapter 2 we introduced *system-to-be*, or more accurately the *software-to-be*, as the software product that a software engineer (or a team of engineers) sets out to develop. Apart from the system, the rest of the world ("environment") has been of concern only as far as it interacts with

the system and it was abstracted as a set of actors. By describing different interaction scenarios as a set of use cases, we were able to develop a software system in an incremental fashion.

However, there are some limitations with this approach. First, by considering only the "actors" that the system directly interacts with, we may leave out some parts of the environment that have no direct interactions with the software-to-be but are important to the problem and its solution. Consider, for example, the stock market fantasy league system and the context within which it operates (Figure 1-32). Here, the real-world stock market exchange does not interact with our software-to-be, so it would not be considered an "actor." Conceivably, it would not even be mentioned in any of the use cases, because it is neither an initiating nor a participating actor! I hope that the reader would agree that this is strange—the whole project revolves about a stock exchange and yet the stock exchange may not appear in the system description at all.

Second, starting by focusing on interaction scenarios may not be the easiest route in describing the problem. Use cases describe the sequence of user's (actor) interaction with the system. I already mentioned that use cases are procedural rather than object-oriented. The focus on sequential procedure may not be difficult to begin with, but it requires being on a constant watch for any branching off of the "main success scenario." Decision making (branching points) may be difficult to detect—it may be hard to conceive what could go wrong—particularly if not guided by a helpful representation of the problem structure.

The best way to start conceptual modeling may be with how users and customers prefer to conceptualize their world, because the developer needs to have a great deal of interaction with customers at the time when the problem is being defined. This may also vary across different application domains.

In this chapter I will present some alternative approaches to problem description (i.e., requirements and specification), which may be more involved but are believed to offer easier routes to solving large-scale and complex problems.

3.1.1 World Phenomena and Their Abstractions

The key to solving a problem is in understanding the problem. Because problems are in the real world, we need good abstractions of world phenomena. Good abstractions will help us to represent accurately the knowledge that we gather about the world (that is, the "application domain," as it relates to our problem at hand). In object-oriented approach, key abstractions are objects and messages and they served us well in Chapter 2 in understanding the problem and deriving the solution. We are not about to abandon them now; rather, we will broaden our horizons and perhaps take a slightly different perspective.

Usually we partition the world in different parts (or regions, or domains) and consider different phenomena, see Figure 3-1. A *phenomenon* is a fact, or object, or occurrence that appears or is perceived to exist, or to be present, or to be the case, when you observe the world or some part of it. We can distinguish world phenomena by different criteria. Structurally, we have two broad categories of phenomena: *individuals* and *relations* among individuals. Logically, we can distinguish *causal* vs. *symbolic* phenomena. In terms of behavior, we can distinguish *deterministic* vs. *stochastic* phenomena. Next I describe each kind briefly.

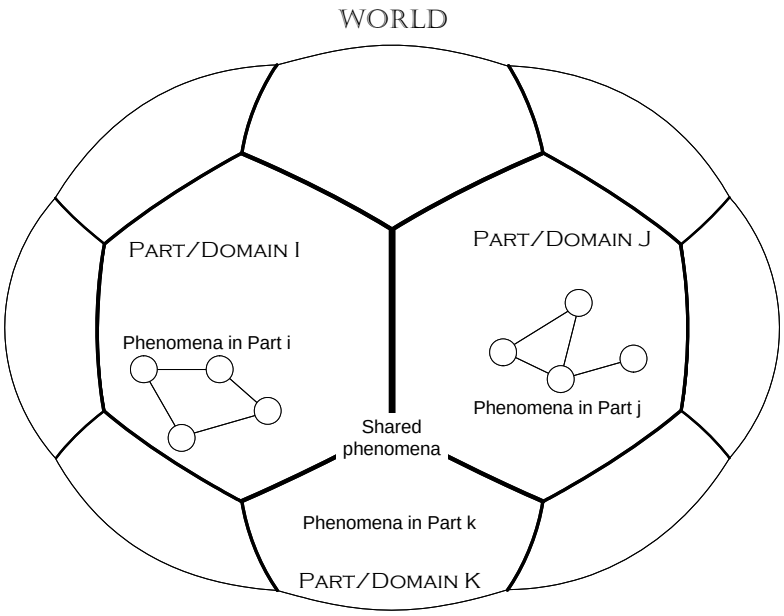


Figure 3-1: World partitioning into domains and their phenomena.

I should like to emphasize that this is only one possible categorization, which seems suitable for software engineering; other categorizations are possible and have been proposed. Moreover, any specific identification of world phenomena is evanescent and bound to become faulty over time, regardless of the amount of effort we invest in deriving it. I already mentioned in Section 1.1.1 the effect of the second law of thermodynamics. When identifying the world phenomena, we inevitably make approximations. Certain kinds of information are regarded as important and the rest of the information is treated as unimportant and ignored. Due to the random fluctuations in the nature and society, some of the phenomena that served as the basis for our separation of important and unimportant information will become intermingled thus invalidating our original model. Hence the ultimate limits to what our modeling efforts can achieve.

Individuals

An individual is something that can be named and reliably distinguished from other individuals. Decisions to treat certain phenomena as individuals are not objective—they depend on the problem at hand. It should be clear by now that the selected level of abstraction is relative to the observer. We choose to recognize just those individuals that are useful to solving the problem and are practically distinguishable. We will choose to distinguish three kinds of individual: events, entities, and values.

♦ An **event** is an individual happening, occurring at a particular point in time. Each event is *indivisible* and *instantaneous*, that is, the event itself has no internal structure and takes no time to happen. Hence, we can talk about "before the event" and "after the event," but not about "during the event." An example event is placing a trading order; another example event is executing a stock trading transaction; yet another example is posting a stock price quotation. Further discussion of events is in Section 3.1.3.

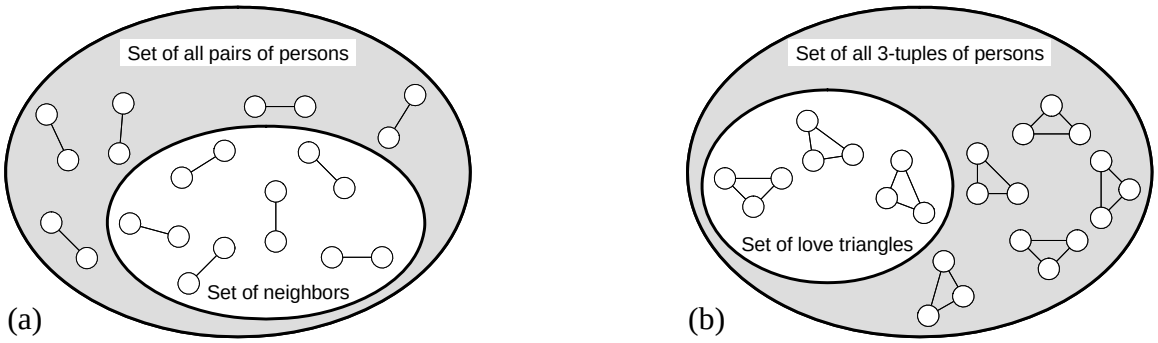


Figure 3-2: Example relations: *Neighbors*(Person_i, Person_j) and *InLoveTriangle*(Person_i, Person_j, Person_k).

♦ An **entity** is an individual with distinct existence, as opposed to a quality or relation. An entity persists over time and can change its properties and states from one point in time to another. Some entities may initiate events; some may cause spontaneous changes to their own states; some may be passive.

Software objects and abstract concepts modeled in Chapter 2 are entities. But entities also include real-world objects. The entities are determined by what part of the world is being modeled. A *financial-trader* in our investment assistant case study (Section 1.3.2) is an entity; so is his *investment-portfolio*; a *listed-stock* is also an entity. They belong to entity classes *trader*, *portfolio*, and *stock*, respectively.

♦ A **value** is an intangible individual that exists outside time and space, and is not subject to change. The values we are interested in are such things as numbers and characters, represented by symbols. For example, a value could be the numerical measure of a quantity or a number denoting amount on some conventional scale, such as 7 kilograms.

In our case study (Section 1.3.2), a particular stock *price* is a number of monetary units in which a stock share is priced—and is therefore a value. Examples of value classes include *integer*, *character*, *string*, and so on.

Relations

> "I have an infamously low capacity for visualizing relationships, which made the study of geometry and all subjects derived from it impossible for me." —Sigmund Freud

We say that individuals are in *relation* if they share a certain characteristic. To define a relation, we also need to specify how many individuals we consider at a time. For example, for any pair of people, we could decide that they are neighbors if their homes are less than 100 meters apart from each other. Given any two persons, Person_i and Person_j, if they pass this test then the relation holds (is true); otherwise it does not hold (is false). All pairs of persons that pass the test are said to be in the relation *Neighbors*(Person_i, Person_j). The pairs of persons that are neighbors form a subset of all pairs of persons as shown in Figure 3-2(a).

Relations need not be established on pairs of individuals only. We can consider any number of individuals and decide whether they are in a relation. The number n of considered individuals can be any positive integer $n \geq 2$ and it must be fixed for every test of the relation; we will call it an *n-tuple*. We will write relation as *RelationName*($Individual_1$, …, $Individual_n$). When one of the

individuals remains constant for all tests of the relation, we may include its name in the relation's name. For example, consider the characteristic of wearing eyeglasses. Then we can test whether a Person_i is in relation *Wearing*(Person_i, Glasses), which is a subset of all persons. Because Glasses remain constant across all tests, we can write *WearingGlasses*(Person_i), or simply *Bespectacled*(Person_i). Consider next the so-called "love triangle" relation as an example for n = 3. Obviously, to test for this characteristic we must consider exactly three persons at a time; not two, not four. Then the relation *InLoveTriangle*(Person_i, Person_j, Person_k) will form a set of all triplets (3-tuples) of persons for whom this characteristic is true, which is a subset of all 3-tuples of persons as shown in Figure 3-2(b). A formal definition of relation will be given in Section 3.2.1 after presenting some notation.

We will consider three kinds of relations: states, truths, and roles.

♦ A **state** is a relation among individual entities and values, which can change over time. I will describe states in Section 3.1.2, and skip them for now.

♦ A **truth** is a fixed relation among individuals that cannot possibly change over time. Unlike states, which change over time, truths remain constant. A bit more relaxed definition would be to consider the relations that are invariable on the time-scale that we are interested in. Example time-scales could be project duration or anticipated product life-cycle. When stating a truth, the individuals are always values, and the truth expresses invariable facts, such as *GreaterThan*(5, 3) or *StockTickerSymbol*("Google, Inc.," "GOOG"). It is reasonably safe to assume that company stock symbols will not change (although mergers or acquisitions may affect this!).

♦ A **role** is a relation between an event and individual that participate in it in a particular way. Each role expresses what you might otherwise think of as one of the "arguments" (or "parameters") of the event.

Causal vs. Symbolic Phenomena

♦ **Causal** phenomena are events, or roles, or states relating entities. These are causal phenomena because they are directly produced or controlled by some entity, and because they can give rise to other phenomena in turn.

♦ **Symbolic** phenomena are values, and truths and states relating only values. They are called symbolic because they are used to symbolize other phenomena and relationships among them. A symbolic state that relates values—for example, the data content of a disk record—can be changed by external causation, but we do not think of it as causal because it can neither change itself nor cause change elsewhere.

Deterministic vs. Stochastic Phenomena

♦ **Deterministic** phenomena are the causal phenomena for which the occurrence or non-occurrence can be established with certainty.

♦ **Stochastic** phenomena are the causal phenomena that are governed by a random distribution of probabilities.

3.1.2 States and State Variables

A state describes what is true in the world at each particular point in time. The state of an individual represents the cumulative results of its behavior. Consider a device, such as a digital video disc (DVD) player. How the device reacts to an input command depends not only upon that input, but also upon the internal state that the device is currently in. So, if the "PLAY" button is pushed on a DVD player, what happens next will depend on various things, such as whether or not the player is turned on, contains a disc, or is already playing. These conditions represent different states of a DVD player.

By considering such options, we may come up with a list of all states for a DVD player, like this:

State 1: *NotPowered* (the player is not powered up)
State 2: *Powered* (the player is powered up)
State 3: *Loaded* (a disc is in the tray)
State 4: *Playing*

We can define **state** more precisely as a *relation* on a set of objects, which simply selects a subset of the set. For the DVD player example, what we wish to express is "The DVD player's power is off." We could write *Is*(DVDplayer, NotPowered) or *IsNotPowered*(DVDplayer). We will settle on this format: *NotPowered*(DVDplayer). *NotPowered*(*x*) is a subset of DVD players *x* that are not powered up. In other words, *NotPowered*(*x*) is true if *x* is currently off. Assuming that one such player is the one in the living room, labeled as DVDinLivRm, then *NotPowered*(DVDinLivRm) holds true if the player in the living room is not powered up.

Upon a closer examination, we may realize that the above list of states implies that a non-powered-up player never contains a disc in the tray. If you are charged to develop software for the DVD player, you must clarify this. Does this mean that the disc is automatically ejected when the power-off button is pushed? If this is not the case or the issue is yet unresolved, we may want to redesign our list of DVD player states as:

State 1: *NotPoweredEmpty* (the player is not powered up and it contains no disc)
State 2: *NotPoweredLoaded* (the player is not powered up but a disc is in the tray)
State 3: *PoweredEmpty* (the player is powered up but it contains no disc)
State 4: *PoweredLoaded* (the player is powered up and a disc is in the tray)
State 5: *Playing*

At this point one may realize that instead of aggregate or "global" system states it may be more elegant to discern different parts (sub-objects) of the DVD player and, in turn, consider the state of each part (Figure 3-3). Each part has its "local" states, as in this table

System part (Object)	State relations
Power button	{*Off, On*}
Disc tray	{*Empty, Loaded*}
Play button	{*Off, On*}
…	…

Note that the relation *Off*(*b*) is defined on the set of buttons. Then these relations may be true: *Off*(PowerButton) and *Off*(PlayButton). Similar holds for *On*(*b*).

Given the states of individual parts, how can we define the state of the whole system? Obviously, we could say that the aggregate system state is defined by the states of its parts. For example, one

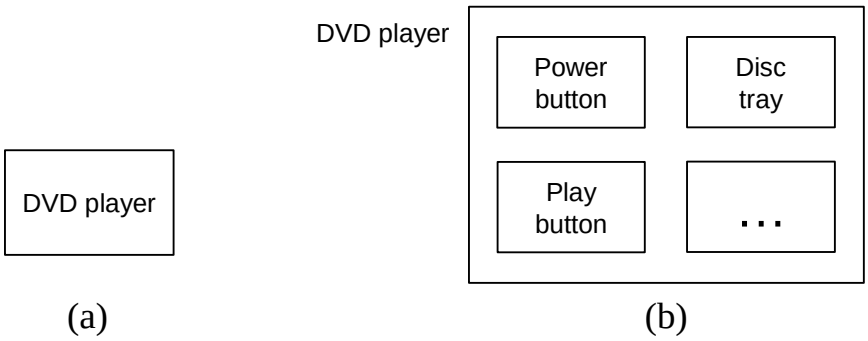


Figure 3-3: Abstractions of a DVD player at different levels of detail: (a) The player as a single entity. (b) The player seen as composed of several entities.

state of the DVD player is { *On*(PowerButton), *Empty*(), *Off*(PlayButton), … }. Note that the relation *Empty*() is left without an argument, because it is clear to which object it refers to. In this case we could also write *Empty* without parentheses. The arrangement of the relations in this "state tuple" is not important as long as it is clear what part each relation refers to.

The question now arises, is every combination of parts' states allowed? Are these parts independent of each other or there are constraints on the state of one part that are imposed by the current states of other parts? Some states of parts of a composite domain may be *mutually exclusive*. Going back to the issue posed earlier, can the disc tray be in the "loaded" state when the power button is in the "off" state? Because these are parts of the same system, we must make explicit any *mutual dependencies of the parts' states*. We may end up with a list of valid system state tuples that does not include all possible tuples that can be constructed.

Both representations of a system state (single aggregate state vs. tuple of parts' states) are correct, but their suitability depends on what kind of details you care to know about the system. In general, considering the system as a set of parts that define state tuples presents a cleaner and more modular approach than a single aggregate state.

In software engineering, we care about the visible aspects of the software system. In general, visible aspects do not necessarily need to correspond to "parts" of the system. Rather, they are any observable qualities of the system. For example, domain-model attributes identified in Section 2.5 represent observable qualities of the system. We call each observable quality a **state variable**. In our first case-study example, variables include the lock and the bulb. Another variable is the counter of the number of attempts at opening the lock. Yet another variable is the amount of timer that counts down the time elapsed since the lock was open, to support auto-lock functionality. The state variables of our system can be summarized as in this table

Variable	State relations
Door lock	{*Disarmed, Armed*}
Bulb	{*Lit, Unlit*}
Counter of failed attempts	{0, 1, …, maxNumOfAttempts}
Auto-lock timer	{0, 1, …, autoLockInterval}

In case of multiple locks and/or bulbs, we have a different state variable for every lock/bulb, similar to the above example of DVD player buttons. So, the state relations for backyard and front door locks could be defined as *Disarmed*(Backyard) and *Disarmed*(Front).

The situation with numeric relations is a bit trickier. We could write 2(Counter) to mean that the counter is currently in state "2," but this is a bit awkward. Rather, just for the sake of convenience I will write *Equals*(Counter, 2) and similarly *Equals*(Timer, 3).

System state is defined as a tuple of state variables containing any valid combination of state relations. State is an aggregate representation of the system characteristics that we care to know about *looking from outside of the system*. For the above example, an example state tuple is: { *Disarmed*(Front), *Lit*, *Armed*(Backyard), *Equals*(Counter, 0), *Equals*(Timer, 0) }.

One way to classify states is by what the object is doing in a given state:

- A state is a *passive quality* if the object is just waiting for an event to happen. For the DVD player described earlier, such states are "Powered" and "Loaded."
- A state is an *active quality* if the object is executing an activity. When the DVD player is in the "Playing" state it is actively playing a disc.

A combination of these options is also possible, i.e., the object may be executing an activity and also waiting for an event.

The movements between states are called *transitions* and are most often caused by events (described in Section 3.1.3). Each state transition connects two states. Usually, not all pairs of states are connected by transitions—only specific transitions are permissible.

Example 3.1 Identifying Stock Exchange States (First Attempt)

Consider our second case study on an investment assistant system (Section 1.3.2), and suppose that we want to identify the states of the stock exchange. There are many things that we can say about the exchange, such as where it is located, dimensions of the building, the date it was built, etc. But, what properties we care to know as it relates to our problem? Here are some candidates:

- What are the operating hours and is the exchange currently "open" or "closed?"
- What stocks are currently listed?
- For each listed stock, what are the quoted price (traded/bid/ask) and the number of offered shares?
- What is the current overall trading volume?
- What is the current market index or average value?

The state variables can be summarized like so:

Variable	State relations
Operating condition (or gate condition)	{*Open, Closed*}
i^{th} stock price	any positive real number∗
i^{th} stock number of offered shares	{0, 1, 2, 3, …}
Trading volume	{0, 1, 2, 3, …}
Market index/average	any positive real number∗

The asterisk∗ in the table indicates that the prices are quoted up to a certain number of decimal places and there is a reasonable upper bound on the prices. In other words, this is a finite set of finite values. Obviously, this system has a great many of possible states, which is, nonetheless, finite. An improvised graphical representation is shown in Figure 3-4. (UML standard symbols for state diagrams are described later in Section 3.2.2.)

An example state tuple is: { *Open*, *Equals*(Volume, 783014), *Equals*(Average, 1582), *Equals*(Price_1, 74.52), *Equals*(Shares_1, 10721), *Equals*(Price_2, 105.17), *Equals*(Shares_2, 51482), … }. Note that the price and number of shares must be specified for all the listed stocks.

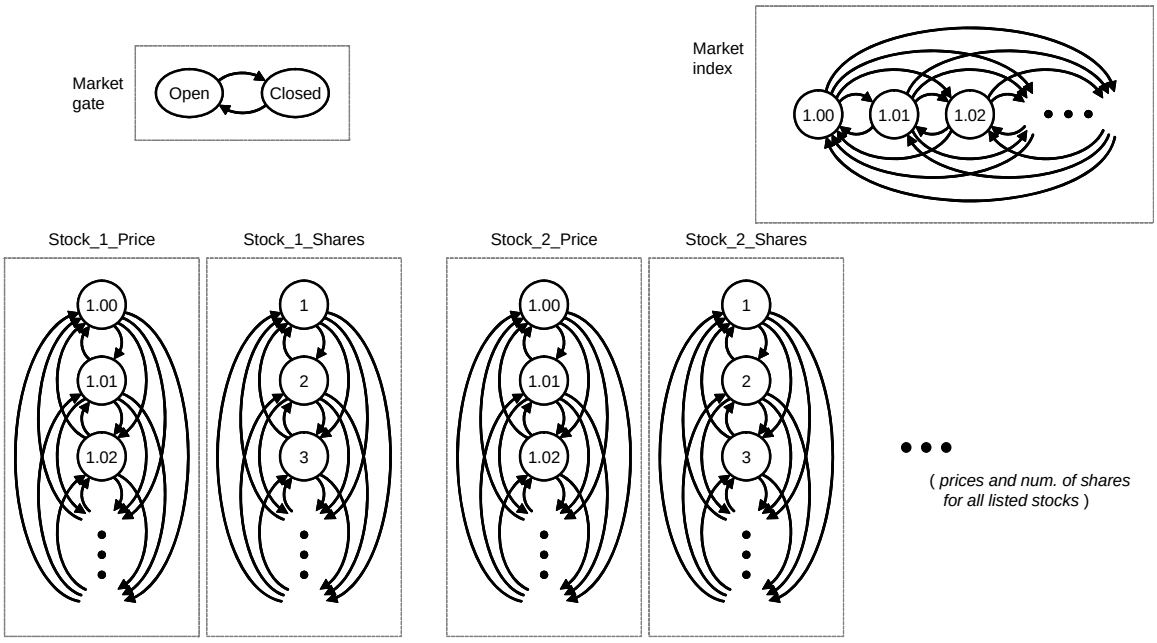
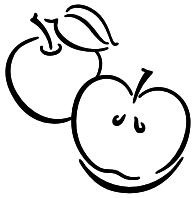


Figure 3-4: Graphical representation of states for Example 3.1. The arrows indicate the permissible paths for transitioning between different states.

As the reader should know by now, the selection of state phenomena depends on the observer and observer's problem at hand. An alternative characterization of a market state is presented later in Example 3.2.

Observables vs. Hidden Variables

States Defined from Observable Phenomena



State is an abstraction, and as such it is subjective—it depends on who is making the abstraction. There are no "objective states"—every categorization of states is relative to the observer. Of course, the same observer can come up with different abstractions. The observer can also *define* new states based on observable phenomena; such states are directly observed. Consider, for example, a fruit states: "green," "semiripe," "ripe," "overripe," and "rotten." The state of "ripeness" of a fruit is defined based on observable parameters such as its skin color and texture, size, scent, softness on touch, etc. Similarly, a "moving" state of an elevator is defined by observing its position over subsequent time moments and calculating the trend.

For the auto-lock timer discussed earlier, we can define the states "CountingDown" and "Idle" like so:

CountingDown(Timer) $\stackrel{\Delta}{=}$ The relation *Equals*(Timer, τ) holds true for τ decreasing with time

Idle(Timer) $\stackrel{\Delta}{=}$ The relation *Equals*(Timer, τ) holds true for τ remaining constant with time

The symbol $\stackrel{\Delta}{=}$ means that this is a defined state.

Example 3.2 Identifying Stock Exchange States (Second Attempt)

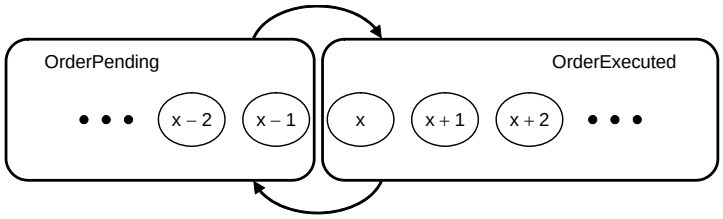


Figure 3-5: Graphical representation of states for Example 3.2. Microstates from Figure 3-4 representing the number of offered shares are aggregated into two macrostates.

Let us revisit Example 3.1. Upon closer examination, one may conclude that the trader may not find very useful the variables identified therein. In Section 1.3.2, we speculated that what trader really cares about is to know if a trading opportunity arises and, once he places a trading order, tracking the status of the order. Let us assume that the trading decision will be made based on the trending direction of the stock price. Also assume that, when an upward trend of Stock_i's price triggers a decision to buy, a market order is placed for *x* shares of Stock_i. To summarize, the trader wants to represent the states of two things:

- "Stock tradability" states ("buy," "sell," "hold") are defined based on considering a time window of recent prices for a given stock and interpolating a line. If the line exhibits an upward trend, the stock state is *Buy*. The states *Sell* and *Hold* are decided similarly. A more financially astute trader may use some of the technical analysis indicators (e.g., Figure 1-23), instead of the simple regression line.
- "Order status" states ("pending" vs. "executed") are defined based on whether there are sufficient shares offered so the buying transaction can be carried out. We have to be careful here, because a selling transaction can be executed only if there are willing buyers. So, the buy and sell orders have the same states, defined differently.

Then the trader could define the states of the market as follows:

Buy $\stackrel{\Delta}{=}$ The regression line of the relation *Equals*(Price_i(*t*), *p*), for $t = t_{current} - Window, \ldots, t_{current} - 2, t_{current} - 1, t_{current}$, has a positive slope

Sell $\stackrel{\Delta}{=}$ The regression line of the relation *Equals*(Price_i(*t*), *p*), for $t = t_{current} - Window, \ldots, t_{current}$, has a negative slope

Hold $\stackrel{\Delta}{=}$ The regression line of the relation *Equals*(Price_i(*t*), *p*), for $t = t_{current} - Window, \ldots, t_{current}$, has a zero slope

SellOrderPending $\stackrel{\Delta}{=}$ The relation *Equals*(Shares_i, *y*) holds true for all values of *y* less than *x*

SellOrderExecuted $\stackrel{\Delta}{=}$ The relation *Equals*(Shares_i, *y*) holds true for all values of *y* greater than or equal to *x*

What we did here, essentially, is to group a large number of detailed states from Example 3.1 into few aggregate states (see Figure 3-5). These grouped states help simplify the trader's work.

It is possible to discern further nuances in each of these states. For example, two sub-states of the state *Sell* could be distinguished as when the trader should sell to avert greater loss vs. when he may wish to take profit at a market top. The most important point to keep in mind is the trader's goals and strategies for achieving them. This is by no means the only way the trader could view the market. A more proficient trader may define the states in terms of long vs. short trading positions (see Section 1.3.2, Figure 1-22). Example states could be:

GoLong – The given stock is currently suitable for taking a long position

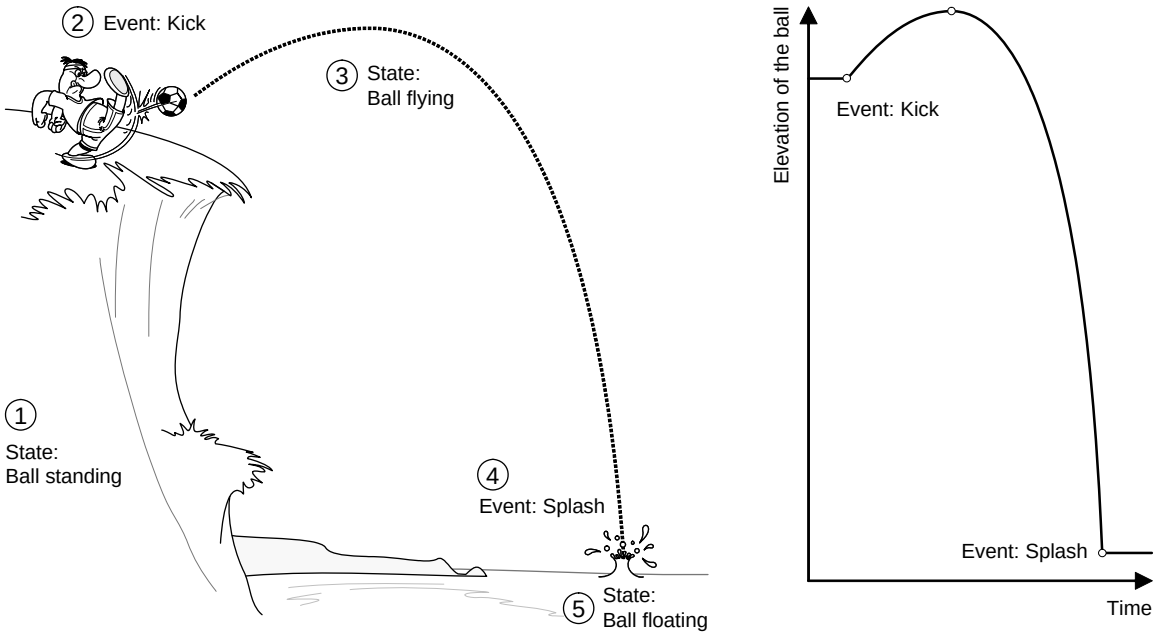


Figure 3-6: Events take place at transitions between the states.

GoShort – The given stock is currently suitable for taking a long position

GoNeutral – The trader should hold or avoid the given stock at this time

---

The states that are directly observable at a given level of detail (coarse graining) will be called **microstates**. A group of microstates is called a **macrostate** (or superstate). The states defined in Example 3.2 are macrostates.

Sometimes our abstraction may identify simultaneous (or **concurrent**) activities that object executes in a given state. For example, when the DVD player is in the "Playing" state it may be simultaneously playing a disc (producing video output) and updating the time-progress display.

Section 3.2.2 describes UML state machine diagrams as a standardized graphical notation for representing states and transitions between them.

3.1.3 Events, Signals, and Messages

Event definition requires that events are indivisible—any happening (or performance, or action) that has an internal time structure must be regarded as two or more events. The motivation for this restriction is to avoid having intermediate states: an event represents a sharp boundary between two different states. We also need to assume that no two events occur simultaneously. All events happen sequentially, and between successive events there are intervals of time in which nothing happens—that is, there are no events. Events and intervals alternate: each event ends one interval and begins another. Consider the example in Figure 3-6. By examining the time diagram we partition time into intervals ("states") and identify what point ("event") separates two intervals. Then we name the resulting five phenomena as shown in Figure 3-6. We cannot have an

uninterrupted sequence of events—this would simply be a wrong model and would require refining the time scale to identify the intervals between successive events.

The developer may need to make a choice of what to treat as a single event. Consider the home-access control case study (Section 1.3.1). When the tenant is punching in his identification key, should this be treated as a single event, or should each keystroke be considered a different event? The answer depends on whether your problem statement requires you to treat it one way or another. Are there any exceptions that are relevant to the problem, which may arise between different keystrokes? If so, then we need to treat each keystroke as an event.

The reader may wonder about the relationship between events and messages, or operations in object-oriented approach. The notion of event as defined above is more general, because it is not limited to object orientation. The notion of *message* implies that a signal is sent from one entity to another. Unlike a message, an *event* is something that happens—it may include one or more individuals but it is not necessarily *directed* from one individual to another. Events just mark transitions between successive states. The advantage of this view is that we can avoid specifying processing detail at an early stage of problem definition. Use case analysis (Section 2.4.3) is different in that it requires making explicit the sequential processing procedure ("scenarios"), which leads to system operations.

Another difference is that events always signify state change—even for situations where system remains in the same state, there is an explicit description of an event and state change. Hence, events depend on how the corresponding state set is already defined. On the other hand, messages may not be related to state changes. For example, an operation that simply retrieves the value of an object attribute (known as accessor operation) does not affect the object's state.

Example events:

listStock – this event marks that it is first time available for trading – marks transition between price states; marks a transition between number-of-shares-available states

splitStock – this event marks a transition between price states; marks transition between number-of-shares-available states

submitOrder – this event marks a transition between the states of a trading order; also marks a transition between price states (the indicative price of the stock gets updated); also marks a transition between number-of-shares-available states, in case of a sell-order

matchFound – this event marks a transition between the states of a trading order when a matching order(s) is(are) found; also marks a transition between price states (the traded price of the stock gets updated); also marks a transition between number-of-shares-available states

The above events can also mark change in "trading volume" and "market index/average." The reader may have observed that event names are formed as verb phrases. The reason for this is to distinguish events from states. Although this is reminiscent of messages in object-oriented approach, events do not necessarily correspond to messages, as already discussed earlier.

Example 3.3 Identifying Stock Exchange Events

Consider Example 3.2, where the states *Buy*, *Sell*, or *Hold*, are defined based on recent price movements. The events that directly lead to transitioning between these states are order placements by other traders. There may be many different orders placed until the transition happens, but we view the

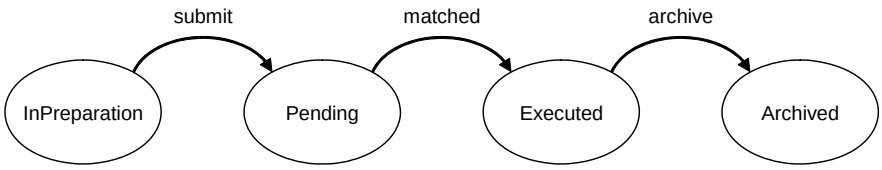


Figure 3-7: Graphical representation of events marking state transitions of a trading order.

transitioning as an indivisible event—the moment when the regression line slope exceeds a given threshold value. The events can be summarized like so:

Event	Description
trade	Causes transition between stock states *Buy*, *Sell*, or *Hold*
submit	Causes transition between trading-order states *InPreparation* → *OrderPending*
matched	Causes transition between trading-order states *OrderPending* → *OrderExecuted*
…	…
…	…

The events marking a trading order transitions are shown in Figure 3-7. Other possible events include *bid* and *offer*, which may or may not lead to transitions among the states of a trading order. We will consider these in Section 3.2.2.

3.1.4 Context Diagrams and Domains

Now that we have defined basic phenomena, we can start the problem domain analysis by placing the planned system in a context—the environment in which it will work. For this we use *context diagrams*, which are essentially a bit more than the commonplace "block diagrams." Context diagrams are not part of UML; they were introduced by Michael Jackson [1995] based on the notation dating back to structured analysis in 1970s. The context diagram represents the *context of the problem* that the developer sets out to solve. The block diagrams we encountered in Figure 1-20(b) and Figure 1-32 are essentially context diagrams. Based on the partitioning in Figure 3-1, we show different domains as rectangular boxes and connect them with lines to indicate that they share certain phenomena. Figure 3-8 is Figure 1-20(b) redrawn as a context diagram, with some details added. Our system-to-be, labeled "machine," subsumes the broker's role and the figure also shows abstract concepts such as portfolio, trading order, and i^{th} stock. Jackson uses the term "machine" to avoid the ambiguities of the word "system," some of which were discussed in Section 2.4.2. We use all three terms, "system-to-be," "software-to-be," and "machine."

A context diagram shows parts of the world (Figure 3-1) that are relevant to our problem and only the relevant parts. Each box in a context diagram represents a different domain. A **domain** is a part of the world that can be distinguished because it is conveniently considered as a whole, and can be considered—to some extent—separately from other parts of the world. Each domain is a different subject matter that appears in the description of the problem. A domain is described by the phenomena that exist or occur in it. In every software development problem there are at least two domains: the *application domain* (or environment, or real world—what is given) and the

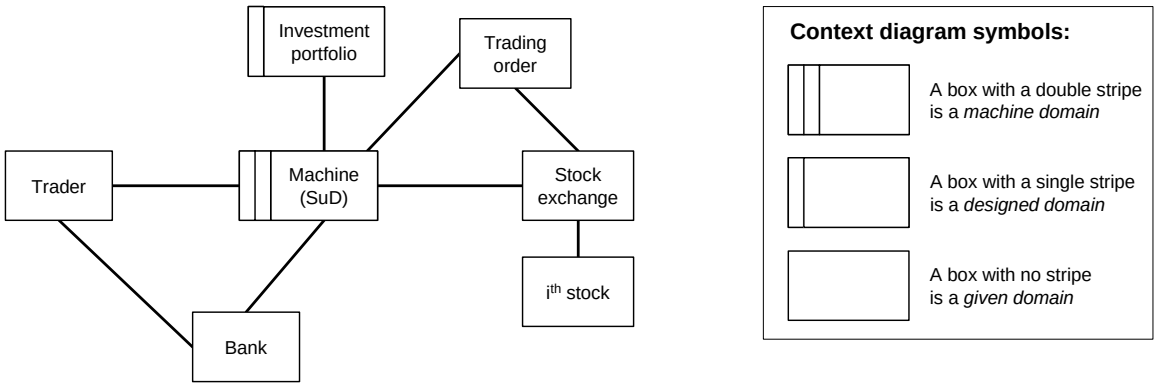


Figure 3-8: Context diagram for our case study 2: investment advisory system.

machine (or system-to-be—what is to be constructed). Some of the domains in Figure 3-8 correspond to what we called "actors" in Chapter 2. However, there are other subject matters, as well, such as "Investment portfolio."

To simplify, we decide that all the domains in the context diagram are *physical*. In Figure 3-8, while this may be clear for other domains, even "Investment portfolio" should be a physical domain. We assume that the corresponding box stands for the physical representation of the information about the stocks that the trader owns. In other words, this is the representation stored in computer memory or displayed on a screen or printed on paper. The reason for emphasizing physical domains and physical interactions is because the point of software development is to build systems that interact with the physical world and help the user solve problems.

Domain Types

Domains can be distinguished as to whether they are given or are to be designed. A *given domain* is a problem domain whose properties are given—we are not allowed to design such a domain. In some cases the machine can influence the behavior of a given domain. For example, in Figure 3-8 executing trading orders influences the behavior of the stock exchange (given domain). A *designed domain* is a problem domain for which data structures and, to some extent, its data content need to be determined and constructed. An example is the "Investment portfolio" domain in Figure 3-8.

Often, one kind of problem is distinguished from another by different domain types. To a large degree these distinctions arise naturally out of the domain phenomena. But it is also useful to make a broad classification into three main types.

♦ A **causal domain** is one whose properties include predictable causal relationships among its causal phenomena.

A causal domain may control some or all or none of the shared phenomena at an interface with another domain.

♦ A **biddable domain** usually consists of people. The most important characteristic of a biddable domain is that it lacks positive predictable internal causality. That is, in most situations it is impossible to compel a person to initiate an event: the most that can be done is to issue instructions to be followed.

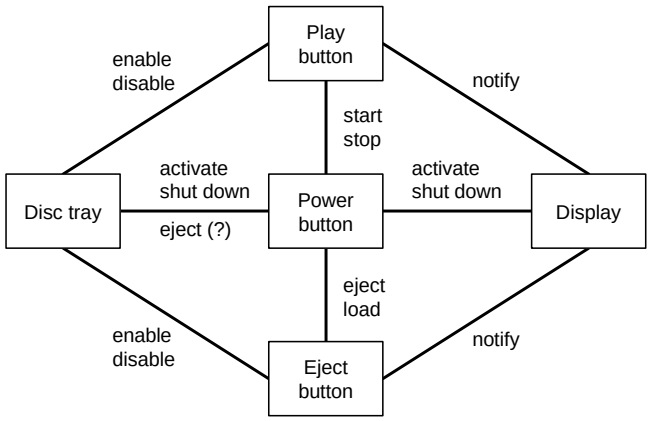


Figure 3-9: Domains and shared phenomena in the problem of controlling a DVD player.

♦ A **lexical domain** is a physical representation of data—that is, of symbolic phenomena.

Shared Phenomena

So far we considered world phenomena as belonging to particular domains. Some phenomena are shared. Shared phenomena, viewed from different domains, are the essence of domain interaction and communication. You can think of the domains as seeing the same event from different points of view.

Figure 3-9 shows

3.1.5 Systems and System Descriptions

Now that we have defined domains as distinguishable parts of the world, we can consider any domain as a system. A **system** is an organized or complex whole, an assemblage of things or parts interacting in a coordinated way. All systems are affected by events in their environment either internal and under the organization's control or external and not controllable by the organization.

Behavior under Perturbations: We need to define the initial state, other equilibrium states, and state transitions.

Most of real-world problems require a dynamical model to capture a process which changes over time. Depending on the application, the particular choice of model may be continuous or discrete (using differential or difference equations), deterministic or stochastic, or a hybrid. Dynamical systems theory describes properties of solutions to models that are prevalent across the sciences. It has been quite successful, yielding geometric descriptions of phase portraits that partition state space into region of solution trajectories with similar asymptotic behavior, characterization of the statistical properties of attractors, and classification of bifurcations marking qualitative changes of dynamical behavior in generic systems depending upon parameters. [Strogatz, 1994]

S. Strogatz, *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering*. Perseus Books Group, 1994.

Given an external perturbation or stimulus, the system responds by traversing a set of transient states until it settles at an equilibrium state. An equilibrium state may involve stable oscillations, e.g., a behavior driven by an internal clock.

In mechanics, when an external force acts on an object, we describe its behavior through a set of mathematical equations. Here we describe it as a sequence of (discrete) action-reaction or stimulus-response events, in plain English.

Figure 3-*x* shows the state transition diagram. Action "turnLightOff" is marked with question mark because we are yet to arrive at an acceptable solution for this case. The state [disarmed, unlit] is not shown because the lock is not supposed to stay for a long in a disarmed state—it will be closed shortly either by the user or automatically.

3.2 Notations for System Specification

"... psychologically we must keep all the theories in our heads, and every theoretical physicist who is any good knows six or seven different theoretical representations for exactly the same physics. He knows that they are all equivalent, and that nobody is ever going to be able to decide which one is right at that level, but he keeps them in his head, hoping that they will give him different ideas for guessing."
—Richard Feynman, *The Character of Physical Law*

3.2.1 Basic Formalisms for Specifications

"You can only find truth with logic if you have already found truth without it."
—Gilbert Keith Chesterton, *The Man who was Orthodox*

"*Logic*: The art of thinking and reasoning in strict accordance with the limitations and incapacities of the human misunderstanding." —Ambrose Bierce, *The Devil's Dictionary*

This section reviews some basic discrete mathematics that often appears in specifications. First I present a brief overview of sets notation. A *set* is a well-defined collection of objects that are called *members* or *elements*. A set is completely defined by its elements. To declare that object x is a member of a set A, write $x \in A$. Conversely, to declare that object y is not a member of a set A, write $x \notin A$. A set which has no members is the *empty set* and is denoted as { } or $\varnothing$.

Sets A and B are *equal* (denoted as $A = B$) if they have exactly the same members. If A and B are not equal, write $A \neq B$. A set B is a *subset* of a set A if all of the members of B are members of A, and this is denoted as $B \subseteq A$. The set B is a proper subset of A if B is a subset of A and $B \neq A$, which is denoted as $B \subset A$.

The *union* of two sets A and B is the set whose members belong to A, B or both, and is denoted as $A \cup B$. The *intersection* of two sets A and B is the set whose members belong to both A and B, and is denoted as $A \cap B$. Two sets A and B are *disjoint* if their intersections is the empty set: $A \cap B = \varnothing$. When $B \subseteq A$, the *set difference* $A \setminus B$ is the set of members of A which are not members of B.

The members of a set can themselves be sets. Of particular interest is the set that contains all the subsets of a given set A, including both $\varnothing$ and A itself. This set is called the *power set* of set A and is denoted $\mathbb{P}(A)$, or $\mathbb{P}A$, or 2^A.

The *ordered pair* $\langle x, y\rangle$ is a pair of objects in which x is the *first object* and y is the *second object*. Two ordered pairs $\langle x, y\rangle$ and $\langle a, b\rangle$ are *equal* if and only if $x = a$ and $y = b$. We define *Cartesian product* or *cross product* of two sets A and B (denoted as $A \times B$) as the set of all ordered pairs $\langle x, y\rangle$ where $x \in A$ and $y \in B$. We can define the n-fold Cartesian product as $A \times A \times \ldots \times A$. Recall the discussion of relations among individuals in Section 3.1.1. An n-ary *relation* R on A, for $n > 1$, is defined as a subset of the n-fold Cartesian product, $R \subseteq A \times A \times \ldots \times A$.

Boolean Logic

The rules of logic give precise meaning to statements and so they play a key role in specifications. Of course, all of this can be expressed in a natural language (such as English) or you can invent your own syntax for describing the system requirements and specification. However, if these descriptions are expressed in a standard and predictable manner, not only they can be easily understood, but also automated tools can be developed to understand such descriptions. This allows automatic checking of descriptions.

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Truth table for $p \Rightarrow q$.

Propositions are the basic building block of logic. A *proposition* is a declarative sentence (a sentence that declares a fact) that is either true or false, but not both. We already saw in Section 1.3 that a proposition is a statement of a relation among concepts, given that the truth value of the statement is known. Examples of declarative sentence are "Dogs are mammals" and "one plus one equals three." The first proposition is true and the second one is false. The sentence "Write this down" is not a proposition because it is not a declarative sentence. Also, the sentence "x is smaller than five" is not a proposition because it is neither true nor false (depends on what x is). The conventional letters used to denote propositions are p, q, r, s, … These are called *propositional variables* or *statement variables*. If a proposition is true, its truth value is denoted by T and, conversely, the truth value of a false proposition is denoted by F.

Many statements are constructed by combining one or more propositions, using logical operators, to form compound propositions. Some of the operators of propositional logic are shown on top of Table 3-1.

A *conditional statement* or, simply a *conditional*, is obtained by combining two propositions p and q to a compound proposition "if p, then q." It is also written as $p \Rightarrow q$ and can be read as "p implies q." In the conditional statement $p \Rightarrow q$, p is called the *premise* (or *antecedent* or *hypothesis*) and q is called the *conclusion* (or *consequence*). The conditional statement $p \Rightarrow q$ is false when the premise p is true and the conclusion q is false, and true otherwise. It is important to note that conditional statements should not be interpreted in terms of cause and effect. Thus, when we say "if p, then q," we do not mean that the premise p causes the conclusion q, but only that when p is true, q must be true as well[1].

[1] This is different from the if-then construction used in many programming languages. Most programming languages contain statements such as **if** p **then** S, where p is a proposition and S is a program segment of

Table 3-1: Operators of the propositional and predicate logics.

Propositional Logic			
$\wedge$	conjunction (p and q)	$\Rightarrow$	implication (if p then q)
$\vee$	disjunction (p or q)	$\Leftrightarrow$	biconditional (p if and only if q)
$\neg$	negation (not p)	$\equiv$	equivalence (p is equivalent to q)
Predicate Logic (extends propositional logic with two quantifiers)			
$\forall$	universal quantification (for all x, $P(x)$)		
$\exists$	existential quantification (there exists x, $P(x)$)		

The statement $p \Leftrightarrow q$ is a *biconditional*, or *bi-implication*, which means that $p \Rightarrow q$ and $q \Rightarrow p$. The biconditional statement $p \Leftrightarrow q$ is true when p and q have the same truth value, and is false otherwise.

So far we have considered propositional logic; now let us briefly introduce predicate logic. We saw ealier that the sentence "x is smaller than 5" is not a proposition because it is neither true nor false. This sentence has two parts: the variable x, which is the subject, and the ***predicate***, "is smaller than 5," which refers to a property that the subject of the sentence can have. We can denote this statement by $P(x)$, where P denotes the predicate "is smaller than 5" and x is the variable. The sentence $P(x)$ is also said to be the value of the *propositional function P* at x. Once a specific value has been assigned to the variable x, the statement $P(x)$ becomes a proposition and has a truth value. In our example, by setting $x = 3$, $P(x)$ is true; conversely, by setting $x = 7$, $P(x)$ is false[2].

There is another way of creating a proposition from a propositional function, called *quantification*. Quantification expresses the extent to which a predicate is true over a range of elements, using the words such as *all*, *some*, *many*, *none*, and *few*. Most common types of quantification are universal quantification and existential quantification, shown at the bottom of Table 3-1.

The *universal quantification* of $P(x)$ is the proposition "$P(x)$ is true for all values of x in the domain," denoted as $\forall x\, P(x)$. The value of x for which $P(x)$ is false is called a *counterexample* of $\forall x\, P(x)$. The *existential quantification* is the proposition "There exists a value of x in the domain such that $P(x)$ is true," denoted as $\exists x\, P(x)$.

In constructing valid arguments, a key elementary step is replacing a statement with another statement of the same truth value. We are particularly interested in compound propositions formed from propositional variables using logical operators as given in Table 3-1. Two types of compound propositions are of special interest. A compound proposition that is always true, regardless of the truth values of its constituent propositions is called a *tautology*. A simple example is $p \vee \neg p$, which is always true because either p is true or it is false. On the other hand, a compound proposition that is always false is called a *contradiction*. A simple example is $p \wedge \neg p$, because p cannot be true and false at the same time. Obviously, the negation of a tautology is a

---

one or more statements to be executed. When such an if-then statement is encountered during the execution of a program, S is executed is p is true, but S is not executed if p is false.

[2] The reader might have noticed that we already encountered predicates in Section 3.1.2 where the state relations for objects actually are predicates.

contradiction, and vice versa. Finally, compound proposition that is neither a tautology nor a contradiction is called a *contingency*.

The compound propositions p and q are said to be *logically equivalent*, denoted as $p \equiv q$, if $p \Leftrightarrow q$ is a tautology. In other words, $p \equiv q$ if p and q have the same truth values for all possible truth values of their component variables. For example, the statements $r \Rightarrow s$ and $\neg r \vee s$ are logically equivalent, which can be shown as follows. Earlier we stated that a conditional statement is false only when its premise is true and its conclusion is false, and true otherwise. We can write this as

$$\begin{aligned} r \Rightarrow s &\equiv \neg(r \wedge \neg s) \\ &\equiv \neg r \vee \neg(\neg s) \qquad \text{by the first De Morgan's law: } \neg(p \wedge q) \equiv \neg p \vee \neg q \\ &\equiv \neg r \vee s \end{aligned}$$

[For the sake of completeness, I state here, as well, the second De Morgan's law: $\neg(p \vee q) \equiv \neg p \wedge \neg q$.]

Translating sentences in natural language into logical expressions is an essential part of specifying systems. Consider, for example, the following requirements in our second case study on financial investment assistant (Section 1.3.2).

Example 3.4 Translating Requirements into Logical Expressions

Translate the following two requirements for our second case study on personal investment assistant (Table 2-2) into logical expressions:

REQ1. The system shall support registering new investors by providing a real-world email, which shall be external to our website. Required information shall include a unique login ID and a password that conforms to the guidelines, as well as investor's first and last name and other demographic information. Upon successful registration, the system shall set up an account with a zero balance for the investor.

REQ2. The system shall support placing Market Orders specified by the action (buy/sell), the stock to trade, and the number of shares. The current indicative (ask/bid) price shall be shown and updated in real time. The system shall also allow specifying the upper/lower bounds of the stock price beyond which the investor does not wish the transaction executed. If the action is to buy, the system shall check that the investor has sufficient funds in his/her account. When the market order matches the current market price, the system shall execute the transaction instantly. It shall then issue a confirmation about the outcome of the transaction (known as "order ticket"), which contains: the unique ticket number, investor's name, stock symbol, number of shares, the traded share price, the new portfolio state, and the investor's new account balance.

We start by listing all the declarative sentences that can be extracted from the requirements. REQ1 yields the following declarative sentences. Keep in mind that these are not necessarily propositions because we still do not know whether they have truth value.

Label	Declarative sentence (not necessarily a proposition!)
a	The investor can register with the system
b	The email address entered by the investor exists in real world
c	The email address entered by the investor is external to our website
d	The login ID entered by the investor is unique
e	The password entered by the investor conforms to the guidelines
f	The investor enters his/her first and last name, and other demographic info
g	Registration is successful
h	Account with zero balance is set up for the investor

Next we need to ascertain their truth value. Recall that the specifications state what is true about the system at the time it is delivered to the customer. The truth value of *a* must be established by the developer before the system is delivered. The truth values of *b*, *c*, *d*, and *e* depends on what the investor will enter. Hence, these are propositional functions at investor's input. Consider the sentence *b*. Assuming that *email* denotes the investor's input and *B* denotes the predicate in *b*, the propositional function is $B(email)$. Similarly, *c* can be written as $C(email)$, *d* as $D(id)$, and *e* as $E(pwd)$. The system can and should evaluate these functions at runtime, during the investor registration, but the specification refers to the system deployment time, not its runtime. I will assume that the truth of sentence *f* is hard to ascertain so the system will admit any input values and consider *f* true.

We have the following propositions derived from REQ1:

REQ1 represented as a set of propositions
a
$(\forall\ email)(\forall\ id)(\forall\ pwd)\ [B(email) \wedge C(email) \wedge D(id) \wedge E(pwd) \Rightarrow g]$
f
$g \Rightarrow h$

The reader should be reminded that conditional statements in logic are different from if-then constructions in programming languages. Hence, $g \Rightarrow h$ does not describe a cause-effect sequence of instructions such as: when registration is successful, do set up a zero-balance account. Rather, this simply states that when *g* is true, *h* must be true as well.

The system correctly implements REQ1 for an assignment of truth values that makes all four propositions true. Note that it would be wrong to simply write $(b \wedge c \wedge d \wedge e) \Rightarrow g$ instead of the second proposition above, for this does not correctly reflect the reality of user choice at entering the input parameters.

Extracting declarative sentences from REQ2 is a bit more involved than for REQ1. The two most complex aspects of REQ2 seem to be about ensuring the sufficiency of funds for the stock purchase and executing the order only if the current price is within the bounds (in case the trader specified the upper/lower bounds). Let us assume that the ticker symbol selected by the trader is denoted by SYM and its current ask price at the exchange is IP (for indicative price). Note that unlike the email and password in REQ1, here we can force the user to select a valid ticker symbol by displaying only acceptable options. The number of shares (volume) for the trade specified by the investor is denoted as VOL. In case the investor specifies the upper/lower bounds, let their values be denoted as UB and LB, respectively. Lastly, the investor's current account balance is denoted as BAL.

Here is a partial list of propositions needed to state these two constraints:

Label	Propositions (partial list)
m	The action specified by the investor is "buy"
n	The investor specified the upper bound of the "buy" price
o	The investor specified the lower bound of the "sell" price

The above table contains propositions because their truth value can be established independent of the user's choice. For example, the developer should allow only two choices for trading actions, "buy" or "sell," so $\neg m$ means that the investor selected "sell." In case the investor specifies the upper/lower bounds, the system will execute the transaction only if $[n \wedge m \wedge (\text{IP} \leq \text{UB})] \vee [o \wedge \neg m \wedge (\text{LB} \leq \text{IP})]$. To verify that the investor's account balance is sufficient for the current trade, the system needs to check that $[\neg n \wedge (\text{VOL} \times \text{IP} \leq \text{BAL})] \vee [n \wedge (\text{VOL} \times \text{UB} \leq \text{BAL})]$.

The additional declarative sentences extracted from REQ2 are:

Label	Propositions (they complete the above list)
p	The investor requests to place a market order
q	The investor is shown a blank ticket where the trade can be specified (action, symbol, etc.)
r	The most recently retrieved indicative price is shown in the currently open order ticket
s	The symbol SYM specified by the investor is a valid ticker symbol
t	The current indicative price that is obtained from the exchange

u The system executes the trade
v The system calculates the player's account new balance
w The system issues a confirmation about the outcome of the transaction
x The system archives the transaction

We have the following propositions derived from REQ2:

REQ2 represented as a set of propositions
$p \Rightarrow q \wedge r$
s
$\boldsymbol{y} = v \wedge \{\neg(n \vee o) \vee [(o \wedge p \vee \neg o \wedge q) \wedge (\exists \text{ IP})(\text{LB} \leq \text{IP} \leq \text{UB})]\}$
$\boldsymbol{z} = \neg m \vee \{[\neg n \wedge (\text{VOL} \times \text{IP} \leq \text{BAL})] \vee [n \wedge (\text{VOL} \times \text{UB} \leq \text{BAL})]\}$
$y \wedge z \Rightarrow u$
$u \Rightarrow v \wedge w \wedge x$

Again, all of the above propositions must evaluate to true for the system to correctly implement REQ2. Unlike REQ1, we have managed to restrict the user choice and simplify the representation of REQ2. It is true that by doing this we went beyond mere problem statement and imposed some choices on the problem solution, which is generally not a good idea. But in this case I believe these are very simple and straightforward choices. It requires the developer's judgment and experience to decide when simplification goes too far into restricting the solution options, but sometimes the pursuit of purity only brings needless extra work.

System specifications should be **consistent**, which means that they should not contain conflicting requirements. In other words, if the requirements are represented as a set of propositions, there should be an assignment of truth values to the propositional variables that makes all requirements propositions true.

Example…

In Section 3.2.3 we will see how logic plays role in the part of the UML standard called Object Constraint Language (OCL). Another notation based on Boolean logic is TLA+, described in Section 3.2.4.

Finite State Machines

The behavior of complex objects and systems depends not only on their immediate input, but also on the past history of inputs. This memory property, represented as a *state*, allows such systems to change their actions with time. A simple but important formal notation for describing such systems is called *finite state machines* (FSMs). FSMs are used extensively in computer science and data networking, and the UML standard extends the FSMs into UML state machine diagrams (Section 3.2.2).

There are various ways to represent a finite state machine. One way is to make a table showing how each input affects the state the machine is in. Here is the *state table* for the door lock used in our case-study example

	Present state

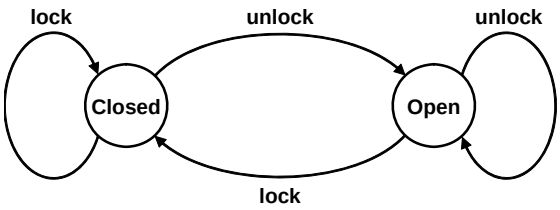
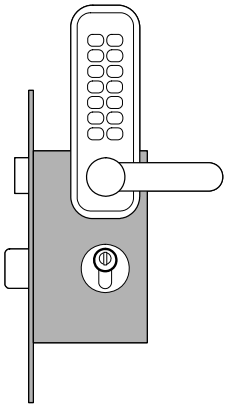


Figure 3-10: State transition diagram for a door lock.

		Armed	Disarmed
Input	lock	Armed	Armed
	unlock	Disarmed	Disarmed

Here, the entries in the body of the table show the next state the machine enters, depending on the present state (column) and input (row).

We can also represent our machine graphically, using a *transition diagram*, which is a directed graph with labeled edges. In this diagram, each state is represented by a circle. Arrows are labeled with the input for each transition. An example is shown in Figure 3-10. Here the states "Disarmed" and "Armed" are shown as circles, and labeled arrows indicate the effect of each input when the machine is in each state.



A *finite state machine* is formally defined to consist of a finite set of states S, a finite set of inputs I, and a transition function with $S \times I$ as its domain and S as its codomain (or range) such that if $s \in S$ and $i \in I$, the $f(s, i)$ is the state the machine moves to when it is in state s and is given input i. Function f can be a partial function, meaning that it can be undefined for some values of its domain. In certain applications, we may also specify an *initial state* s_0 and a set of *final* (or *accepting*) *states* $S' \subset S$, which are the states we would like the machine to end in. Final states are depicted in state diagrams by using double concentric circles. An example is shown in Figure 3-11, where M = maxNumOfAttempts is the final state: the machine will halt in this state and needs to be restarted externally.

A *string* is a finite sequence of inputs. Given a string $i_1 i_2 \ldots i_n$ and the initial state s_0, the machine successively computes $s_1 = f(s_0, i_1)$, then $s_2 = f(s_1, i_2)$, and so on, finally ending up with state s_n. For the example in Figure 3-11, the input string *iiv* transitions the FSM through the states $s_0 s_1 s_2 s_0$. If $s_n \in S'$, i.e., it is an accepting state, then we say that the string is *accepted*; otherwise it is *rejected*. It is easy to see that in Figure 3-11, the input string of M i's (denoted as i^M) will be accepted. We say that this machine *recognizes* this string and, in this sense, it recognizes the

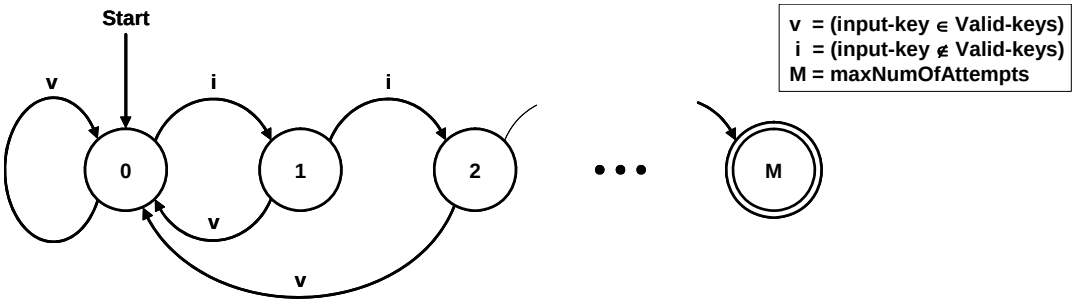


Figure 3-11: State transition diagram for the counter of unsuccessful lock-opening attempts.

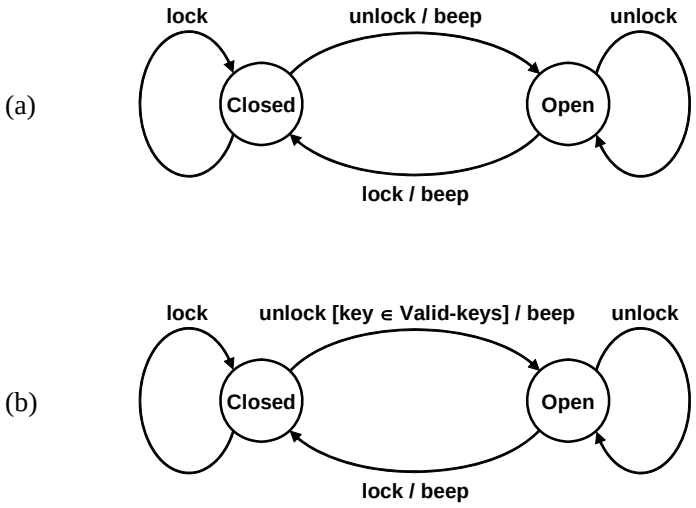


Figure 3-12: State transition diagram from Figure 3-10, modified to include output labels (a) and guard labels (b).

attempted intrusion.

A slightly more complex machine is an FSM that yields output when it transitions to the next state. Suppose that, for example, the door lock in Figure 3-10 also produces an audible signal to let the user know that it is armed or disarmed. The modified diagram is shown in Figure 3-12(a). We use a slash symbol to separate the input and the output labels on each transition arrow. (Note that here we choose to produce no outputs when the machine receives duplicate inputs.)

We define a *finite state machine with output* to consist of a finite set of states *S*, a finite set of inputs *I*, a finite set of outputs *O*, along with a function $f : S \times I \rightarrow S$ that assigns to each (state, input) pair a new state and another function $g : S \times I \rightarrow O$ that assigns to each (state, input) pair an output.

We can enrich the original FSM model by adding new features. Figure 3-12(b) shows how we can add *guards* to transitions. The full notation for transition descriptions is then ⟨*input*[*guard*]/*output*⟩, where each element is optional. A guard is a Boolean proposition that permits or blocks the transition. When a guard is present, the transition takes place if the guard evaluates to true, but the transition is blocked if the guard is false. Section 3.2.2 describes how UML adds other features to extend the FSM model into UML state machine diagrams.

3.2.2 UML State Machine Diagrams

One of the key weaknesses of the original finite-state-machines model (described in the preceding section) in the context of system and software specification is the lack of *modularization* mechanisms. When considering the definitions of states and state variables in Section 3.1.2, FSMs are suitable for representing individual simple states (or microstates). UML state machine diagrams provide a standardized diagrammatic notation for state machines and also incorporate extensions, such as macrostates and concurrent behaviors.

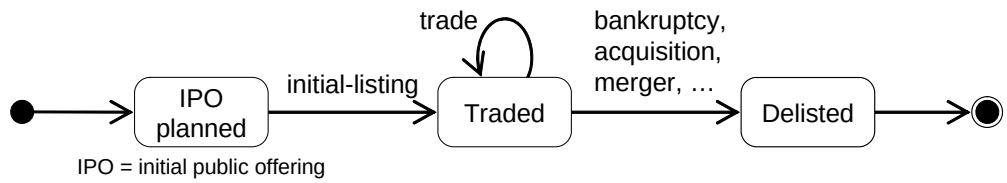


Figure 3-13: UML state machine diagram showing the states and transitions of a stock.

Basic Notation

In every state machine diagram, there must be exactly one default *initial state*, which we designate by writing an unlabeled transition to this state from a special icon, shown as a filled circle. An example is shown in Figure 3-13. Sometimes we also need to designate a stop state. In most cases, a state machine associated with an object or the system as a whole never reaches a stop state—the state machine just vanishes when the object it represents is destroyed. We designate a *stop state* by drawing an unlabeled state transition from this state to a special icon, shown as a filled circle inside a slightly larger hollow circle.[3] Initial and final states are called *pseudostates*.

Transitions between pairs of states are shown by directed arrows. Moving between states is referred to as *firing the transition*. A state may transition to itself, and it is common to have many different state transitions from the same state. All transitions must be unique, meaning that there will never be any circumstances that would trigger more than one transition from the same state.

There are various ways to control the firing of a transition. A transition with no annotation is referred to as a *completion transition*. This simply means that when the object completes the execution of an activity in the source state, the transition automatically fires, and the target state is entered.

In other cases, certain events have to occur for the transition to fire. Such events are annotated on the transition. (Events were discussed is Section 3.1.3.) In Figure 3-13, one may argue that *bankruptcy* or *acquisition* phenomena should be considered states rather than events, because company stays in bankruptcy for much longer than an instant of time. The correct answer is relative to the observer. Our trader would not care how long the company will be in bankruptcy—the only thing that matters is that its stock is not tradable anymore starting with the moment the bankruptcy becomes effective.

We have already seen for FSMs that a *guard condition* may be specified to control the transition. These conditions act as guards so that when an event occurs, the condition will either allow the transition (if the condition is true) or disallow the transition (if the condition is false).

State Activities: Entry, Do, and Exit Activities

I already mentioned that states can be passive or active. In particular, an activity may be specified to be carried out at certain points in time with respect to a state:

---

[3] The *Delisted* state in Figure 3-13 is the stop state with respect to the given exchange. Although investors can no longer trade shares of the stock on that exchange, it may be traded on some other markets

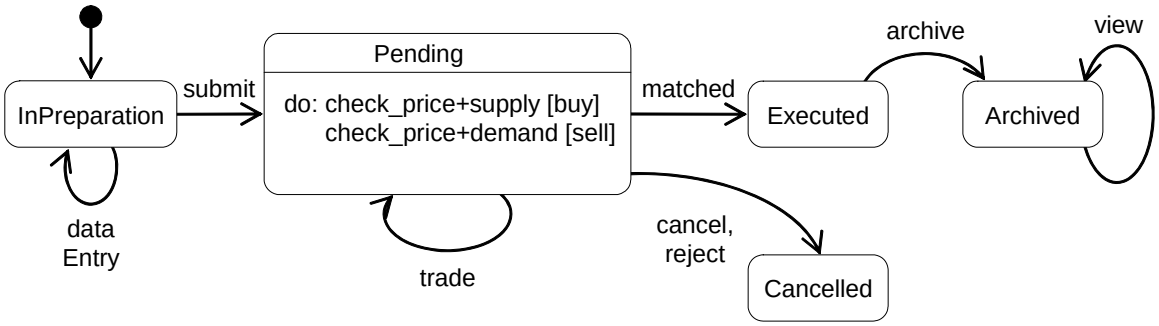


Figure 3-14: Example of state activities for a trading order. Compare with Figure 3-7.

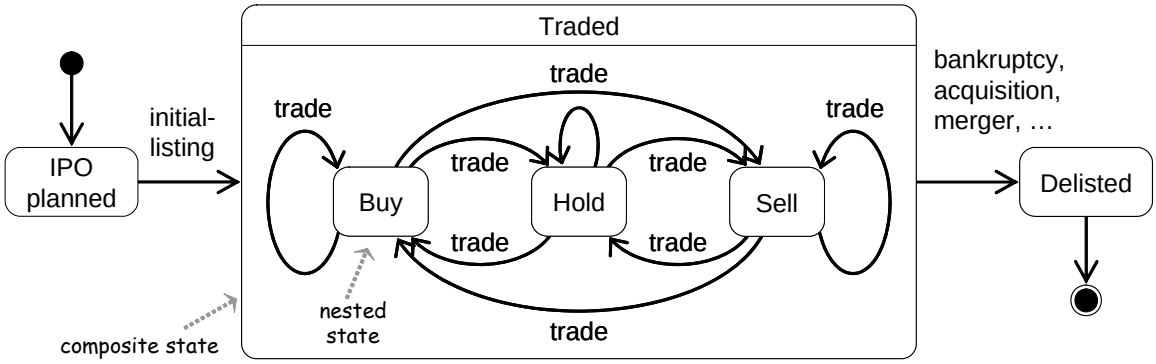


Figure 3-15: Example of composite and nested states for a stock. Compare with Figure 3-13.

- Perform an activity upon entry of the state
- Do an activity while in the state
- Perform an activity upon exit of the state

An example is shown in Figure 3-14.

Composite States and Nested States

UML state diagrams define *superstates* (or macrostates). A superstate is a complex state that is further refined by decomposition into a finite state machine. A superstate can also be obtained by aggregation of elementary states, as already seen in Section 3.1.2.

Suppose now that we wish to extend the diagram in Figure 3-13 to show the states *Buy*, *Sell*, and *Hold*, which we defined in Example 3.2. These states are a refinement of the *Traded* state within which they are nested, as shown in Figure 3-15. This nesting is depicted with a surrounding boundary known as a *region* and the enclosing boundary is called a *composite state*. Given the composite state *Traded* with its three substates, the semantics of nesting implies an exclusive OR (XOR) relationship. If the stock is in the *Traded* state (the composite state), it must also be in exactly one of the three substates: *Buy*, *Hold*, or *Sell*.

Nesting may be to any depth, and thus substates may be composite states to other lower-level substates. For simplicity in drawing state transition diagrams with depth, we may zoom in or zoom out relative to a particular state. Zooming out conceals substates, as in Figure 3-13, and

Figure 3-16: Example of concurrency in states.

zooming in reveals substates, as in Figure 3-15. Zoomed out representation may improve comprehensibility of a complex state machine diagram.

Concurrency

Figure 3-16

Applications

State machine diagrams are typically used to describe the behavior of individual objects. However, they can also be used to describe the behavior of any abstractions that the developer is currently considering. We may also provide state machine diagrams for the entire system under consideration. During the analysis phase of the development lifecycle (described in Section 2.5), we are considering the event-ordered behavior of the system as a whole; hence, we may use state machine diagrams to represent the behavior of the system. During the design phase (described in Section 2.6), we may use state machine diagrams to capture dynamic behavior of individual classes or of collaborations of classes.

In Section 3.3 we will use state machine diagrams to describe problem domains when trying to understand and decompose complex problems into basic problems.

3.2.3 UML Object Constraint Language (OCL)

"I can speak French but I cannot understand it." —Mark Twain

The UML standard defines Object Constraint Language (OCL) based on Boolean logic. Instead of using mathematical symbols for operators (Table 3-1), OCL uses only ASCII characters which makes it easier for typing and computer processing. It also makes OCL a bit wordy in places.

OCL is not a standalone language, but an integral part of the UML. An OCL expression needs to be placed within the context of a UML model. In UML diagrams, OCL is primarily used to write constraints in class diagrams and guard conditions in state and activity diagrams. OCL expressions, known as *constraints*, are added to express facts about elements of UML diagrams.

Table 3-2: Basic predefined OCL types and operations on them.

Type	Values	Operations
Boolean	true, false	and, or, xor, not, implies, if-then-else
Integer	1, 48, –3, 84967, …	*, +, –, /, abs()
Real	0.5, 3.14159265, 1.e+5	*, +, –, /, floor()
String	'With more exploration comes more text.'	concat(), size(), substring()

Any implementation derived from such a design model must ensure that each of the constraints always remains true.

We should keep in mind that for software classes there is no notion of a computation to specify in the sense of having well-defined start and end points. A class is not a program or subroutine. Rather, any of object's operations can be invoked at arbitrary times with no specific order. And the *state* of the object can be an important factor in its behavior, rather than just input-output relations for the operation. Depending on its state, the object may act differently for the same operation. To specify the effect of an operation on object's state, we need to be able to describe the present state of the object which resulted from any previous sequence of operations invoked on it. Because object's state is captured in its attributes and associations to other objects, OCL constraints usually relate to these properties of objects.

OCL Syntax

OCL's syntax is similar to object-oriented languages such as C++ or Java. OCL expressions consist of model elements, constraints, and operators. Model elements include class attributes, operations, and associations. However, unlike programming languages OCL is a pure specification language, meaning that an OCL expression is guaranteed to be without side effects. When an OCL expression is evaluated, it simply returns a value. The state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to *specify* a state change, such as in a post-condition specification.

OCL has four built-in types: Boolean, Integer, Real, and String. Table 3-2 shows example values and some examples of the operations on the predefined types. These predefined value types are independent of any object model and are part of the definition of OCL.

When writing an OCL contract, the first step is to decide the *context*, which is the software class for which the OCL expression is applicable. Within the given class context, the keyword `self` refers to all instances of the class. Other model elements can be obtained by navigating using the dot notation from the `self` object. Consider the example of the class diagram in Figure 2-35 (Section 2.6). To access the attribute `numOfAttempts_` of the class `Controller`, we write

```
self.numOfAttempts_
```

Due to encapsulation, object attributes frequently must be accessed via accessor methods. Hence, we may need to write `self.getNumOfAttempts()`.

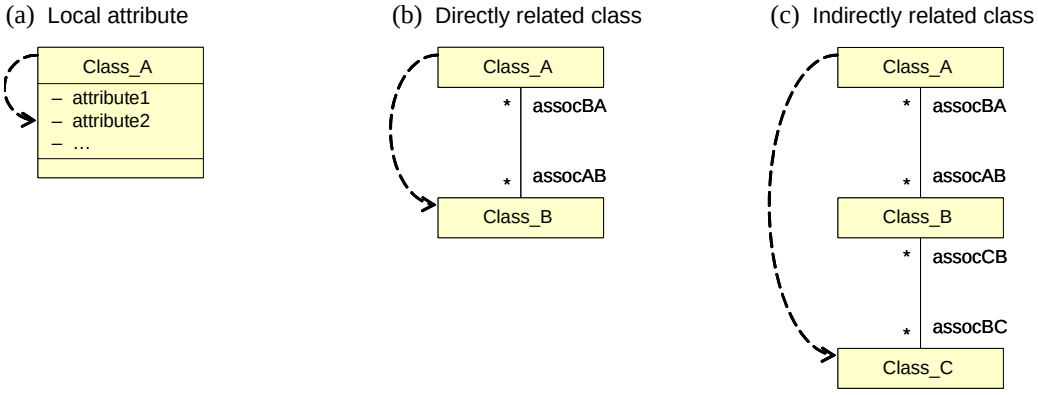


Figure 3-17: Three basic types of navigation in a UML class diagram. (a) Attributes of class A accessed from an instance of class A. (b) Accessing a set of instances of class B from an instance of class A. (c) Accessing a set of instances of class C from an instance of class A.

Starting from a given context, we can *navigate associations* on the class diagram to refer to other model elements and their properties. The three basic types of navigation are illustrated in Figure 3-17. In the context of `Class_A`, to access its local attribute, we write `self.attribute2`. Similarly, to access instances of a directly associated class we use the name of the opposite association-end in the class diagram. So in Figure 3-17(b), in the context of `Class_A`, to access the set of instances of `Class_B`, we write `self.assocAB`. Lastly in Figure 3-17(c), in the context of `Class_A`, to access instances of an indirectly associated class `Class_C`, we write `self.assocAB.assocBC`. (This approach should not come as a surprise to the reader familiar with an object programming language, such as Java or C#.)

We already know from UML class diagrams that object associations may be individual objects (association multiplicity equals 1) or collections (association multiplicity > 1). Navigating a one-to-one association yields directly an object. Figure 2-35 shows a single `LockCtrl` (assuming that a single lock device is controlled by the system). Assuming that this association is named `lockCtrl_` as in Listing 2.2, the navigation `self.lockCtrl_` yields the single object `lockCtrl_ : LockCtrl`. However, if the `Controller` were associated with multiple locks, e.g., on front and backyard doors, then this navigation would yield a collection of two `LockCtrl` objects.

OCL specifies three types of *collections*:

- *OCL sets* are used to collect the results of navigating immediate associations with one-to-many multiplicity.
- *OCL sequences* are used when navigating immediate ordered associations.
- *OCL bags* are used to accumulate the objects when navigating indirectly related objects. In this case, the same object can show up multiple times in the collection because it was accessed via different navigation paths.

Note that in the example in Figure 3-17(c), the expression `self.assocAB.assocBC` evaluates to the set of all instances of class `Class_C` objects associated with all instances of class `Class_B` objects that, in turn, are associated with class `Class_A` objects.

Table 3-3: Summary of OCL operations for accessing collections.

OCL Notation	Meaning
EXAMPLE OPERATIONS ON ALL OCL COLLECTIONS	
`c->size()`	Returns the number of elements in the collection `c`.
`c->isEmpty()`	Returns true if `c` has no elements, false otherwise.
`c1->includesAll(c2)`	Returns true if every element of `c2` is found in `c1`.
`c1->excludesAll(c2)`	Returns true if no element of `c2` is found in `c1`.
`c->forAll(var | expr)`	Returns true if the Boolean expression `expr` true for all elements in `c`. As an element is being evaluated, it is bound to the variable `var`, which can be used in `expr`. This implements universal quantification ∀.
`c->forAll(var1, var2 | expr)`	Same as above, except that `expr` is evaluated for every possible *pair* of elements from `c`, including the cases where the pair consists of the same element.
`c->exists(var | expr)`	Returns true if there exists at least one element in `c` for which `expr` is true. This implements existential quantification ∃.
`c->isUnique(var | expr)`	Returns true if `expr` evaluates to a different value when applied to every element of `c`.
`c->select(expr)`	Returns a collection that contains only the elements of `c` for which `expr` is true.
EXAMPLE OPERATIONS SPECIFIC TO OCL SETS	
`s1->intersection(s2)`	Returns the set of the elements found in `s1` and also in `s2`.
`s1->union(s2)`	Returns the set of the elements found either `s1` or `s2`.
`s->excluding(x)`	Returns the set `s` without object `x`.
EXAMPLE OPERATION SPECIFIC TO OCL SEQUENCES	
`seq->first()`	Returns the object that is the first element in the sequence `seq`.

To distinguish between attributes in classes from collections, OCL uses the dot notation for accessing attributes and the arrow operator **`->`** for accessing collections. To access a property of a collection, we write the collection's name, followed by an arrow `->`, and followed by the name of the property. OCL provides many predefined operations for accessing collections, some of which are shown in Table 3-3.

Constants are unchanging (non-mutable) values of one of the predefined OCL types (Table 3-2). *Operators* combine model elements and constants to form an *expression*.

OCL Constraints and Contracts

Contracts are constraints on a class that enable the users of the class, implementers, and extenders to share the same assumptions about the class. A contract specifies constraints on the class state that must be valid always or at certain times, such as before or after an operation is invoked. The contract is between the class implementer about the promises of what can be

expected and the class user about the obligations that must be met before the class is used. There are three types of constraints in OCL: invariants, preconditions, and postconditions.

One important characterization of object states is describing what remains invariant throughout the object's lifetime. This can be described using an invariant predicate. An **invariant** must always evaluate to true for all instance objects of a class, regardless of what operation is invoked and in what order. An invariant applies to a class attribute.

In addition, each operation can be specified by stating a precondition and a postcondition. A **precondition** is a predicate that is checked before an operation is executed. A precondition applies to a specific operation. Preconditions are frequently used to validate input parameters to an operation.

A **postcondition** is a predicate that must be true after an operation is executed. A postcondition also applies to a specific operation. Postconditions are frequently used to describe how the object's state was changed by an operation.

We already encountered some preconditions and postconditions in the context of domain models (Section 2.5.4). Subsequently, in Figure 2-35 we assigned the domain attributes to specific classes. Therein, we used an informal, ad-hoc notation. OCL provides a formal notation for expressing constraints. For example, one of the constraints for our case study system is that the maximum allowed number of failed attempts at disarming the lock is a positive integer. This constraint must be always true, so we state it as an invariant:

```
context Controller inv:
      self.getMaxNumOfAttempts() > 0
```

Here, the first line specifies the context, i.e., the model element to which the constraint applies, as well as the type of the constraint. In this case the `inv` keyword indicates the *invariant* constraint type. In most cases, the keyword *self* can be omitted because the context is clear.

Other possible types of constraint are *precondition* (indicated by the `pre` keyword) and *postcondition* (indicated by the `post` keyword). A precondition for executing the operation `enterKey()` is that the number of failed attempts is less than the maximum allowed number:

```
context Controller::enterKey(k : Key) : boolean pre:
      self.getNumOfAttempts() < self.getMaxNumOfAttempts()
```

The postconditions for `enterKey()`are that (Poc1) a failed attempt is recorded, and (Poc2) if the number of failed attempts reached the maximum allowed number, the system becomes blocked and the alarm bell is sounded. The first postcondition (Poc1) can be restated as:

(Poc1′) If the provided key is not element of the set of valid keys, then the counter of failed attempts after exiting from `enterKey()` must be by one greater than its value before entering `enterKey()`.

The above two postconditions (Poc1′) and (Poc2) can be expressed in OCL as:

```
context Controller::enterKey(k : Key) : Boolean

   -- postcondition (Poc1′):
   post: let allValidKeys : Set = self.checker.validKeys()
         if allValidKeys.exists(vk | k = vk) then
```

```
            getNumOfAttempts() = getNumOfAttempts()@pre
        else
            getNumOfAttempts() = getNumOfAttempts()@pre + 1
    -- postcondition (Poc2):
    post: getNumOfAttempts() >= getMaxNumOfAttempts() implies
            self.isBlocked() and self.alarmCtrl.isOn()
```

There are three features of OCL used in stating the first postcondition above that the reader should note. First, the `let` expression allows one to define a variable (in this case `allValidKeys` of the OCL collection type `Set`) which can be used in the constraint.

Second, the `@pre` directive indicates the value of an object as it existed *prior* to the operation. Hence, `getNumOfAttempts()@pre` denotes the value returned by `getNumOfAttempts()`before invoking `enterKey()`, and `getNumOfAttempts()`denotes the value returned by the same operation after invoking `enterKey()`.

Third, the expressions about `getNumOfAttempts()` in the `if-then-else` operation are *not assignments*. Recall that OCL is not a programming language and evaluation of an OCL expression will never change the state of the system. Rather, this just evaluates the equality of the two sides of the expression. The result is a Boolean value true or false.

SIDEBAR 3.1: The Dependent Delegate Dilemma

♦ The class invariant is a key concept of object-oriented programming, essential for reasoning about classes and their instances. Unfortunately, the class invariant is, for all but non-trivial examples, not always satisfied. During the execution of the method that client object called on the server object ("dependent delegate"), the invariant may be temporarily violated. This is considered acceptable because in such an intermediate state the server object is not directly usable by the rest of the world—it is busy executing the method that client called—so it does not matter that its state might be inconsistent. What counts is that the invariant will hold before and after the execution of method calls.

However, if during the executing of the server's method the server calls back a method on the client, then the server may catch the client object in an inconsistent state. This is known as the *dependent delegate dilemma* and is difficult to handle. The interested reader should check [Meyer, 2005] for more details.

The OCL standard specifies only contracts. Although not part of the OCL standard, nothing prevents us from specifying program behavior using Boolean logic. [give example]

3.2.4 TLA+ Notation

This section presents TLA+ system specification language, defined by Leslie Lamport. The book describing TLA+ can be downloaded from http://lamport.org/. There are many other specification languages, and TLA+ reminds in many ways of Z (pronounced Zed, not Zee) specification

```
1  ──────────────────── MODULE AccessController ────────────────────
2  CONSTANTS validKeys,                    The set of valid keys.
3                ValidateKey( _ )          A ValidateKey(k) step checks if k is a valid key.
4  ASSUME validKeys ⊂ STRING
5  ASSUME ∀ key ∈ STRING : ValidateKey(key) ∈ BOOLEAN
6  VARIABLE status
7  TypeInvariant ≜ status ∈ [lock : {"disarmed", "armed"}, bulb : {"lit", "unlit"}]
8  ──────────────────────────────────────────────────────────────────
9  Init ≜ ∧ TypeInvariant                  The initial predicate.
10        ∧ status.lock = "armed"
11        ∧ status.bulb = "unlit"

12 Unlock(key) ≜ ∧ ValidateKey(key)        Only if the user enters a valid key, then
13        ∧ status'.lock = "disarmed"      unlock the lock and
14        ∧ status'.bulb = "lit"           turn on the light (if not already lit).

15 Lock ≜ ∧ status'.lock = "armed"         Anybody can lock the doors
16        ∧ UNCHANGED status.bulb          but not to play with the lights.

17 Next ≜ Unlock(key) ∨ Lock               The next-state action.
18
19 Spec ≜ Init ∧ □[Next]_status            The specification.
20 ──────────────────────────────────────────────────────────────────
21 THEOREM Spec ⇒ □ TypeInvariant          Type correctness of the specification.
22 ──────────────────────────────────────────────────────────────────
```

Figure 3-18: TLA+ specification of the cases study system.

language. My reason for choosing TLA+ is that it uses the language of mathematics, specifically the language of Boolean algebra, rather than inventing another formalism.

A TLA+ specification is organized in a module, as in the following example, Figure 3-18, which specifies our home access case study system (Section 1.3.1). Observe that TLA+ language reserved words are shown in SMALL CAPS and comments are shown in a highlighted text. A module comprises several sections

- Declaration of *variables*, which are primarily the manifestations of the system visible to an outside observer
- Definition of the *behavior*: the *initial state* and all the subsequent (*next*) *states*, which combined make the specification
- The *theorems* about the specification

The variables could include internal, invisible aspects of the system, but they primarily address the external system's manifestations. In our case-study of the home access controller, the variables of interest describe the state of the lock and the bulb. They are aggregated in a single *status* record, lines 6 and 7.

The separator lines 8 and 20 are a pure decoration and can be omitted. Unlike these, the module start and termination lines, lines 1 and 22, respectively, have semantic meaning and must appear.

Lines 2 and 3 declare the constants of the module and lines 4 and 5 list our assumptions about these constants. For example, we assume that the set of valid passwords is a subset of all character strings, symbolized with STRING. Line 5 essentially says that we expect that for any key *k*, *ValidateKey*(*k*) yields a BOOLEAN value.

TypeInvariant in line 7 specifies all the possible values that the system variable(s) can assume in a behavior that satisfies the specification. This is a property of a specification, not an assumption. That is why it is stated as a theorem at the end of the specification, line 21.

The definition of the initial system state appears in lines 9 and 10.

Before defining the next state in line 17, we need to define the functions that could be requested of the system. In this case we focus only on the key functions of disarming and arming the lock, *Disarm* and *Arm*, respectively, and ignore the rest (see all the use cases in Section 2.2). Defining these functions is probably the most important part of a specification.

The variable *status′* with an apostrophe symbol represents the state variable in the next step, after an operation takes place.

3.3 Problem Frames

"Computers are useless. They can only give you answers." —Pablo Picasso

"Solving a problem simply means representing it so as to make the solution transparent."
—Herbert Simon, *The Sciences of the Artificial*

Problem frames were proposed by Michael Jackson [1995; 2001] as a way for understanding and systematic describing the problem as a first step towards the solution. Problem frames decompose the original complex problem into simple, known subproblems. Each frame captures a problem class stylized enough to be solved by a standard method and simple enough to present clearly separated concerns.

We have an intuitive feeling that a problem of data acquisition and display is different from a problem of text editing, which in turn is different from writing a compiler that translates source code to machine code. Some problems combine many of these simpler problems. The key idea of problem frames is to identify the categories of simple problems, and to devise a methodology for representing complex problems in terms of simple problems.

There are several issues to be solved for successful formulation of a problem frame methodology. First we need to identify the frame categories. One example is the *information frame*, which represents the class of problems that are primarily about data acquisition and display. We need to define the *notation* to be used in describing/representing the frames. Then, given a complex problem, we need to determine how to *decompose* it into a set of problem frames. Each individual frame can then be considered and solved independently of other frames. A key step in solving a frame is to address the *frame concerns*, which are generic aspects of each problem type that need to be addressed for solving a problem of a particular type.

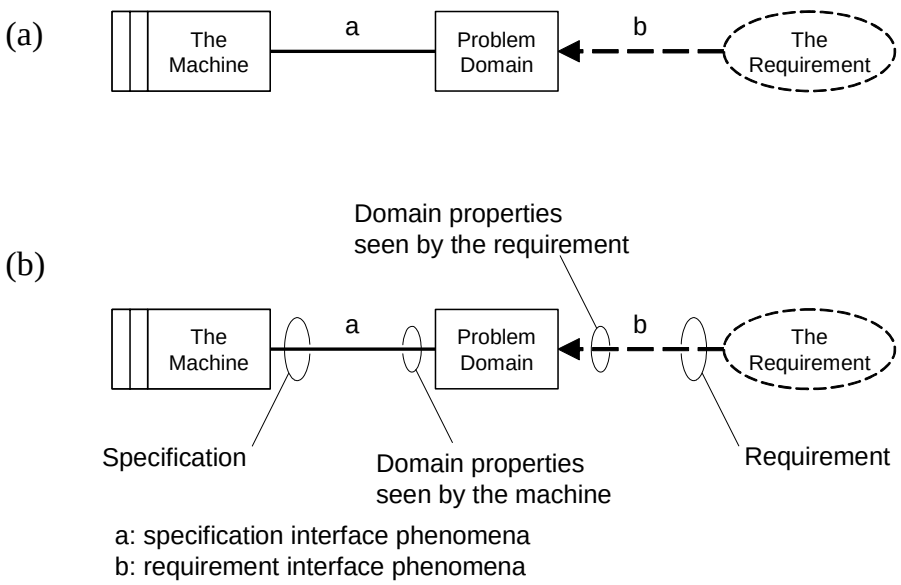


Figure 3-19: (a) The Machine and the Problem Domain. (b) Interfaces between the problem domain, the requirements and the machine.

Finally, we need to determine how to *compose* the individual solutions into the overall solution for the original problem. We need to determine how individual frames interact with each other and we may need to resolve potential conflicts of their interaction.

3.3.1 Problem Frame Notation

We can picture the relationship between the computer system to be developed and the real world where the problem resides as in Figure 3-19. The task of software development is to construct the Machine by programming a general-purpose computer. The machine has an interface *a* consisting of a set of phenomena—typically events and states—shared with the Problem Domain. Example phenomena are keystrokes on a computer keyboard, characters and symbols shown on a computer screen, signals on the lines connecting the computer to an instrument, etc.

The purpose of the machine is described by the Requirement, which specifies that the machine must produce and maintain some relationship among the phenomena of the problem domain. For example, to disarm the lock device when a correct code is presented, or to ensure that the figures printed on a restaurant check correctly reflect the patron's consumption.

Phenomena *a* shared by a problem domain and the machine are called *specification phenomena*. Conversely, phenomena *b* articulate the requirements and are called the *requirement phenomena*. Although *a* and *b* may be overlapping, they are generally distinct. The requirement phenomena are the subject matter of the customer's requirement, while the specification phenomena describe the interface at which the machine can monitor and control the problem domain.

A problem diagram as in Figure 3-19 provides a basis for problem analysis because it shows you what you are concerned with, and what you must describe and reason about in order to analyze the problem completely. The key topics of your descriptions will be:

- The *requirement* that states what the machine must do. The requirement is what your customer *would like* to achieve in the problem domain. Its description is *optative* (it describes the *option* that the customer has chosen). Sometimes you already have an exact description of the requirement, sometimes not. For example, requirement REQ1 given in Table 2-2 states precisely how users are to register with our system.
- The *domain properties* that describe the relevant characteristics of each problem domain. These descriptions are *indicative* because they describe what is true regardless of the machine's behavior. For example, Section 1.3.2 describes the functioning of financial markets, which we must understand to implement a useful system that will provide investment advice.
- The machine *specification*. Like the requirement, this is an *optative* description: it describes the machine's desired behavior at its interfaces with the problem domain.

Obviously, the indicative domain properties play a key role: without a clear understanding of how financial markets work we would never be able to develop a useful investment assistant system.

3.3.2 Problem Decomposition into Frames

Problem analysis relies on a strategy of *problem decomposition* based on the type of problem domain and the domain properties. The resulting *subproblems* are treated independently of other subproblems, which is the basis of effective separation of concerns. Each subproblem has its own machine (specification), problem domain(s), and requirement. Each subproblem is a *projection* of the full problem, like color separation in printing, where colors are separated independently and then overlaid (superimposed) to form the full picture.

Jackson [2001] identifies five primitive *problem frames*, which serve as the basic units of problem decomposition. These are (*i*) required behavior, (*ii*) commanded behavior, (*iii*) information display, (*iv*) simple workpieces, and (*v*) transformation. They differ in their requirements, domain characteristics, domain involvement (whether the domain is controlled, active, inert, etc.), and the frame concern. These problem frames correspond to the problem types identified earlier in Section 2.3.1 (see Figure 2-11).

Each frame has a particular concern, which is a set of generic issues that need to be solved when solving the frame:

(a) *Required behavior frame concern*: To describe precisely (1) how the controlled domain currently behaves; (2) the desired behavior for the domain, as stated by the requirement; and, (3) what the machine (software-to-be) will be able to observe about the domain state, by way of the sensors that will be used in the system-to-be.

(b) *Commanded behavior frame concern*: To identify (1) all the commands that will be possible in the envisioned system-to-be; (2) the commands that will be supported or

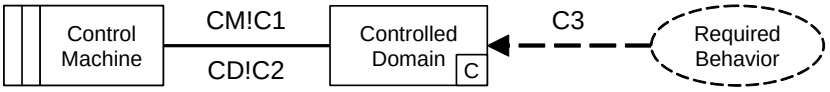


Figure 3-20: Problem frame diagram for the *required behavior frame*.

allowed under different scenarios; and, (3) what should happen if the user tries to execute a command that is not supported/allowed under the current scenario.

(c) *Information display frame concern*: To identify (1) the information that the machine will be able to observe from the problem domain, by way of the sensors that will be used in the system-to-be; (2) the information that needs to be displayed, as stated by the requirement; and, (3) the transformations needed to process the raw observed information to obtain displayable information.

(d) *Simple workpieces frame concern*: To describe precisely (1) the data structures of the workpieces; (2) all the commands that will be possible in the envisioned system-to-be; (3) the commands that will be supported or allowed under different scenarios; and, (4) what should happen if the user tries to execute a command that is not supported/allowed under the current scenario.

(e) *Transformation frame concern*: To describe precisely (1) the data structures of the input data and output data; (2) how each data structure will be traversed (travelled over); and, (3) how each element of the input data structure will be transformed to obtain the corresponding element in the output data structure.

Identification and analysis of *frame flavors*, reflecting a finer classification of domain properties

The *frame concern* is to make the requirement, specification, and domain descriptions and to fit them into a correctness argument for the machine to be built. Frame concerns include: initialization, overrun, reliability, identities, and completeness. The *initialization concern* is to ensure that a machine is properly synchronized with the real world when it starts.

… *frame variants*, in which a domain is usually added to a problem frame

Basic Frame Type 1: Required Behavior

In this scenario, we need to build a machine which controls the behavior of a part of the physical world according to given conditions.

Figure 3-20 shows the frame diagram for the required behavior frame. The control machine is the machine (system) to be built. The controlled domain is the part of the world to be controlled. The requirement, giving the condition to be satisfied by the behavior of the controlled domain, is called the required behavior.

The controlled domain is a causal domain, as indicated by the C in the bottom right corner of its box. Its interface with the machine consists of two sets of causal phenomena: C1, controlled by the machine, and C2, controlled by the controlled domain. The machine imposes the behavior on the controlled domain by the phenomena C1; the phenomena C2 provide feedback.

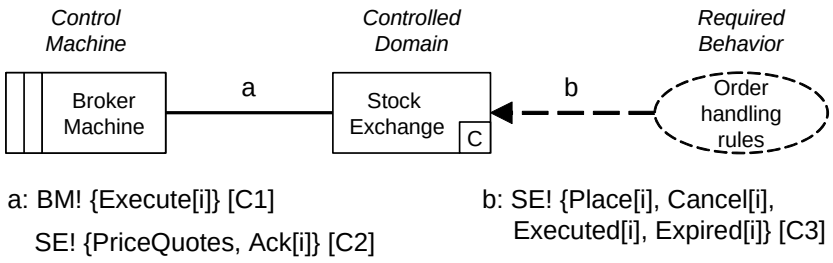


Figure 3-21: Example of a Required Behavior basic frame: handling of trading orders.

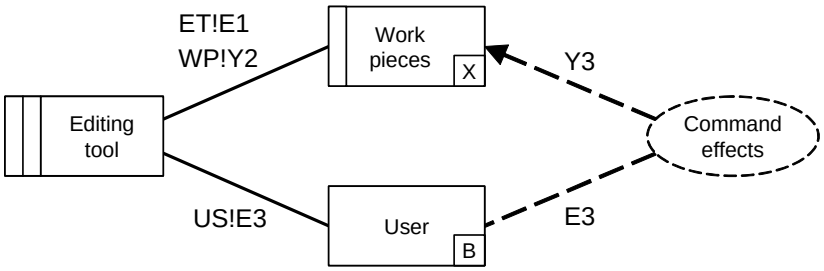


Figure 3-22: Problem frame diagram for the *simple workpieces frame*.

An example is shown in Figure 3-21 for how a stock-broker's system handles trading orders. Once the user places an order, the order is recorded in the broker's machine and from now on the machine monitors the quoted prices to decide when the conditions for executing the order are met. When the conditions are met, e.g., the price reaches a specified value, the controlled domain (stock exchange) is requested to execute the order. The controlled domain will execute the order and return an acknowledgement, known as "order ticket."

Basic Frame Type 2: Commanded Behavior

In this scenario, we need to build a machine which allows an operator to control the behavior of a part of the physical world by issuing commands.

Basic Frame Type 3: Information Display

In this scenario, we need to build a machine which acquires information about a part of the physical world and presents it at a given place in a given form.

Basic Frame Type 4: Simple Workpieces

In this scenario, we need to build a machine which allows a user to create, edit, and store some data representations, such as text or graphics. The lexical domain that will be edited may be relatively simple to design, such as text document for taking notes. It may also be very complex, such as creating and maintaining a "social graph" on a social networking website. A videogame is another example of a very complex digital (lexical) domain that is edited as the users play and issue different commands.

Figure 3-22 shows the frame diagram for the simple workpieces frame.

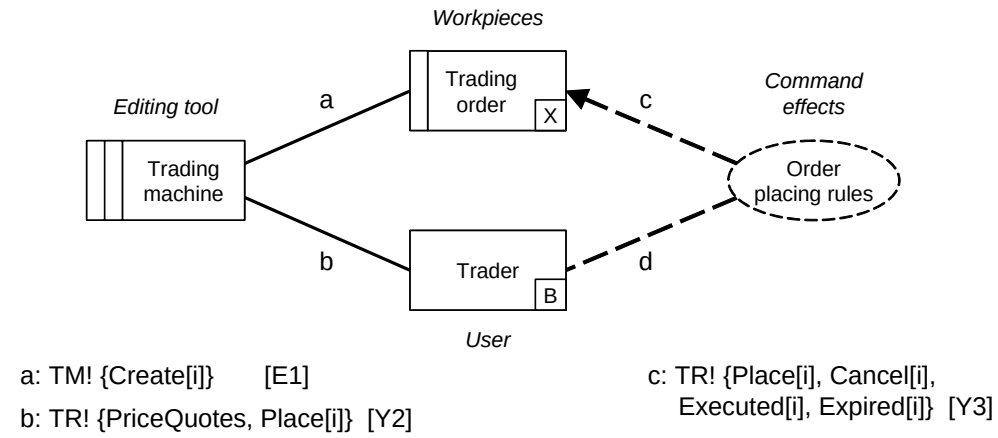


Figure 3-23: Example of a Simple Workpieces basic frame: placing a trading order.

An example is shown in Figure 3-23.

Basic Frame Type 5: Transformation

In this scenario, we need to build a machine takes an input document and produces an output document according to certain rules, where both input and output documents may be formatted differently. For example, given the records retrieved from a relational database, the task is to render them into an HTML document for Web browser display.

A key concern for a transformation frame problem is to define the order in which the data structures of the input data and output data will be traversed and their elements accessed. For example, if the input data structure is a binary tree, then it can be traversed in pre-order, in-order, or post-order manner.

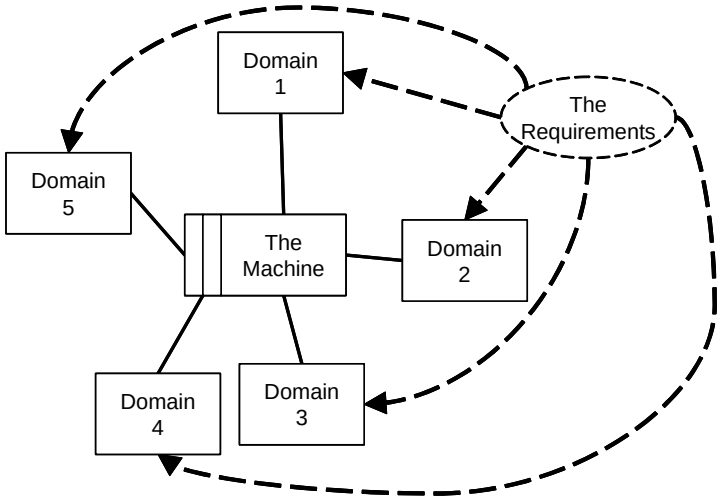
Figure 3-24 shows the key idea behind the frame decomposition. Given a problem represented as a complex set of requirements relating to a complex application domain, our goal is to represent the problem using a set of basic problem frames.

3.3.3 Composition of Problem Frames

Real-world problems almost always consist of combinations of simple problem frames. Problem frames help us achieve understanding of simple subproblems and derive solutions (machines) for these problem frames. Once the solution is reached at the local level of specialized frames, the integration (or composition) or specialized understanding is needed to make a coherent whole.

There are some standard *composite frames,* consisting of compositions of two or more simple problem frames.

(a)



(b)

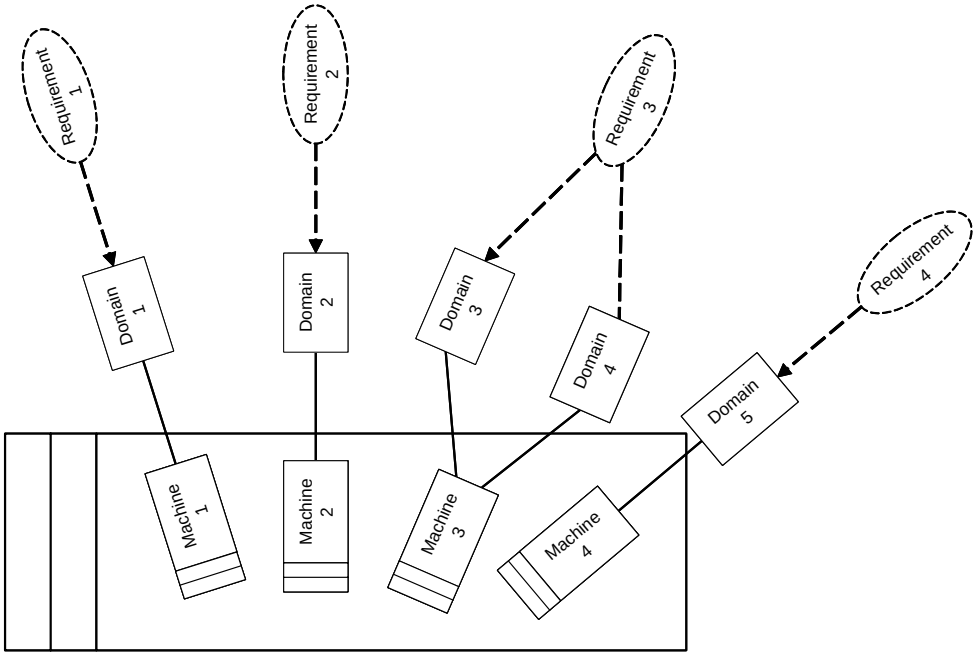


Figure 3-24: The goal of frame decomposition is to represent a complex problem (a) as a set of basic problem frames (b).

3.3.4 Models

Software system may have world representation and this is always idealized. E.g., in our lock system, built-in (as opposed to runtime sensed/acquired) knowledge is: IF valid key entered AND sensing dark THEN turn the light on.

3.4 Specifying Goals

"Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little." —Bertrand Meyer

The basic idea of goal-oriented requirements engineering is to start with the aggregate goal of the whole system, and to *refine* it by successive steps into a goal hierarchy.

AND-OR refinements …

Problem frames can be related to goals. Goal-oriented approach distinguishes different kinds of goal, as problem-frames approach distinguishes different problem classes. Given a problem decomposition into basic frames, we can restate this as an AND-refinement of the goal hierarchy: to satisfy the system requirement goal, it is necessary to satisfy each individual subgoal (of each basic frame).

When programmed, the program "knows" its goals *implicitly* rather than *explicitly*, so it cannot tell those to another component. This ability to tell its goals to others is important in autonomic computing, as will be seen in Section 9.3.

State the goal as follows: given the states A=armed, B=lightOff, C=user positively identified, D=daylight

(Goal is the equilibrium state to be reached after a perturbation.)

Initial state: $A \wedge B$, goal state: $\neg A \wedge \neg B$.

Possible actions: α—setArmed; α^{-1}—setDisarmed; β—setLit; β^{-1}—setUnlit

Preconditions for α^{-1}: C; for β: D

We need to make a plan to achieve $\neg A \wedge \neg B$ by applying the permitted actions.

Program goals, see also "fuzzy" goals for multi-fidelity algorithms, MFAs, [Satyanarayanan & Narayanan, 2001]. http://www.cs.yale.edu/homes/elkin/ (Michael Elkin)

The survey "Distributed approximation," by Michael Elkin. *ACM SIGACT News*, vol. 36, no. 1, (Whole Number 134), March 2005. http://theory.lcs.mit.edu/~rajsbaum/sigactNewsDC.html

The purpose of this formal representation is not to automatically build a program; rather, it is to be able to establish that a program meets its specification.