Formal Methods in Software Engineering

Course Notes for CISC/CMPE 422/835

Fall 2022

School of Computing
Queen's University
Kingston, Ontario

September 2022

Contents

List of Figures

Part I

Introduction and Overview (Week 1)

Chapter 1

Complexity

> *"Complexity, I would assert, is the biggest factor involved in anything having to do with the software field."*
>
> Robert L. Glass [Gla02]

> *"The problem is that we are attempting to build systems that are beyond our ability to intellectually manage."*
>
> Nancy G. Leveson [Lev11]

The development of computing is remarkable in many ways, and perhaps most of all in its progress and impact. However, due to the economic significance of computing and the pace of societal and technological change, we are constantly presented with new questions, challenges, and problems, giving us little time to reflect on how far we have come. Also, computing has become such a large and fragmented field that it is impossible to keep abreast all research developments.

To motivate the contents of the course, we want to discuss a phenomenon what has been and, in all likelihood, will continue to be of central importance to computing and software engineering: complexity [Din16].

In general, complex systems are characterized by a large number of entities, components or parts, many of which are highly interdependent and tightly coupled such that their combination creates synergistic, emergent, and non-linear behaviour [HD01]. One of the prime examples of a complex system is the human brain consisting, approximately, of 10^{11} neurons connected by 10^{15} synapses [Chu]. Other examples can be found in many different domains including law (e.g., in 2014 it took a 74,000-page document to describe the US federal code [Sau14]), public infrastructure (e.g., the US road system has over 300,000 intersections with traffic lights [oTFHAU]), manufacturing (e.g., a Boeing 747-400 consists of 6 million individual parts and 171 miles of wiring [Kel11]), and, of course, software.

	Lines of code (approx.) (in million)
Operating Systems	
Windows NT 3.1 (1993)	0.5
Windows 95	11
Windows 2000	29
Windows XP (2001)	35
Windows Vista (2007)	50
Windows 7	40
Mac OS X	85
Android OS	12
Automobiles	
1981	0.05
2005	10
2015 (high end)	100
Miscellaneous	
Pacemaker	0.1
Mars Curiosity Rover	5
Firefox	10
Intuit Quickbook	10
Boeing 787	14
Airbus 380	120
F-35 fighter jet	24
Large Hadron Collider	50
Facebook	60
Google (gmail, maps, etc)	2,000

Figure 1.1: Approximate size of software in various products [Cha05, McCb, Bal14]

Figure 1.1 shows the size of software in different kinds of products. Noteworthy here are not only the absolute numbers, but also the rate of increase. Automotive software is a good example here. Just over 40 years ago, cars were devoid of software. In 1977, the General Motors Oldsmobile Tornado pioneered the first production automotive microcomputer ECU: a single-function controller used for electronic spark timing. By 1981, General Motors was using microprocessor-based engine controls executing about 50,000 lines of code across its entire domestic passenger car production. Since then, the size, significance, and development costs of automotive software has grown to staggering levels: Modern cars can be shipped with as much as 1GB of software encompassing more than 100 million lines of code; experts estimate that more than 80% of automotive innovations will be driven by electronics and 90% thereof by software, and that the cost of software and electronics can reach 40% of the cost of a car [Gri03].

The history of avionics software tells a similar story: Between 1965 and 1995, the amount of software in civil aircraft has doubled every two years [dM93]. If growth continues at this pace, experts believe that limits of affordability will soon be reached [War13].

Lines of code is a doubtful measure of complexity[1]. Nonetheless, it appears fair to say the modern software is one of the most complex man-made artifacts.

1.1 Why has software complexity increased so much?

An enabler necessary for building and running modern software certainly is modern hardware. Today's software could not run on yesterday's hardware. The hardware industry has produced staggering advances in chip design and manufacturing which have managed to deliver exponentially increasing computing power at exponentially decreasing costs. Compared to the Apollo 11 Guidance Computer used 1969[2] a standard smart phone from 2015 (e.g., iPhone 6) has several tens of million of times the computational power (in terms of instructions per second)[3]. In 1985, an 2011 iPad2 would have rivaled a four-processor version of the Cray 2 supercomputer in performance, and in 1994, it still would have made the list of world's fastest supercomputers [Mar11]. According to [McCa], the price of a megabyte of memory dropped from US$411,041,792 in 1957 to US$0.0037 in December 2015 — a factor of over 100 billion! The width of each conducting line in a circuit (approx. 15 nanometers) is approaching the width of an atom (approx. 0.1 to 0.5 nanometers).

But, it is not just technology that is getting more complex, life in general does, too. According to anthropologist and historian Josef Tainter, "the history of cultural complexity is the history of human problem solving" [Tai96]. Societies get more complex because "complexity is a problem solving strategy that emerges under conditions of compelling need or perceived benefit". Complexity allows us to solve problems (e.g., food or energy distribution) or enjoy some benefit. Ideally, this benefit is greater than the costs of creating and sustaining the complexity introduced by the solution.

1.2 Consequences of software complexity

On the positive side, complex systems are capable of impressive feats. AlphaGo, the Go playing system that in March 2016 became the first program to beat a professional human Go player without handicaps on a full-sized board in a five-game match, was said by experts to be capable of developing its own moves: "All but the very best Go players craft their style by imitating top players. AlphaGo seems to have totally original moves it creates itself" [BL16], providing a great example of — seemingly or real — emergent, synergistic behaviour.

[1] So many alternative ones have been proposed [Rig96] that even the study of complexity appears complex

[2] A web-based simulator can be found at `http://svtsim.com/moonjs/agc.html`

[3] https://www.quora.com/How-much-more-computing-power-does-an-iPhone-6-have-than-Apollo-11-What-is-another-modern-object-I-can-relate-the-same-computing-power-to

On the negative side, complexity increases risk of failure. Examples of software failures are numerous [Blo18, Arb16, oCP, Pet96]. Some of the more high-profile ones include

- the Therac-25 radiation accidents between 1985 and 1987 in which patients received radiation doses hundreds of times greater than normal [LT93],
- the Ariane 5 explosion in 1996 which destroyed the rocket and its cargo valued at $500 million [LLF+96],
- the Northeast Blackout in 2003 which affected 50 million people, contributed to the death of eleven people and cost an estimated US$6 billion [Min08],
- the Flash Crash on May 6, 2010, causing a trillion dollars in lost value for a short amount of time [Bos14, Wea15],
- Toyota's unintended acceleration issue which killed multiple people [Bar14, Koo14, Arb16].

Data on the failures of software or software development are hard to come by. According to the US National Institute of Standards and Technology, the cost of software errors in the US in 2001 was US$ 60 billion [RTI02] and in 2012 the worldwide cost of IT failure has been estimated to be $3 trillion[4].

A recent example illustrates how subtle bugs can be and how difficult it is to build software systems correctly: Chord is a protocol and algorithm for a peer-to-peer distributed hash table first presented in 2001 [SMK+01]. The work identified relevant properties and provided informal proofs for them in a technical report. Chord has been implemented many times[5] and went on to win the SIGCOMM Test-of-Time Award in 2011. The original paper currently has over 12,000 citations on Google scholar and is listed by CiteSeer as the 9th most cited Computer Science article. In 2012, it was shown that the protocol was not correct [Zav12].

1.3 The future is expected to bring even more reliance on even more complex software

The extent to which virtually all aspects of modern society depend on software is impressive and, it appears, will only increase. For instance, the very significant reliance of companies in general on software to stay competive has been discussed widely [Kir11, And11]. It appears that this reliance will only grow [Pat16]. For instance, a recent New York Times article "G.E., the 124-Year-Old Software Start-Up", illustrates the expected impact and reliance on large, complex software in manufacturing [Loh16]. Similar observations could be made about the importance of software for public infrastructure [Str12].

[4] http://www.zdnet.com/article/worldwide-cost-of-it-failure-revisited-3-trillion

[5] At least 8 implementations are listed at https://github.com/sit/dht/wiki/faq

Another example is the Internet of Things (IoT), i.e., the idea to equip everyday objects with sensors and computing power to collect, aggregate and analyze different kinds of information to improve the operation, maintenance, and evolution of a system, which has captured the imagination of practitioners and researchers in many domains. In manufacturing, for instance, IoT applications promise advances in supply chain management, predictive maintenance, fault diagnosis and correction, operation optimization, and product evolution. Similarly, the management, maintenance, and evolution of infrastructure in buildings and cities could benefit greatly from the IoT. In health care, IoT-based home automation and monitoring systems might help improve quality of life, emergency response, lifestyle monitoring (e.g., via diet-aware glasses [HWZ17]), treatment delivery, and diagnosis for people with special medical conditions. Almost every aspect of modern life is expected to be impacted (apart from manufacturing, smart buildings, smart cities, transportation, and health care, other potential areas of application include defense, insurance, agriculture, retail, logistics, banking, food services, hospitality). Insiders project that, by 2020, there will be over 20 billion IoT devices and about $6 trillion investment in IoT technology, and that, overall, "the Internet of Things will be as transformative to the world as was the Industrial Revolution" [BI 16]. Given the large-scale and diverse use that the IoT is expected to have, including in applications that can be considered safety-critical, the potential damage caused by badly designed, malfunctioning, or vulnerable IoT software is significant. Together with the expected complexity of IoT software and the wide range of requirements it has to satisfy, there is evidence that "we will need some meaningfully trustworthy hardware and software components, and much better development and deployment practices than we have at present to enable the IoT to provide adequate human safety, security, reliability, usability, and satisfied users" [LN17].

Our expected reliance on complex, perhaps poorly understood and buggy, software has been cause for concern. For instance, in 1999, the advisory committee to the US President on Information Technology described the situation as follows [Com99]:

> "The demand for software has grown far faster than our ability to produce it [...] Furthermore, the nation needs software that is far more usable, reliable, and powerful than what is being produced today [...] We have become dangerously dependent on large software systems whose behaviour is not well understood [...] Increases in research on software should be given high priority [...] [We should] fund fundamental research in software development methods and component technologies [...]"

In "Geekonomics: The Real Cost of Insecure Software" the author is motivated by the same topic [Ric07]. A more recent account can be found in [Som17].

1.4 How to deal with software complexity

Computer science curricula teach students a combination of techniques to deal with complexity, the most prominent of which are decomposition, abstraction, reuse, automation, and analysis. Of these, abstraction, automation, and analysis lie at the heart not only of software development, but also of formal methods. Without some kind of formal modeling though, analysis is reduced to the standard software quality assurance techniques of which testing is, by far, the most prominent (followed, probably, by static analysis and inspection). But, as we will see, testing has limitations that, for certain kinds of software at least (such as *safety-critical* or *mission-critical*), are not acceptable. For these, some industries have put certification guidelines and procedures in place that require the creation and use of many different artifacts (e.g., requirements specifications, test cases, design documents, failure models) throughout the development lifecycle in a structured, organized fashion. Examples of certification guidelines include DO-178C and ISO 26262 for avionics and automotive software, respectively, as well as the guidelines published by the U.S. Food and Drug Adminstration for the development of software in medical devices [RTC12, (IS11, oHHSA02] Many of these certification activities either require or encourage the use construction and use of formal models of the kind presented in this course.

Chapter 2

Formal methods

The by far most widely used software quality assurance technique is testing. Testing alone, however, is often not sufficient. Reasons include the complexity of software which often makes exhaustive and even sufficiently comprehensive testing infeasible, and the fact that testing requires executable code which complicates finding bugs in artifacts that are produced before coding actually starts (e.g., requirements, designs).

Modern software development is a complex process that typically involves many different people and artifacts. To substantially improve the software produced, all parts of this process have to be addressed and be supported with automatic analysis tools. The techniques and tools discussed in this course cover the following phases:

- Requirements analysis and design: Studies have shown repeatedly that requirements and architecture, i.e., the early phases of software development, are responsible for the majority of defects (e.g., for safety-critical software reliant systems approximately 70% of defects are introduced during these phases and 80% of these defects are discovered late in the development cycle [RWCP10]). Consequently, during the requirements analysis and design phases it may be important to be able to write down the requirements precisely and unambiguously. Part III contains a short and incomplete introduction to the formal specifications. Then, Part IV presents *Alloy*, a language and tool for capturing and analyzing the structure, relationships, and evolution of collections of objects as found in object-oriented systems. Alloy is useful to prototype and validate designs and identify relevant properties.

- Implementation: During the implementation phase it may be necessary to ensure that a protocol satisfies certain properties. For many protocols, *model checking* (i.e., exhaustive state space exploration) is the perfect technique for that. One of the most popular forms of model checking (called temporal logic model checking) is discussed in Part VI.

- Testing: The purpose of testing is to determine to what extent the implemented software behaves as expected. Formal specifications can be very useful to describe exactly what constitutes expected behaviour, i.e., which properties the output produced by the software in response to some input is expected to have. *Property-based testing* (PBT) is an approach to testing that leverages these properties to increase the automation and effectiveness of testing. PBT is discussed in Part V.

Before we can discuss these three topics, however, Part II reviews the foundation on which these approaches are built: *propositional and predicate logic*. Why? As we will see, all of the analysis techniques presented are built on one or more formal specification languages, i.e., languages that allow the precise description of aspects of the software (e.g., the structure and relationship between objects, the behaviour of a protocol and properties, and the expected behaviour of a method). What makes these languages formal is the fact that not only their syntax but also their semantics is precisely defined (typically, by establishing some kind of mapping to an appropriate mathematical structure), which prevents ambiguities and facilitates the creation of effective analysis tools. Propositional and pedicate logic are two of the most well-known and understood formal languages and have greatly influenced the formal specification languages we will study. In other words, without properly understanding propositional and predicate logic, we will not be able to understand the formal specification languages covered in the course.

Formal methods are techniques and tools that use formal specification languages. They can help improve the quality of software development processes and their resulting software significantly and also reduce development time and costs [WLBF09, McC16]. They are based on formally defined, high-level notations to allow for rigorous, succinct expression of relevant aspects of the software and automatic analyses that check for desirable properties using decicated tools. As mentioned, abstraction, analysis and automation are the key weapons formal methods use to deal with complexity and bring more rigour to software development. The use of formal methods and modeling is explicitly encouraged by certification authorities such as the Federal Aviation Administration (FAA) in the US and Transport Canada through their DO-178C guidelines [RTC12]. They are in use in companies such as Amazon [NRZ+15], Facebook [O'H15] and Microsoft [Mic], and have helped ensure correctness of the 120 million lines of code in the Airbus A380 [Bal14]. The ACM's 'Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering' consider modeling and analysis of requirements and software a knowledge area, representing a particular subdiscipline of software engineering that is generally recognized as a significant part of the software engineering knowledge that an undergraduate should know [ACM15].

Part II

Propositional and Predicate Logic: A Review

Chapter 3

Propositional logic (Week 2)

Our presentation of propositional logic is standard. Sections 3.1 and 3.3 define a syntactic domain while Section 3.2 defines a semantic domain. Sections 3.4 and 3.5 connect the two domains by establishing soundness and completeness. Finally, Section 3.6 discusses the complexity of propositional logic and to what extent it can be implemented.

3.1 Syntax

3.1.1 Formulas

We begin by defining the formulas we can legally write. A *declarative* statement is either true or false. For instance, *"Tom hates logic"*, and *"The number 6 is prime"* are declarative statements. *"Go Steelers!"*, *"Could you please take the garbage out?"*, or *"May the force be with you"* are not declarative. An *atomic proposition* p is a declarative statement that cannot be decomposed such as *door_is_open*, *ac_on*, *temperature_high*, but also $x > 0$ and $x = 13$. Formulas are declarative statements that are built from a given set of atomic propositions AP and the two *connectives* $\neg$ and $\wedge$. The grammar in Backus-Naur form (BNF) for a formula φ is:

φ	::=	p |	an atomic proposition is a formula
		$(\neg\, \varphi)$ |	the *negation* of a formula is an formula
		$(\varphi \wedge \varphi)$ |	the *conjunction* of two formulas is a formula
		(φ)	a formula in matching parentheses is a formula

where $p \in AP$. Suppose, for instance, that the set of atomic propositions contains *ac_on*, *temperature_high*; then the above grammar allows the formation of the following formulas: *ac_on*, $\neg$ *ac_on*, (*ac_on* $\wedge$ *temperature_high*), etc.

Note that this definition is *inductive* (*recursive*) with one base case and three inductive cases.

To save parentheses, the connectives are assigned the following *binding priorities*:

$\neg$	negation binds most tightly
$\wedge$	conjunction binds least tightly

Thus, for instance, $\neg\, p \wedge q$ is equivalent to $((\neg\, p) \wedge q)$.

3.2 Semantics

What does a formula φ mean? To answer this question, we present a semantic model. More precisely, we fix a semantic domain $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ with two elements: *true* to represent semantic truth and *false* to represent semantic falsehood. Then, we show how the meaning of a formula can be obtained by evaluating it to one of the values in $\mathbb{B}$. Before we can do that, though, we need to determine what our two connectives $\neg$ and $\wedge$ and atomic propositions mean.

The meaning of the connectives is given by the following *truth tables*

φ	$\neg\, \varphi$
true	*false*
false	*true*

φ	ψ	$\varphi \wedge \psi$
true	*true*	*true*
true	*false*	*false*
false	*true*	*false*
false	*false*	*false*

This table tells us, for instance, that whenever the meaning of a formula φ is *true*, then the meaning of $\neg\, \varphi$ is *false*, whereas the meaning of $\varphi \wedge \varphi$ is *true*.

What, however, does an atomic proposition such as p mean? How is it to be evaluated? The answer is: It depends, p can evaluate to *true* or *false*, and we need additional information to tell us to which. Given a formula φ, a *model* for φ, sometimes also called a *truth assignment* or *valuation*, is a mapping

$$l : AP \to \mathbb{B}$$

where AP is the set of atomic propositions in φ.

Exercise 3.2.1. How many models does a formula have?

Given a model l for φ, we are now fully equipped to precisely define how to evaluate φ with respect to l. There are two equivalent ways to describe this evaluation: one using a *satisfaction relation*, and one using an *evaluation function*. Both notions are defined inductively over the structure of the formula φ.

1. **Satisfaction relation $\models$**
   The so-called *satisfaction relation* $\models$ describes when a formula holds in a model. Formally, $\models$ is a binary relation between models (that is, elements in $AP \to \mathbb{B}$) and formulas. More precisely, let l be a model of φ. φ holds

in l if and only if $l \models \varphi$, which is short for $(l, \varphi) \in \models$. If φ does not hold in l, that is, $(l, \varphi) \notin \models$, we write $l \not\models \varphi$. The relation $\models$ is defined inductively over the structure of φ using the truth table definition of the connectives. More precisely, $\models$ is a binary relation between models l and formulas φ such that

$$\begin{array}{ll} l \models p & \text{iff } l(p) = \mathit{true} \\ l \models \neg\, \varphi & \text{iff } l \not\models \varphi \\ l \models \varphi \wedge \psi & \text{iff } l \models \varphi \text{ and } l \models \psi. \end{array}$$

For instance, the mapping l defined by

$$\begin{array}{rcl} l(p) & = & \mathit{true} \\ l(q) & = & \mathit{false} \\ l(r) & = & \mathit{true} \end{array}$$

is a model for the formula $\neg\ (p \wedge q) \wedge (r \wedge p)$. l makes this formula true, that is, we have $l \models \neg\ (p \wedge q) \wedge (r \wedge p)$. This is because l makes $\neg\ (p \wedge q)$ true (we have $l \models \neg\ (p \wedge q)$) and $r \wedge p$ true (we have $l \models r \wedge p$). Similarly, l is also a model for $((p \wedge \neg\ q) \wedge r)$. However, $((p \wedge q) \wedge r)$ does not hold in l, that is, we have $l \not\models ((p \wedge q) \wedge r)$.

2. **Evaluation function** *eval*
An alternative, but equivalent account of the semantics of a formula is obtained by using an *evaluation function* that maps every formula to a truth value. Given a truth assignment l, the meaning $\mathit{eval}_l(\varphi)$ of a formula φ is defined inductively by

$$\begin{array}{rcl} \mathit{eval}_l(p) & = & l(p) \\ \mathit{eval}_l(\neg\, \varphi) & = & \mathit{eval}_\neg\ (\mathit{eval}_l(\varphi)) \\ \mathit{eval}_l(\varphi \wedge \psi) & = & \mathit{eval}_\wedge(\mathit{eval}_l(\varphi), \mathit{eval}_l(\psi)) \end{array}$$

where $\mathit{eval}_\neg$ and $\mathit{eval}_\wedge$ represent two functions implementing the truth tables for negation and conjunction respectively. That is, $\mathit{eval}_\neg$ is defined by

$$\begin{array}{rcl} \mathit{eval}_\neg\ (\mathit{true}) & = & \mathit{false} \\ \mathit{eval}_\neg\ (\mathit{false}) & = & \mathit{true} \end{array}$$

and $\mathit{eval}_\wedge$ is defined by

$$\mathit{eval}_\wedge(x, y) \quad = \quad \begin{cases} \mathit{true}, & \mathit{if}\ x = y = \mathit{true} \\ \mathit{false}, & \mathit{otherwise} \end{cases}$$

Using the truth assignment l defined above, we thus get

$$\begin{array}{rcl} \mathit{eval}_l\big(\neg\ (p \wedge q) \wedge (r \wedge p)\big) & = & \mathit{true} \\ \mathit{eval}_l((p \wedge q) \wedge r) & = & \mathit{false} \end{array}$$

The connection between the two definitions is that for all truth assignments l, we have $l \models \varphi$ if and only if $eval_l(\varphi) = true$.

Exercise 3.2.2. Prove that $l \models \varphi$ if and only if $eval_l(\varphi) = true$. Hint: Use structural induction (as covered in CISC 360) over the structure of φ.

3.2.1 Satisfiability, validity, entailment, and counter examples

Given a formula φ, every truth assignment l that makes φ true (that is, we have $l \models \varphi$ or $eval_l(\varphi) = true$) is called a *satisfying assignment of* φ. For instance, the formula $((p \wedge q) \wedge \neg r)$ has the satisfying assignment l defined as

$$\begin{aligned} l(p) &= true \\ l(q) &= true \\ l(r) &= false \end{aligned}$$

For the formula $\neg (p \wedge \neg p)$ every truth assignment is a satisfying assignment. The formula $p \wedge \neg p$ has no satisfying assignments. A formula φ is called *satisfiable* if it has a satisfying assignment. For instance, p and $\neg (p \wedge \neg p)$ are satisfiable. A formula is called *unsatisfiable* if it is not satisfiable. For instance, $p \wedge \neg p$ is unsatisfiable.

While satisfiability expresses the truth of a formula in one particular model, validity says that a formula is true in all its models. A formula φ is called *valid*, $\models \varphi$ for short, if φ evaluates to true under all truth assignments, that is, $l \models \varphi$ for all l. For instance, out of the formulas p, $p \wedge \neg p$ and $\neg (p \wedge \neg p)$ only $\neg (p \wedge \neg p)$ is valid. Note that φ is valid if and only if $\neg \varphi$ is not satisfiable. Valid formulas are sometimes also called *tautologies*.

Remember that the symbol "$\vdash$" expresses derivability and is thus a syntactic notion. Now that we have a semantic model, we can complement it with a semantic notion called *entailment*. More precisely, the notation

$$\varphi_1, \varphi_2, \ldots, \varphi_n \models \psi$$

expresses that whenever all premises φ_i evaluate to true for all $1 \leq i \leq n$, then ψ will, too. More precisely, for all truth assignments l, whenever $l \models \varphi_i$ holds for all $1 \leq i \leq n$ (or, equivalently, $eval_l(\varphi_i) = true$ for all $1 \leq i \leq n$) then $l \models \psi$ holds as well (or, $eval_l(\varphi) = true$).

Conversely, we say that $\varphi_1, \varphi_2, \ldots, \varphi_n$ does not entail ψ,

$$\varphi_1, \varphi_2, \ldots, \varphi_n \not\models \psi$$

if there exists a truth assignment l such that $l \models \varphi_i$ holds for all $1 \leq i \leq n$, but $l \not\models \psi$. In this case, l is said to be a *counter example* for the sequent.

3.2.2 New connectives as abbreviations

The attentive reader might be missing other standard connectives like disjunction or implication. It turns out that these and other connectives can be obtained through the following abbreviations.

$$
\begin{array}{rcll}
\mathit{ff} & \stackrel{def}{=} & \varphi \wedge \neg\, \varphi & \textit{falsehood, contradiction} \\
\varphi \vee \psi & \stackrel{def}{=} & \neg\, (\neg\, \varphi \wedge \neg\, \psi) & \textit{disjunction} \\
\mathit{tt} & \stackrel{def}{=} & \varphi \vee \neg\, \varphi & \textit{truth} \\
\varphi \rightarrow \psi & \stackrel{def}{=} & \neg\, \varphi \vee \psi & \textit{implication} \\
\varphi \leftrightarrow \psi & \stackrel{def}{=} & (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) & \textit{equivalence}
\end{array}
$$

Note that *tt* and *ff* are syntactic symbols, whereas *true* and *false* are semantic. From the above definitions we can read off the truth table definitions of the new connectives.

φ	ψ	$\varphi \vee \psi$
true	*true*	*true*
true	*false*	*true*
false	*true*	*true*
false	*false*	*false*

φ	ψ	$\varphi \rightarrow \psi$
true	*true*	*true*
true	*false*	*false*
false	*true*	*true*
false	*false*	*true*

φ	ψ	$\varphi \leftrightarrow \psi$
true	*true*	*true*
true	*false*	*false*
false	*true*	*false*
false	*false*	*true*

The binding priorities of the new connectives are:

$\neg$	binds most tightly
$\wedge, \vee$	bind tighter than $\rightarrow$, but less than $\neg$
$\rightarrow$	binds tighter than $\leftrightarrow$, but less than $\wedge$ and $\vee$
$\leftrightarrow$	binds least tightly

For instance, $\neg\, p \vee q \rightarrow p \leftrightarrow q$ stands for $(((\neg\, p) \vee q) \rightarrow p) \leftrightarrow q$.

When determining the value of a formula φ containing new connectives, we now have a choice:

- In a preprocessing step, we replace every subformula in φ that contains a new connective by the subformula it abbreviates. For instance, if φ is $p \rightarrow (p \vee q)$ we obtain

$$
\begin{array}{rcl}
p \rightarrow (p \vee q) & = & \neg\, p \vee (p \vee q) \\
& = & \neg\, (\neg\, \neg\, p \wedge \neg\, (p \vee q)) \\
& = & \neg\, (\neg\, \neg\, p \wedge \neg\, \neg\, (\neg\, p \wedge \neg\, q))
\end{array}
$$

  The resulting φ' formula only contains atomic propositions, negation, and conjunction and thus the evaluation function or the satisfaction relation presented on page 14 can be used to determine the value of φ'.

- Alternatively, we augment the definition of the evaluation function and the satisfaction relation to handle the new connectives directly. For instance,

the value of a disjunction $\varphi \vee \psi$ is *false* if both subformulas evaluate to *false*. Otherwise, it is *true*. More formally,

$$eval_l(\varphi \vee \psi) \quad = \quad eval_\vee(eval_l(\varphi), eval_l(\psi))$$

where $eval_\vee$ is defined as

$$eval_\vee(v_1, v_2) \quad = \quad \begin{cases} \mathit{false}, & \mathit{if}\ v_1 = v_2 = \mathit{false}, \\ \mathit{true}, & \mathit{otherwise}. \end{cases}$$

Exercise 3.2.3. Augment the definitions of the satisfaction relation $l \models$ and the evaluation function such that they can handle formulas containing tt, ff, $\vee$, $\rightarrow$, or $\leftrightarrow$ directly.

3.2.3 De Morgan laws

Sometimes, the connectives $\vee$ and $\rightarrow$ are defined directly rather than in terms of $\neg$ and $\wedge$. The above definitions then arise as equivalences in the context of the *De Morgan laws*. Here are some of them:

$$\begin{array}{rcl} \neg\,(\neg\,\varphi) & \leftrightarrow & \varphi \\ \neg\,(\varphi \vee \psi) & \leftrightarrow & \neg\,\varphi \wedge \neg\,\psi \\ \neg\,(\varphi \wedge \psi) & \leftrightarrow & \neg\,\varphi \vee \neg\,\psi \\ \varphi \rightarrow \psi & \leftrightarrow & \neg\,\varphi \vee \psi \\ \varphi_1 \wedge (\varphi_2 \vee \varphi_3) & \leftrightarrow & (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3) \\ \varphi_1 \vee (\varphi_2 \wedge \varphi_3) & \leftrightarrow & (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3) \end{array}$$

Adequate connectives

A set of connectives is called *adequate* for a logic, if for every formula in that logic there exists an equivalent formula with only connectives from that set. When we defined the syntax and semantics we made use of the fact that the set of connectives $\{\neg, \wedge\}$ is adequate for propositional logic. Other adequate sets are $\{\neg, \vee\}$, $\{\neg, \rightarrow\}$, and $\{\rightarrow, ff\}$. In general, if $C \subseteq \{\neg, \wedge, \vee, \rightarrow, ff\}$ and C is adequate, then $\neg \in C$ or $ff \in C$. What are the advantages and disadvantages of defining a logic using a set of adequate connectives and obtaining the remaining connectives as abbreviations? Advantages include:

- The relationship between the connectives is revealed.
- The BNF is shorter. Consequently, proofs over the structure of a formula are shorter, because there are fewer cases to consider.
- The definition of *eval* is shorter. Consequently, the implementation of formula evaluation is easier.

Disadvantages include:

- The intuitive meaning of a connective may be obscured. For instance, abbreviating $\neg\, \varphi \vee \psi$ by $\varphi \to \psi$ may be surprising and counterintuitive at first.
- The evaluation of a formula requires a preprocessing step in which occurrences of derived connectives are replaced by their corresponding expansion. The resulting formula is longer by a constant factor. Consequently, manual and automatic evaluations take longer.

3.2.4 Necessary and sufficient conditions

Given two statements φ and ψ, φ is said to be *sufficient* for ψ, if ψ is true whenever φ is. That is, the implication $\varphi \to \psi$ holds. Moreover, φ is *necessary* for ψ, if φ is true whenever ψ is. That is, the reverse implication $\psi \to \varphi$ holds. Consider, for instance, the statement

"If you have a Canadian passport, you are allowed to work in Canada".

In other words, having a Canadian passport is *sufficient* for being allowed to work in Canada. Since you may also work in Canada with a work permit or permanent residency status, a passport is not *necessary* for a work authorization, that is, the reverse implication

"If you are allowed to work in Canada, you have a Canadian passport"

is not true. Note, however, that *"You are allowed to work in Canada"* is necessary for *"You have a Canadian passport"*, that is, if φ is sufficient for ψ, then, by definition, ψ is necessary for φ.

If the truth of statement φ is both necessary and sufficient for the truth of another statement ψ, then the two are logically equivalent, that is, φ holds if and only if ψ does, that is, $\varphi \leftrightarrow \psi$ holds. For instance, the statement

"Tom can sing 'The Star-spangled Banner' "

is necessary and sufficient for the statement

"Tom can sing the American national anthem",

because both are logically equivalent.

3.3 Proof theory

We present the proof theory of propositional logic using *natural deduction*. The proof theory of a logic shows how a new formula can be obtained from other, old formulas by means of a syntactic process called *derivation*.

3.3.1 Sequents

Let $\varphi_1, \ldots, \varphi_n$ and ψ be formulas. Then,

$$\varphi_1, \ldots, \varphi_n \vdash \psi$$

is called a *sequent*. Intuitively, this sequent means that the *conclusion* ψ is derivable from the *premises* $\varphi_1, \ldots, \varphi_n$ using the proof rules to be presented in Section 3.3. Note that a sequent is not a formula. It expresses a syntactic relationship between formulas.

3.3.2 Proof rules

By definition the sequent

$$\varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi$$

expresses that ψ is derivable from the formulas $\varphi_1, \varphi_2, \ldots, \varphi_n$ using a set of proof rules. We now present these rules.

General shape of rules

Every rule has the following general form:

$$\frac{\varphi_1 \quad \ldots \quad \varphi_n \qquad \boxed{\begin{array}{c}\psi_1\\ \vdots\\ \psi'_1\end{array}} \quad \ldots \quad \boxed{\begin{array}{c}\psi_m\\ \vdots\\ \psi'_m\end{array}}}{\eta} \; \textit{name of rule}$$

The formulas φ_1 through φ_n above the bar are called *premises* of the rule. The m boxes above the bar represent subproofs in which ψ'_i must be shown using ψ_i as additional assumption. This additional assumption must only be used inside the box, in other words, the box demarcates the scope of the assumption. The formula η below the bar is the *conclusion* of the rule. The name of the rule appears on the right. The rule allows us to add the conclusion η to our proof, but only if the premises φ_1 through φ_n already occur in the proof and all the m subproofs have been completed, that is, in every box i the vertical dots have been replaced by a valid proof of ψ'_i.

The following two special cases are worth mentioning:

- $n = 0$: The rule requires no premises.
- $m = 0$: The rule requires no subproofs.

If $n = m = 0$ the space above the bar is empty. There is no constraint on when the rule can be applied. Such a rule is often called *axiom* which is Latin for "that which is assumed".

Soundness of a rule

An important property of rules is *soundness*. Informally, soundness means the "stuff above the bar implies the conclusion". In other words, the rule never allows you to draw a false conclusion. Formally, the rule with the general form above is sound iff the premises φ_1 through φ_n and the implications $\psi_1 \to \psi_1'$ through $\psi_m \to \psi_m'$ entail the conclusion η, that is, we have

$$\varphi_1, \ldots, \varphi_n, \psi_1 \to \psi_1', \ldots, \psi_m \to \psi_m' \models \eta$$

Examples for sound rules are

$$\frac{\varphi \wedge \psi}{\varphi} \wedge e_1 \qquad \frac{\varphi \vee \psi \quad \begin{array}{|c|}\hline \varphi \\ \vdots \\ \eta \\ \hline \end{array} \quad \begin{array}{|c|}\hline \psi \\ \vdots \\ \eta \\ \hline \end{array}}{\eta} \vee e$$

Examples for unsound rules are

$$\frac{\varphi}{\varphi \wedge \psi} \qquad \frac{\varphi \vee \psi \quad \begin{array}{|c|}\hline \psi \\ \vdots \\ \eta \\ \hline \end{array}}{\eta}$$

The proof rules for Propositional Logic

All proof rules for Propositional Logic can be found in Figure 3.3 on page 36. For instance, the three proof rules for conjunction $\wedge$ are

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \wedge i \qquad \frac{\varphi \wedge \psi}{\varphi} \wedge e_1 \qquad \frac{\varphi \wedge \psi}{\psi} \wedge e_2$$

The rule $\wedge\ i$ allows us to conclude $\varphi \wedge \psi$ provided that we have already proved φ and ψ. Since it introduces a conjunction, it is called an *introduction rule*. This rule is obviously sound, because we have the entailment

$$\varphi, \psi \models \varphi \wedge \psi$$

as can easily be seen from the semantics of $\wedge$.

The rules $\wedge\ e_1$ and $\wedge\ e_2$ allow us to eliminate a conjunction. Given a conjunction, we can also write down each of the conjuncts. They are examples of *elimination rules*. Again, the soundness of these two rules follows from the semantics of $\wedge$.

Next, consider the rules for the introduction and elimination of an implication:

$$\frac{\boxed{\begin{array}{c}\varphi\\ \vdots\\ \psi\end{array}}}{\varphi \to \psi} \to i \qquad\qquad \frac{\varphi \qquad \varphi \to \psi}{\psi} \to e$$

The premise of rule $\to i$ requires us to derive ψ from φ. The box demarcates the scope of the temporary assumption φ, that is, the assumption φ must not be used outside the box. As we will see in Section 3.4, these boxes are necessary to ensure that our rules are sound. Rule $\to e$ is also known as "modus ponens".

Figure 3.3 on page 36 contains these and the other natural deduction proof rules for propositional logic. All rules should be fairly intuitive. Note that rule $ff\ e$ says that everything is derivable from a contradiction.

3.3.3 Proofs

Now we are ready to define when the application of these rules makes up a legal proof. Informally, a proof of a sequent $\varphi_1, \ldots, \varphi_n \vdash \varphi$ is a sequence of formulas $\psi_1, \ldots, \psi_m$ such that

- the last formula in the sequence is the desired conclusion, that is, $\psi_m = \varphi$, and
- every formula ψ_i that is not a premise, that is, $\psi_i \notin \{\varphi_1, \ldots, \varphi_n\}$, must be either
  - an exact copy of a formula that appeared earlier in the sequence that does not depend on an assumption whose box has already been closed, or
  - the consequence of a proof rule applied to formulas earlier in the sequence.

Writing this down precisely requires a little work. A *proof line* is a triple

$$(\textit{line-number}, \varphi, \textit{justification})$$

where *line-number* is a natural number, φ is a formula, and *justification* is either "*premise*", "*assumption*", "*copy*" or the name of a rule followed by one, two or three line numbers. A *set of proof boxes* is a set of pairs of line numbers

$$\{(a_1, b_1), \ldots, (a_n, b_n)\}$$

where a_i and b_i mark the beginning and end of box i, respectively, such that

- each box has at least size 1, that is, $a_i \leq b_i$ for all $1 \leq i \leq n$, and
- every pair of boxes (a_i, b_i) and (a_j, b_j) is either

  - *disjoint*, that is, $b_i < a_j$ or $b_j < a_i$, or
  - *nested*, that is, $a_i < a_j \leq b_j < b_i$ or $a_j < a_i \leq b_i < b_j$

  for all $1 \leq i, j \leq n$ with $i \neq j$.

The above box discipline ensures that subproofs are like subroutine calls: If a proof requires a subproof (through the rules $\vee\ e$, $\rightarrow\ i$, or $\neg\ i$), then that subproof must be completed before the containing proof can complete.

A *natural deduction proof* of the sequent $\varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi$ is a sequence S of proof lines together with a set B of proof boxes such that

- the lines in S are numbered consecutively starting with 1, and
- each proof line
$$(\textit{line-number}, \varphi, \textit{justification})$$
in S is either
  - a premise, that is, φ is one of $\varphi_1, \varphi_2, \ldots, \varphi_n$ and *justification* is "*premise*", or
  - an assumption at the beginning of a new box, that is, *justification* is "*assumption*" and there exists b such that (*line-number*, b) is in B, or
  - the copy of a previous formula that does not occur inside a box that has already been closed, that is, *justification* is "*copy(k)*" and there exists a proof line $(k, \varphi, \textit{justification}_k)$ such that $k <$ *line-number* and there is no box $(a, b) \in B$ such that $a \leq k \leq b \leq$ *line-number*, or
  - the consequence of a unary proof rule r ($\neg\neg\ e$, *ff* e, $\wedge\ e_1$, $\wedge\ e_2$, $\vee\ i_1$, or $\vee\ i_2$) applied to a formula that appears before the current line in the sequence and that does not occur inside a box that has already been closed, that is, *justification* is "*r(k)*" such that $k <$ *line-number* and φ arises from applying rule r to the formula in line k and there is no box $(a, b) \in B$ such that $a \leq k \leq b \leq$ *line-number*, or
  - the consequence of a unary proof rule r ($\rightarrow\ i$ or $\neg\ i$) applied to a box that appears before the current line in the sequence, that is, *justification* is "*r(k)*" such that φ arises from applying rule r to the box ending in line k, and $(k', k) \in B$ for some k', or
  - the consequence of a binary proof rule r ($\wedge\ i$, $\rightarrow\ e$, or $\neg\ e$) applied to two formulas that appear before the current line in the sequence and that do not occur inside a box that has already been closed, that is, *justification* is "*r(k,l)*" such that $k, l <$ *line-number* and φ arises from applying rule r to the formulas in lines k and l, and there is no box $(a, b) \in B$ such that $a \leq k \leq b \leq$ *line-number* or $a \leq l \leq b \leq$ *line-number*, or

  - the consequence of a tertiary proof rule r ($\vee$ e) applied to a formula and two boxes that appear before the current line in the sequence such that the formula does not occur inside a box that has already been closed, that is, *justification* is "*r(k,l,m)*" such that $k, l, m <$ *line-number* and φ arises from applying rule r to the formula in line k and two boxes $(l', l) \in B$ and $(m', m) \in B$ for some l', m', and there is no box $(a, b) \in B$ such that $a \leq k \leq b \leq$ *line-number*, and

- the desired conclusion ψ occurs in the sequence. Typically, the proof ends as soon as the conclusion has been derived.

If a sequent has a proof it is said to be *valid*.

3.3.4 Constructing proofs

Finding a proof for a sequent can be easy, tricky, or impossible. We always start with a proof that has a "hole", i.e., an incomplete part, in it. Suppose, for instance, we want to find a proof for the sequent $\varphi_1, \varphi_2 \vdash \psi$. We write down what we have, i.e., the premises, on top and what we need to get, i.e., the conclusion, at the bottom. In the middle, we leave a lot of space.

1	φ_1	*premise*
2	φ_2	*premise*
3	$\vdots$	
4	ψ	

Then, we start filling the space with proof steps that eventually connect the top with the bottom. We typically fill in the space from both directions: from the top by thinking about how what we already have can be used to conclude a new formula; from the bottom by thinking from which new (hopefully also derivable) formulas the conclusion can be obtained.

3.3.5 Examples

We illustrate the definition of a proof with some examples. We will represent a proof graphically. The correspondence between the graphical representation and the textual one used in the definition above should be obvious.

1. The following example is taken from [HR00]. The sequent $p \wedge q \rightarrow r \vdash$

$p \to (q \to r)$, for instance, has the following proof:

1	$p \wedge q \to r$	*premise*
2	p	*assumption*
3	q	*assumption*
4	$p \wedge q$	$\wedge\, i(2,3)$
5	r	$\to e(1,4)$
6	$q \to r$	$\to i(5)$
7	$p \to (q \to r)$	$\to i(6)$

2. The *contraposition* (or *contrapositive*) of an implication $\varphi \to \psi$ is $\neg\, \psi \to \neg\, \varphi$. We will show that every implication implies its contraposition. For instance, consider the sentence *"If you're French, then you're European"*. If this implication holds, then the sentence *"If you're not European, then you're not French"* is also true. We prove that the sequent $p \to q \vdash \neg\, q \to \neg\, p$ is valid.

1	$p \to q$	*premise*
2	$\neg\, q$	*assumption*
3	p	*assumption*
4	q	$\to e(1,3)$
5	ff	$\neg\, e(2,4)$
6	$\neg\, p$	$\neg\, i(5)$
7	$\neg\, q \to \neg\, p$	$\to i(6)$

3. Moreover, every contraposition also implies the original implication. For instance, if the contraposition *"If you're not European, then you're not French"* holds, then *"If you're French, then you're European"* is also true. The proof of $\neg\, q \to \neg\, p \vdash p \to q$ is like that of $p \to q \vdash \neg\, q \to \neg\, p$ except that it additionally requires the use of rule $\neg\, \neg\, e$.

4. The next example illustrates the use of the rules for disjunction. It is

taken from [HR00]. We prove $q \to r \vdash p \vee q \to p \vee r$.

1	$q \to r$	*premise*
2	$p \vee q$	*assumption*
3	p	*assumption*
4	$p \vee r$	$\vee\, i_1(3)$
5	q	*assumption*
6	r	$\to e(1,5)$
7	$p \vee r$	$\vee\, i_2(6)$
8	$p \vee r$	$\vee\, e(2,4,7)$
9	$p \vee q \to p \vee r$	$\to i(8)$

5. Our last example is closer to reality. Consider each of the following statements and their abbreviations. Let

   "If JD oversleeps and the printer is not working, then JD is late for class"
   "JD oversleeps"
   "JD is not late for class"

   be abbreviated by, respectively,

   $$JDos \wedge \neg\, pw \to JDlate$$
   $$JDos$$
   $$\neg\, JDlate$$

   We want to conclude from these facts that pw holds, that is, that the printer is working.

1	$JDos \wedge \neg\, pw \to JDlate$	*premise*
2	$JDos$	*premise*
3	$\neg\, JDlate$	*premise*
4	$\neg\, pw$	*assumption*
5	$JDos \wedge \neg\, pw$	$\wedge\, i(2,4)$
6	$JDlate$	$\to e(1,5)$
7	ff	$\neg\, e(3,6)$
8	$\neg\, \neg\, pw$	$\neg\, i(4,7)$
9	pw	$\neg\, \neg\, e(8)$

3.3.6 Remarks about proof rules

Rules for derived connectives

Remember that in Section 3.2.2 we introduced ff and the connectives $\vee$ and $\to$ as abbreviations of combinations of $\neg$ and $\wedge$. We could use these abbreviations to

obtain sound proof rules for ff, $\vee$, and $\rightarrow$. For instance, since $\varphi \vee \psi$ abbreviates $\neg\,(\neg\,\varphi \wedge \neg\,\psi)$ we could use

$$\dfrac{\boxed{\begin{array}{c}\neg\,\varphi \wedge \neg\,\psi \\ \vdots \\ ff\end{array}}}{\varphi \vee \psi}\ \vee\, i$$

instead of

$$\dfrac{\varphi}{\varphi \vee \psi}\ \vee\, i_1 \qquad \dfrac{\psi}{\varphi \vee \psi}\ \vee\, i_2$$

As can be seen in this example, the problem with using the rules obtained through the abbreviations is that they tend to be lot more difficult to use than the ones in Figure 3.3.

A resulting slight inconvenience of the new rules for the derived connectives $\vee$ and $\rightarrow$ is that they can now be read in two different ways. For instance, the rule $\rightarrow i$ in Figure 3.3, suggests that $\varphi \rightarrow \psi$ means "if φ holds, then ψ holds, too". However, if we read $\varphi \rightarrow \psi$ as an abbreviation of $\neg\,\varphi \vee \psi$, then $\varphi \rightarrow \psi$ means "either φ doesn't hold or ψ does". Both readings are obviously quite different. However, both are correct and can be shown to be equivalent.

Derived rules

A proof rule is *derived* if it is obtained using other rules. Intuitively, derived rules are similar to macros in programming. Figure 3.4 on page 36 lists a few useful derived rules where MT, RAA, and LEM stand for "modus tollens", "reducto ad absurdum", and "law of excluded middle" respectively. The rule MT, for instance, is derived, because we can find a proof of

$$\varphi \rightarrow \psi, \neg\,\psi \vdash \neg\,\varphi$$

just using the rules in Figure 3.3:

1	$\varphi \rightarrow \psi$	*premise*
2	$\neg\,\psi$	*premise*
3	φ	*assumption*
4	ψ	$\rightarrow e(1,3)$
5	ff	$\neg\, e(2,4)$
6	$\neg\,\varphi$	$\neg\, i(5)$

3.3.7 Proof techniques I

This is a good place to discuss some general techniques to prove a formula. Suppose we want to prove that $\varphi \vdash \psi$. Each technique will be described using a

"proof schema" for the above sequent. Note that the step denoted by ... may consist of several steps possibly using φ.

- Direct proof. ψ can be shown directly from φ. If, for instance, φ says "n is divisible by 4" and ψ says "n is divisible by 2", then ψ can be shown directly from φ (and the definition of "divisible"). To write the proof down formally, we would use the rules for implication:

1	φ	*premise*
2	φ	*assumption*
3	$\vdots$	...
4	ψ	...
5	$\varphi \to \psi$	$\to i(4)$
6	ψ	$\to e(1,5)$

- Proof by contradiction. We derive a contradiction from the premises φ and $\neg\,\psi$. In other words, if φ holds, then $\neg\,\psi$ cannot also be true. Thus, if we know that φ holds, then ψ must hold, too.

1	φ	*premise*
2	$\neg\,\psi$	*assumption*
3	$\vdots$	...
4	ff	...
5	ψ	*RAA(4)*

  Here is an example. Suppose, for instance, we want to show that Tom must be older than 16, if he is licensed to drive a car in Canada. Assuming Canada's traffic laws and that Tom has a license and is younger than 16 leads us to a contradiction. Therefore, if the laws apply and he has a license, he must be older than 16.

- Proof by case study. Suppose, we additionally know

$$\varphi \to \varphi_1 \vee \varphi_2$$

  This implication allows us to prove ψ by considering two cases. The boxes used by the elimination rule for disjunction $\vee\ e$ are written next to each

other.

1	φ				*premise*
2	$\varphi \to (\varphi_1 \vee \varphi_2)$				*premise*
3	$\varphi_1 \vee \varphi_2$				$\to e(2,1)$
4	φ_1	*assumption*		φ_2	*assumption*
5	$\vdots$	$\ldots$		$\vdots$	$\ldots$
6	ψ	$\ldots$		ψ	$\ldots$
7	ψ				$\vee\, e(3,6)$

For instance, suppose we want to prove that the trip from Kingston to Montreal will take at least 2 hours, if you use public transport. We consider every means of public transportation and try to conclude that the trip always takes at least 2 hours using that means of transportation.

The list omits proof by induction, because only universally quantified formulas can be proved inductively. In other words, induction is not applicable to propositional formulas.

Exercise 3.3.1. For each of the following sequents, either prove it using Jape or show a counter example for the sequent.

1. $p \vee q \vdash \neg\, (\neg\, p \wedge \neg\, q)$
2. $\vdash (\neg\, p \vee q) \to (p \to q)$
3. $\neg\, (p \vee q) \vdash \neg\, p \wedge \neg\, q$
4. $p \vee (q \wedge r) \vdash (p \vee q) \wedge (p \vee r)$
5. $p \vee (q \vee r) \vdash (p \vee q) \vee r$

3.4 Soundness

In Section 3.3.2 we already introduced the notion of soundness for rules. Intuitively, a rule is said to be sound if its application never resulted in a false conclusion. A proof theory is said to be *sound*, if everything that is derivable using the rules also is valid in the model. More precisely, the theory is sound, if the following holds: Whenever $\varphi_1, \ldots, \varphi_n \vdash \psi$ is derivable using the proof rules in the theory, then $\varphi_1 \ldots, \varphi_n \models \psi$. Before we show that our theory is indeed sound, we present an example for an unsound theory. Suppose, for instance, we dropped the side condition from the rule for implication introduction, that is, whenever the rule $\to i$ is used, we allow the assumption introduced at the beginning of the box to be used outside the box. We show that the resulting theory is unsound. Let the atomic proposition p stand for *"The sun is shining"*.

If we dropped the side condition from rule $\rightarrow i$, the following would be a legal proof of $\vdash p$.

1	p	*assumption*
2	p	*copy(1)*
3	$p \rightarrow p$	$\rightarrow i(1)$
4	p	$\rightarrow e(1,3)$ *without side condition*

Consequently, without any assumptions we can always conclude p, that is, that the sun is shining. However, p clearly is not valid. Not even in Kingston. More precisely, p evaluates to *false* under a truth assignment l with $l(p) = \mathit{false}$. Thus, p is derivable but not valid. The theory is unsound. With the side condition, however, our theory is indeed sound.

Theorem 1. (Soundness)
For every sequent $\varphi_1, \ldots, \varphi_n \vdash \psi$ that is derivable using the proof rules in Figures 3.3 and 3.4, we have $\varphi_1, \ldots, \varphi_n \models \psi$.

Proof: The proof proceeds by induction over the length of the proof of

$$\varphi_1, \ldots, \varphi_n \vdash \psi.$$

Base The shortest proof has length 1. In that case, no rule is applied and the sequent is of the form $\varphi_1, \ldots, \varphi_n \vdash \varphi_i$ for some $1 \leq i \leq n$. Since all φ_i hold by assumption, we have $\varphi_1, \ldots, \varphi_n \models \varphi_i$.

Step Let the length of the proof be $k+1$ for $k \geq 0$. Thus, at least one rule was applied. Let r be the rule that was used to obtain ψ, the last the formula in the proof. We perform a complete case study over r. We discuss only one case. The remaing cases are similar. Suppose $r = \wedge\, i$, that is, ψ is of the form $\psi_1 \wedge \psi_2$. Then, then we have two proofs: One for ψ_1 and the other for ψ_2. Their length must be less than $k+1$. Thus, by inductive hypothesis, $\varphi_1, \ldots, \varphi_n \models \psi_1$ and $\varphi_1, \ldots, \varphi_n \models \psi_2$. Consequently, $\varphi_1, \ldots, \varphi_n \models \psi$. ■

Thus, the inductive step consists of showing for each rule that if all premises of the rule evaluate to true, then the conclusion does too.

Exercise 3.4.1. Consider adding each of the following three rules to our proof theory.

$$\frac{}{\varphi} \quad \textit{NEW 1} \qquad \frac{\varphi \wedge \psi}{\varphi \vee \psi} \quad \textit{NEW 2} \qquad \frac{\varphi \vee \psi}{\varphi \wedge \psi} \quad \textit{NEW 3}$$

Which rules would make the theory unsound? Explain your answer, that is, argue why the rule is sound or show a formula that is derivable in the new theory but not valid.

3.5 Completeness

A proof theory is said to be *complete*, if whenever the entailment $\varphi_1, \ldots, \varphi_n \models \psi$ holds, then there exists a proof for the sequent $\varphi_1, \ldots, \varphi_n \vdash \psi$. The following proposition is strongly related to completeness and will aid in its proof.

Proposition 3.5.1. (Completeness)
Every tautology is also a theorem, that is, if $\models \psi$, then $\vdash \psi$ for all formulas ψ.

Proof: Suppose ψ contains n distinct atomic propositions $p_1, \ldots, p_n$. Ignoring the details, the proof proceeds as follows:

1. If ψ holds, we know that ψ evaluates to *true* in all 2^n rows of the truth table for ψ. Each of the rows can be written as a sequent. More precisely, for row k we have the sequent

   $$p'_1, \ldots, p'_n \vdash \psi$$

   where p'_i is p_i if the entry for p_i in row k is *true*, otherwise p'_i is $\neg\, p_i$. For instance, if ψ is $p \vee \neg\, p$, we obtain four sequents

   $$\begin{array}{l} p, q \vdash p \vee \neg\, p \\ p, \neg\, q \vdash p \vee \neg\, p \\ \neg\, p, q \vdash p \vee \neg\, p \\ \neg\, p, \neg\, q \vdash p \vee \neg\, p \end{array}$$

2. By induction over the structure of ψ, we can show that each of the above 2^n sequents is provable.

3. Using the law of excluded middle, these 2^n proofs can be assembled into a single proof of ψ. ■

Note that the above proof crucially depends on the fact that the number of atomic propositions in a formula is finite.

Theorem 2. (Completeness)
For every entailment $\varphi_1, \ldots, \varphi_n \models \psi$ that holds, there exists a proof of the sequent $\varphi_1 \ldots, \varphi_n \vdash \psi$ using the rules in Figure 3.3.

Proof: The proof proceeds as follows. Assume $\varphi_1, \varphi_2, \varphi_3, \ldots, \varphi_n \models \psi$.

1. A simple truth table argument shows that

   $$\models \varphi_1 \to (\varphi_2 \to (\varphi_3 \to (\ldots \to (\varphi_n \to \psi) \ldots)))$$

   also holds.

2. Using Proposition 3.5.1 we can conclude that the entailment is provable, that is, there is a proof for the sequent

   $$\vdash \varphi_1 \to (\varphi_2 \to (\varphi_3 \to (\ldots \to (\varphi_n \to \psi) \ldots)))$$

3. A proof for $\varphi_1, \ldots, \varphi_n \models \psi$ is obtained as follows. First, we take the proof of $\vdash \varphi_1 \to (\varphi_2 \to (\varphi_3 \to (\ldots \to (\varphi_n \to \psi)\ldots)))$ and augment it by introducing $\varphi_1, \ldots, \varphi_n$ as premises. Then, the modus ponens rule $\to e$ is used n times to derive the conclusion ψ. We thus have a proof of $\varphi_1, \ldots, \varphi_n \vdash \psi$. ■

Soundness and completeness: The big picture. We have seen the semantic account of propositional logic in Section 3.2 and the syntactic account in Section 3.3. The notion of soundness links the two in one direction: Everything that is derivable also holds in the semantics. The notion of completeness links the two in the other direction: Everything that holds in the semantics also is derivable. We conclude this section with some more general thoughts on this bidirectional connection between syntax and semantics. Soundness and completeness are ...

1. **... signs of quality:** Informally, both soundness and completeness say something about the *quality* of the proof rules of a theory. If we have soundness then the rules don't allow the derivation of a formula that's wrong in the model. Compare the derivation of a formula with looking up a number in a telephone book. A sound telephone book contains only correct entries. Completeness, on the other hand, says that in some sense there are "enough" rules in the theory such that everything that holds in the model can be derived. A complete telephone book contains an entry for every household with a phone.

2. **... more general than you think:** As the above analogy illustrates, the notions of soundness and completeness not only make sense in the context of logic and proof systems. For instance, they also apply to analysis techniques that attempt to extract properties from a specification, an object model, or a program and will thus be used frequently throughout this course. A sound analysis technique only yields true results. A complete analysis technique is capable of determining every true property.

3. **... a big help:** Let Γ denote a sequence of premises $\varphi_1, \ldots, \varphi_n$. To show that ψ is derivable from Γ we find a proof of $\Gamma \vdash \psi$. However, suppose we have difficulty finding such a proof. We can be in one of three cases:

   - ψ is a not a consequence of Γ, and thus no sound proof theory will allow us to write down a proof of $\Gamma \vdash \psi$.
   - ψ is a consequence of Γ, but our proof theory does not contain the rules required to write down the proof of $\Gamma \vdash \psi$.
   - ψ is a consequence of Γ and our logic contains the required rules, but we have simply not yet found a proof.

   In the first two cases, trying to find a proof is a waste of time. However, to prove that there is no proof the theory offers us no other choice but to somehow systematically consider every possible candidate proof and argue

that it is not a proof of $\Gamma \vdash \psi$. In other words, showing that a sequent is provable is much easier than showing that it is not. Thus, in a sense, proof theory thus gives a 'optimistic' account of the logic. Semantics, on the other hand, works in the opposite way. Showing that that ψ is not a consequence of Γ is easy, because we just need to find a 'counter example', that is, a truth assignment that makes all formulas in Γ true but not ψ. To show that ψ is a consequence of Γ, however, is much harder, because we need to argue that every truth assignment that makes the formulas in Γ true also makes ψ true. Since the number of possible assignments to consider grows exponentially in the number of atomic propositions, this can be an impossible task. Semantics thus gives us a 'pessimistic' account of the logic. We draw two conclusions from these observations:

- It is important to study both the proof theory and the semantics.
- Soundness and completeness form a connection between syntax and semantics that allows us to have the best of both worlds. That is, whether or not ψ is a consequence of Γ and whether or not we know that, we can be assured that "there is an easy way out there":
  - Case: ψ is a consequence of Γ. Then, by completeness, there exists a proof of $\Gamma \vdash \psi$. Once we have a proof of $\Gamma \vdash \psi$, soundness allows us to conclude $\Gamma \models \psi$.
  - Case: ψ is not a consequence of Γ. Then, there exists a counter example. Once we have found a counter example, we can conclude $\Gamma \not\models \psi$. Soundness then allows us to conclude that there is no proof of $\Gamma \vdash \psi$, that is, we have $\Gamma \nvdash \psi$.

  Of course, given arbitrary Γ and ψ we may not know which one of the two cases we are in. Experience and gut feeling helps here. And, of course, it's comforting to know that an easy way to the truth is out there.

4. **... not always necessary:** In the best case, a proof theory (or a telephone book or an analysis technique) is sound and complete. Then, the syntactic notion of proof/derivation ($\vdash$) and the semantic notion of evaluation/truth tables ($\models$) are freely interchangeable. However, completeness can be hard or even impossible to come by. As long as we can live with the fact that certain valid formulas are not derivable, a sound but incomplete theory could still be useful. In contrast, logicians have always deemed unsound theories useless. Their argument is that one cannot rely on the results of an unsound theory. With an unsound telephone book you cannot be absolutely certain that you won't get the wrong person out of bed. However, a more pragmatic view holds that unsound theories can be just as useful as incomplete ones. More precisely, the usefulness of an unsound theory depends on your use and the "degree of unsoundness". For instance, if Kingston's telephone book contains only one wrong entry and that entry happens to be that of a funeral home, the book would

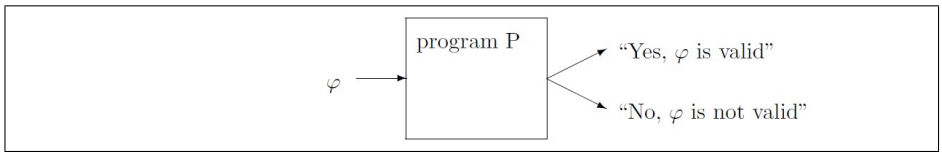


Figure 3.1: Deciding the validity problem

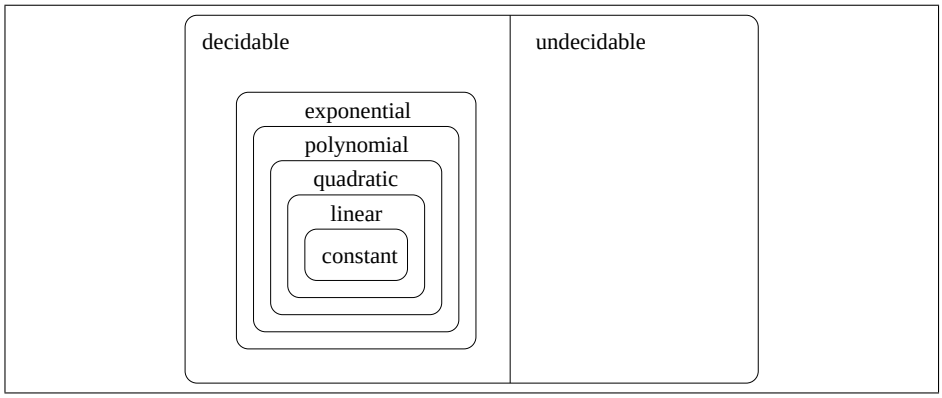


Figure 3.2: Relationships between most common complexity classes

still be very useful for the average college student. Lately, researchers in automatic program analysis have proposed unsound, but yet useful analysis techniques. Towards the end of the course we will hopefully see some examples of this strategy.

3.6 Decidability

The problem of deciding whether a formula in a given logic is valid is known as the *validity problem*. A solution is a mechanical procedure, that is, a program P that takes a formula as input, always terminates, and outputs "yes" if the input formula is valid and "no" otherwise (Figure 3.1). If such a program exists, the logic is said to be *decidable*. Decidability of a problem is an indication of its complexity. Undecidable problems are more difficult than decidable ones, just like problems that can only be solved in quadratic time are more difficult than problems solvable in linear or constant time. The relationship between decidability and the measures of algorithmic complexity such as "constant", "linear", etc is given in Figure 3.2.

We see that undecidable problems top the scale of difficulty. Undecidable problems are impossible to solve.

Theorem 3. (Decidability of propositional logic)
The validity problem for propositional logic is decidable.

Proof: We present a mechanical procedure that always terminates with the correct output. Suppose the input is a formula φ. The procedure consists of two steps:

1. Determine the distinct atomic propositions in φ. Since formulas are finite, this step always terminates.

2. Suppose there are n distinct atomic propositions in φ. Evaluate φ under all possible 2^n truth assignments, that is, compute $eval_l(\varphi)$ for all truth assignments l. As soon as we find a truth assignment l' such that $eval_{l'}(\varphi) = \mathit{false}$, we terminate and output "no". If none of the 2^n possible truth assignments makes φ false, we terminate and output "yes". Since φ is finite, the computation of $eval_l(\varphi)$ always terminates. Thus, this step also always terminates. ■

Note that a solution to the decidability problem does not have to be efficient, it just needs to terminate always. The procedure described in the above proof is $O(2^n)$ and thus not efficient ("efficient" usually means better than cubic). In applications, formulas φ with at least 100 atomic propositions are not uncommon. Even if we assume, somewhat optimistically, that we can check 10^9 truth assignments per second, the algorithm would take 10^{14} years and thus longer than the age of the universe to determine whether φ is valid. It is unknown whether the validity problem for propositional logic can be solved in polynomial time. In fact, the validity problem is NP-complete which means that the existence of a polynomial decision procedure for validity for propositional logic implies P=NP. We conclude that the validity problem for propositional logic is easy enough to be decidable, but most likely too hard to be solvable efficiently.

Remember that a formula φ is valid if and only if its negation $\neg\, \varphi$ is not satisfiable. In other words, the satisfiability problem is reducible to the validity problem. This has two consequences:

- The satisfiability problem for propositional logic also is decidable, because the above decision procedure can also be used to determine satisfiability.

- The satisfiability problem for propositional logic also is NP-complete.

Exercise 3.6.1. Devise a procedure that decides the satisfiability problem for propositional logic formulas.

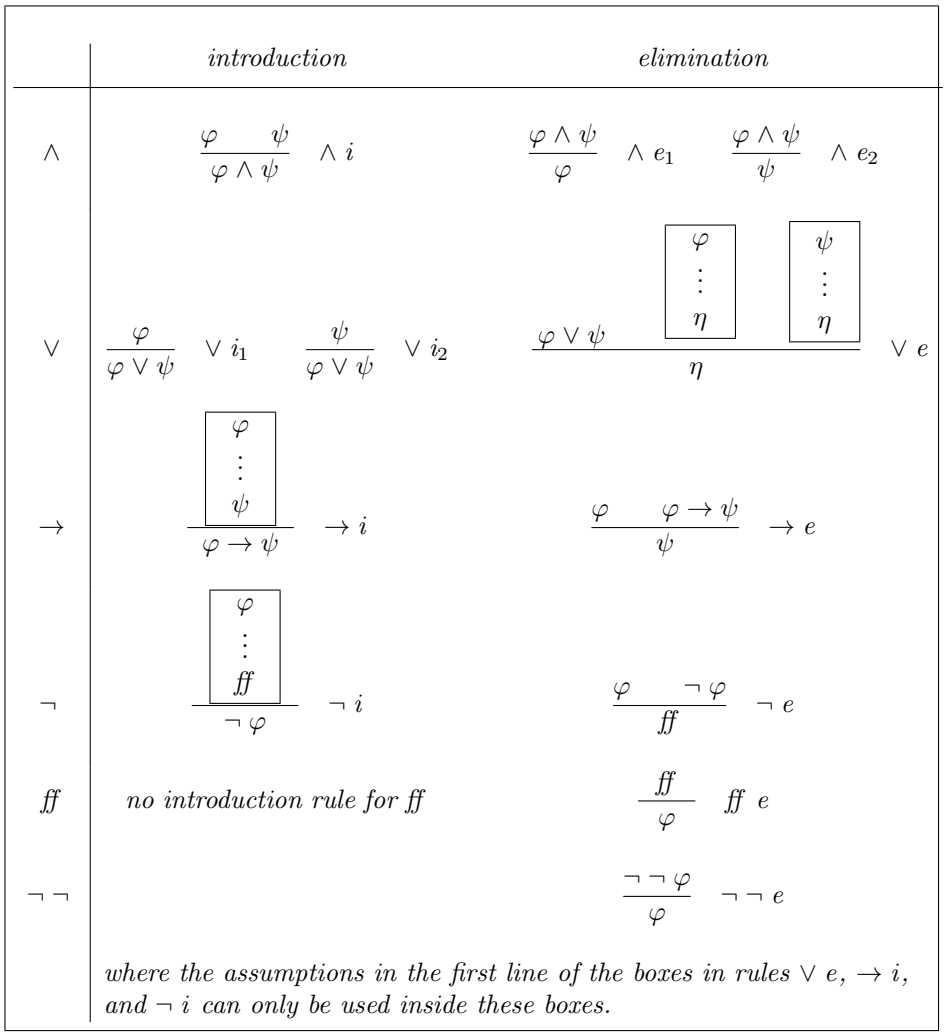


Figure 3.3: Natural deduction rules for propositional logic

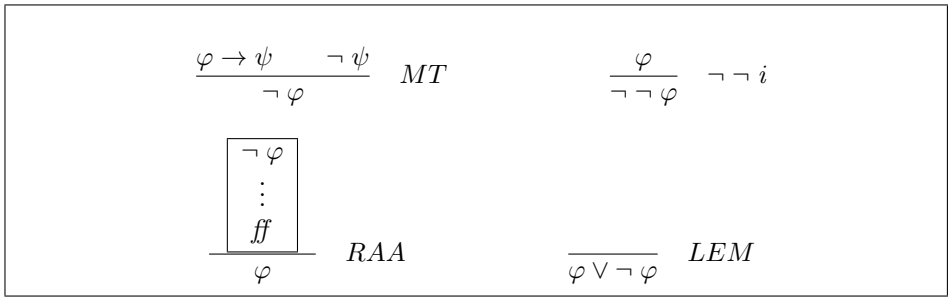


Figure 3.4: Some derived rules for propositional logic

Chapter 4

Predicate logic with equality (Week 3)

Before we plunge into the details of predicate logic[1], we will investigate in what sense propositional logic is insufficient by means of an example. Can the statement "*All months have 31 days*" be expressed in propositional logic? Our first attempt is to identify the statement simply with a single atomic proposition p. However, this does not adequately express the statement. To see this, suppose that another atomic proposition q expresses "*April has 30 days*". The intuitively obvious relationship between the two statements is completely hidden. We know that the two statements cannot both be true. However, we cannot derive this, because $p \wedge q$ is satisfiable. The idea behind our second attempt is the insight that "*All months have 31 days*" really is an abbreviation for the twelve-fold conjunction "*January has 31 days and February has 31 days and ... and December has 31 days*". A more detailed model therefore would be

$$p_{January} \wedge p_{February} \wedge \ldots \wedge p_{December}$$

where p_m abbreviates the atomic proposition "*m has 31 days*". This model is more accurate because we are now able to derive that the two statements "*All months have 31 days*" and "*April has 30 days*" are contradictory. What makes this example work is the fact that there are only finitely many months. A formula with infinitely many conjunctions would be necessary to model, for instance, "*All prime numbers are odd*" using the same technique. However, formulas of infinite length are prohibited, since they cannot be derived with the grammar on page 13. At this point we have to admit defeat and concede that statements like "*All prime numbers are odd*", although declarative, cannot be expressed in propositional logic. Predicate logic overcomes this deficiency by allowing us to syntactically separate the objects that a statement talks about from the properties that the statement claims the objects have. In other words,

[1] Predicate logic is sometimes also called *first order logic* or *first order predicate logic*

predicate logic contains *terms* to denote objects and *predicates* to express properties of these objects. For instance, "*April has 30 days*" asserts that the month April has the property of consisting of 30 days. Predicate logic allows us to write $P(apr)$ where apr is a constant denoting the month April and P is a predicate that is true of a month if and only if that month has 30 days. "*All prime numbers are odd*" can be expressed as $\forall x.Prime(x) \to Odd(x)$ where $\forall\, x$ denotes a universal quantification over the variable x, *Prime* represents the predicate that holds for a number if and only if that number is prime, and *Odd* represents the predicate that holds for a number if and only if that number is odd.

4.1 Syntax

4.1.1 Terms

We have already used constants and variables to denote a single object. The difference is that a constant always denotes a specific object. In contrast, a variable denotes an unspecified object. We will find it convenient to apply a function to one or more objects to obtain some other object. For instance, it may be useful to have a function that when given a month returns the month that succeeds it. To this end, the syntax contains function symbols. Using constants, variables, and function symbols we can now build expressions that denote objects. These expressions are called *terms*. Note that 0-ary predicate symbols are atomic propositions as we used them in Chapter 3 for propositional logic. Similarly, constants can be viewed as 0-ary functions. We adopt this convention because it simplifies the formal definition of terms. Let $\mathcal{V}$ be a set of variables and let $\mathcal{F}$ be a set of function symbols. The BNF for terms over $(\mathcal{V}, \mathcal{F})$ is

$$t \quad ::= \quad x \mid f(t_1, t_2 \ldots, t_n)$$

where $x \in \mathcal{V}$ and f is a function symbol in $\mathcal{F}$ with arity n with $n \geq 0$. Note that function symbols with $n = 0$ denote constant values such as "3" or "Sunday". Again, note how this definition is inductive. As we will see, this will allow us to, e.g., define functions on terms inductively.

4.1.2 Formulas

Just like for propositional logic, we define the syntax of formulas only in terms of negation and conjunctions. The remaining connectives again arise as abbreviations. Let $\mathcal{P}$ be sets of predicate symbols. The BNF for formulas over $(\mathcal{V}, \mathcal{F}, \mathcal{P})$ is

$$\varphi \quad ::= \quad P(t_1, \ldots, t_n) \mid (t_1 = t_2) \mid (\neg\, \varphi) \mid (\varphi \wedge \varphi) \mid (\forall x.\varphi) \mid (\varphi)$$

where P is a predicate symbol in $\mathcal{P}$ with arity n, all t_i are terms over $(\mathcal{V}, \mathcal{F})$, and x is a variable in $\mathcal{V}$. Since we are studying predicate logic with equality, we can always compare two terms for equality.

The binding priorities are as before except that quantification binds like negation.

$\neg, \forall$	negation and quantification bind most tightly
$\wedge$	conjunction binds least tightly

Example Let

$$\begin{aligned} \mathcal{V} &= \{x, y, z\} \\ \mathcal{F} &= \{zero, succ, add\} \end{aligned}$$

where *zero*, *succ*, and *add* have arity 0, 1, and 2 respectively. Also, let

$$\mathcal{P} = \{\leq, isZero\}$$

where $\leq$ takes two arguments and *isZero* takes one argument. Examples for well-formed formulas over $(\mathcal{V}, \mathcal{F}, \mathcal{P})$ include:

$$\begin{gathered} isZero(succ(zero)) \\ zero = x \\ \forall\, y.succ(zero) = zero \\ \forall\, x.\neg\, (add(x, succ(succ(x))) \leq x) \end{gathered}$$

Exercise 4.1.1. Show how propositional logic arises as a special case from predicate logic.

4.1.3 Free and bound variables

Intuitively, the occurrence of a variable x in a formula is called *bound* if it is under the scope of a quantifier $\forall\, x$. Otherwise it is called *free*. For instance, the occurrence of x is bound in $\forall x.P(x, y)$ while the occurrence of y is free. We need to distinguish between free and bound occurrences of a variable, because they are evaluated differently. A bound occurrence of x stands for *any possible value* of x, while additional information is needed to determine which value a free occurrence of x stands for. A variable is free in a formula if it has a free occurrence in the formula. Given a formula φ, the function *fv* computes the set $fv(\varphi)$ of free variables in φ:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(f(t_1, \ldots, t_n)) &= \textstyle\bigcup_{i=1}^{n} fv(t_i) \\ fv(P(t_1, \ldots, t_n)) &= \textstyle\bigcup_{i=1}^{n} fv(t_i) \\ fv(t_1 = t_2) &= fv(t_1) \cup fv(t_2) \\ fv(\neg\, \varphi) &= fv(\varphi) \\ fv(\varphi \wedge \psi) &= fv(\varphi) \cup fv(\psi) \\ fv(\forall x.\varphi) &= fv(\varphi) \backslash \{x\} \end{aligned}$$

where $S_1 \backslash S_2$ denotes the set subtraction. Note that a variable can both occur bound and free in a formula. Consider $Q(x) \wedge \forall x.P(x)$ for instance. However, each individual occurrence of a variable in a formula are always either bound or free, never both at the same time.

4.1.4 Substitution

Since variables are place holders one important operation on them is to replace them with more concrete information. This information can be given in the form of a constant, another variable, or some function application. That is, we need to be able to replace variables by terms. Consider, for instance, the formula $y > x + 1$. We know from elementary arithmetic that if $x = 3 + z$ then $y > (3 + z) + 1$. Substitution allows us to formalize this process. As a first attempt, let $\varphi[t/x]$ denote the formula that looks just like φ except that every free occurrence of x in φ is replaced by t. For example, $(y > x + 1)[3 + z/x]$ denotes $y > 3 + z + 1$. For a second example, consider $x = z$ and $\neg\, \forall z.z = x$. Intuitively, $\neg\, \forall z.z = x$ is true because says that it is not the case that x is equal to all numbers. Substituting z for x in $\neg\, \forall z.z = x$ however, yields $\neg\, \forall z.z = z$ expressing that it is not the case that all numbers are equal to themselves which clearly is a contradiction. This example tells us that a substitution must be carried out with care. More precisely, in a substitution $\varphi[t/x]$ it must not be the case that a variable in t becomes bound in $\varphi[t/x]$. We say that t *is free for* x *in* φ if no free occurrence of x in φ is under the scope of a quantification $\forall\, y$ for any variable y in t. For instance, z is not free for x in $\neg\, \forall z.z = x$. Substitutions will be subject to this condition. Note that whenever a term t fails to be free for a variable x in some formula φ, then there always is a formula φ' such that φ and φ' are logically equivalent and t is free for x in φ'. That formula φ' is obtained from φ by renaming the violating bound variables consistently and appropriately. Consequently, the freeness condition does not really limit the applicability of substitutions. For instance, z is the bound variable that causes z not to be free for x in $\neg\, \forall z.z = x$. However, the formula $\neg\, \forall y.y = x$ is logically equivalent to $\neg\, \forall z.z = x$ and now z is free for x in $\neg\, \forall y.y = x$. The result of the substitution then is $\neg\, \forall y.y = z$. In summary, a substitution $\varphi[t/x]$ proceeds in two steps:

1. If t is not free for x in φ, then determine the bound variables in φ that capture the variables in t, rename them appropriately and consistently, call the result φ, and

2. Return the formula that is like φ except that every free occurrence of x in φ is replaced by t. Formally, this step is defined by induction over the structure of φ.

Examples We have

$$\begin{aligned}
\big(x = y + 1\big)[3/y] &= x = (3 + 1)\\
\big(x = y + 1 \wedge \forall\, y.y > 10\big)[3/y] &= x = (3 + 1) \wedge \forall\, y.y > 10\\
\big(x = y + 1\big)[3 + x/y] &= x = (3 + x) + 1\\
\big(\forall x.x = y + 1\big)[3 + z/y] &= \forall x.x = (3 + z) + 1\\
\big(\forall x.x = y + 1\big)[3 + x/y] &= \forall z.z = (3 + x) + 1.
\end{aligned}$$

As we will see, substitution is crucial for the formulation of sound proof rules for quantification.

4.2 Semantics

The syntax of predicate logic allows us to build formulas such as $\forall\, x.\, \leq (x, succ(x))$ or $\forall\, x.\, \forall\, y.\, \leq (x, y)$. But what exactly do these formulas mean? Under precisely what circumstances is one formula true and another false? When are two formulas equivalent? How do we evaluate them? The formulation of a formal semantics for predicate logic will allow us to answer these questions.

The evaluation of propositional formulas over atomic propositions AP requires a truth assignment l assigning truth values to each of the atomic propositions in AP. The evaluation of predicate formulas requires a bit more machinery. Given a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$, the triple $(\mathcal{D}^{\mathcal{M}}, \mathcal{F}^{\mathcal{M}}, \mathcal{P}^{\mathcal{M}})$ is a $(\mathcal{F}, \mathcal{P})$-model, if

- $\mathcal{D}^{\mathcal{M}}$ is a non-empty set of concrete values, sometimes called the *domain* or *universe* of the interpretation. If, for instance, we want to interpret a formula in the natural numbers or the set of strings over $\{0, 1\}$, we would choose $\mathcal{D} = \mathbb{N}$ or $\mathcal{D} = \{0, 1\}^*$ respectively.

- $\mathcal{F}^{\mathcal{M}}$ is a set of functions such that for each n-ary function symbol f in $\mathcal{F}$, there is a function $f^{\mathcal{M}} : (\mathcal{D}^{\mathcal{M}})^n \to \mathcal{D}^{\mathcal{M}}$ in $\mathcal{F}^{\mathcal{M}}$. Formally,

$$\mathcal{F}^{\mathcal{M}} \quad = \quad \{f^{\mathcal{M}} : (\mathcal{D}^{\mathcal{M}})^n \to \mathcal{D}^{\mathcal{M}} \mid f \in \mathcal{F} \wedge f \textit{ has arity } n\}$$

  where $(\mathcal{D}^{\mathcal{M}})^n$ denotes the n-fold cartesian product of $\mathcal{D}^{\mathcal{M}}$, that is,

$$\underbrace{\mathcal{D}^{\mathcal{M}} \times \ldots \times \mathcal{D}^{\mathcal{M}}}_{n \text{ times}}$$

- $\mathcal{P}^{\mathcal{M}}$ is a set of predicates such that for each n-ary predicate symbol P in $\mathcal{P}$, there is a predicate $P^{\mathcal{M}} : (\mathcal{D}^{\mathcal{M}})^n \to \mathbb{B}$ in $\mathcal{P}^{\mathcal{M}}$. Formally,

$$\mathcal{P}^{\mathcal{M}} \quad = \quad \{P^{\mathcal{M}} : (\mathcal{D}^{\mathcal{M}})^n \to \mathbb{B} \mid P \in \mathcal{P} \wedge P \textit{ has arity } n\}$$

We need one more piece before we can formally define how a predicate logic formula is evaluated. For instance, to evaluate $\forall x.\varphi$, we need to determine what φ evaluates to for all concrete values in our model. More precisely, for all values $d \in \mathcal{D}^{\mathcal{M}}$, we evaluate φ assuming that x is a place holder for d. We currently have no way of formalizing these evaluations, because they use a syntactic entity φ and a semantic entity d simultaneously. For instance, performing substitution and evaluating $\varphi[d/x]$ for each d does not work, because $\varphi[d/x]$ is not a legal formula. We are forced to store the corresponding value for x in a mapping $l : \mathcal{V} \to \mathcal{D}^{\mathcal{M}}$ called an *environment* and to perform the evaluation relative to this environment.

Now, finally, we're ready to rumble. Given a formula φ over $(\mathcal{V}, \mathcal{F}, \mathcal{P})$, a $(\mathcal{F}, \mathcal{P})$-model $\mathcal{M}$ with environment l, we define an evaluation function

$$eval_{\mathcal{M},l} : t \to \mathcal{D}^{\mathcal{M}}$$

that determines which concrete value a term stands for. Since terms have an inductive structure, our definition of $eval_{\mathcal{M},l}$ will also be inductive.

$$\begin{array}{rcl} eval_{\mathcal{M},l}(x) & = & l(x) \\ eval_{\mathcal{M},l}(f(t_1, \ldots, t_n)) & = & f^{\mathcal{M}}(eval_{\mathcal{M},l}(t_1), \ldots, eval_{\mathcal{M},l}(t_n)) \end{array}$$

The satisfaction relation $\mathcal{M}, l \models \varphi$ holds if φ evaluates to *true* in the model $\mathcal{M}$ relative to the environment l. Just like for propositional logic, we define this relation by induction over the structure of φ.

$\mathcal{M}, l \models P(t_1, \ldots, t_n)$	iff $P^{\mathcal{M}}(eval_{\mathcal{M},l}(t_1), \ldots, eval_{\mathcal{M},l}(t_n)) = true$
$\mathcal{M}, l \models t_1 = t_2$	iff $eval_{\mathcal{M},l}(t_1) = eval_{\mathcal{M},l}(t_2)$
$\mathcal{M}, l \models \neg\, \varphi$	iff it is not the case that $\mathcal{M}, l \models \varphi$ holds
$\mathcal{M}, l \models \varphi \wedge \psi$	iff $\mathcal{M}, l \models \varphi$ and $\mathcal{M}, l \models \psi$ holds
$\mathcal{M}, l \models \forall x.\varphi$	iff $\mathcal{M}, l[x \mapsto d] \models \varphi$ holds for all d in the domain $\mathcal{D}^{\mathcal{M}}$

where $l[x \mapsto d]$ is the environment that is just like l except that x is associated with value d. We typically write $\mathcal{M}, l \not\models \varphi$ if $\mathcal{M}, l \models \varphi$ does not hold.

Given φ, let l and l' be two environments that agree on all free variables in φ. Then, $\mathcal{M}, l \models \varphi$ holds if and only if $\mathcal{M}, l' \models \varphi$ does. The proof proceeds by induction over the height of the parse tree of φ. Thus, when φ is closed, that is, has no free variables, $\mathcal{M}, l \models \varphi$ holds, or does not hold, for all environments l. Intuitively, since l determines the values of free variables, the truth of a closed formulas φ in $\mathcal{M}$ is independent of l. Thus, if φ is closed, we write $\mathcal{M} \models \varphi$.

4.2.1 Examples

Let

$$\mathcal{F} \stackrel{def}{=} \{zero, succ, add\}$$

where *zero*, *succ*, and *add* have arity 0, 1, and 2 respectively. Also, let

$$\mathcal{P} \stackrel{def}{=} \{\leq, isZero\}$$

where $\leq$ takes two arguments and *isZero* takes one argument.

4.2.2 Using the semantics to evaluate a formula in a model

We present three different $(\mathcal{F}, \mathcal{P})$-models called $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$.

1. Let the model $\mathcal{M}_1$ be given by

   - $\mathcal{D}^{\mathcal{M}_1} \stackrel{def}{=} \mathbb{N}$,

- $\mathcal{F}^{\mathcal{M}_1} \stackrel{def}{=} \{zero^{\mathcal{M}_1}, succ^{\mathcal{M}_1}, add^{\mathcal{M}_1}\}$ where $zero^{\mathcal{M}_1} = 0 \in \mathbb{N}$, $succ^{\mathcal{M}_1}$ is the successor function on $\mathbb{N}$ and $add^{\mathcal{M}_1}$ is the addition function on $\mathbb{N}$, and
- $\mathcal{P}^{\mathcal{M}_1} \stackrel{def}{=} \{\leq^{\mathcal{M}_1}, isZero^{\mathcal{M}_1}\}$ where $\leq^{\mathcal{M}_1}$ is "less than or equal" on $\mathbb{N}$, and $isZero^{\mathcal{M}_1}(x) = true$ if and only if $x = 0$.

2. Let the model $\mathcal{M}_2$ be given by

- $\mathcal{D}^{\mathcal{M}_2} \stackrel{def}{=} \{0,1\}^*$,
- $\mathcal{F}^{\mathcal{M}_2} \stackrel{def}{=} \{zero^{\mathcal{M}_2}, succ^{\mathcal{M}_2}, add^{\mathcal{M}_2}\}$ where $zero^{\mathcal{M}_2}$ is the empty word ϵ, $succ^{\mathcal{M}_2}(w) = w1$ for all $w \in \{0,1\}^*$, and $add^{\mathcal{M}_1}(w_1, w_2) = w_1 w_2$ for all $w_1, w_2 \in \{0,1\}^*$, and
- $\mathcal{P}^{\mathcal{M}_2} \stackrel{def}{=} \{\leq^{\mathcal{M}_2}, isZero^{\mathcal{M}_2}\}$ where $\leq^{\mathcal{M}_2}$ is the prefix relation on $\{0,1\}^*$, that is, $w_1 \leq^{\mathcal{M}_2} w_2$ if and only if w_1 is a prefix of w_2, and $isZero^{\mathcal{M}_2}$ returns *true* if and only if the argument is ϵ.

3. Let the model $\mathcal{M}_3$ be given by

- $\mathcal{D}^{\mathcal{M}_3} \stackrel{def}{=} \{off, on\}$,
- $\mathcal{F}^{\mathcal{M}_3} \stackrel{def}{=} \{zero^{\mathcal{M}_3}, succ^{\mathcal{M}_3}, add^{\mathcal{M}_3}\}$ where $zero^{\mathcal{M}_3} = off$ and

$$\begin{aligned} succ^{\mathcal{M}_3}(off) &= on \\ succ^{\mathcal{M}_3}(on) &= off, \end{aligned}$$

  and $add^{\mathcal{M}_3}$ is the function defined by

$$\begin{aligned} add^{\mathcal{M}_3}(off, off) &= off \\ add^{\mathcal{M}_3}(off, on) &= add^{\mathcal{M}_3}(on, off) = on \\ add^{\mathcal{M}_3}(on, on) &= off, \end{aligned}$$

  and
- $\mathcal{P}^{\mathcal{M}_3} \stackrel{def}{=} \{\leq^{\mathcal{M}_3}, isZero^{\mathcal{M}_3}\}$ where

$$\leq^{\mathcal{M}_3}(x, y) = \begin{cases} false, & \text{if } x = on \text{ and } y = off \\ true, & \text{otherwise.} \end{cases}$$

  and $isZero^{\mathcal{M}_3}(x) = true$ if and only if $x = off$.

As different as these models seem on first glance, they have a lot in common. For instance, the formulas

$$\begin{gathered} isZero(zero) \\ \neg\, isZero(succ(zero)) \\ \forall\, x.add(x, zero) = x \\ \forall\, x. \forall\, y.add(x, y) = x \rightarrow isZero(y) \\ \forall\, x. \forall\, y.\big(x \leq add(x, y) \wedge \neg\, (x = add(x, y))\big) \rightarrow \neg\, (isZero(y)) \end{gathered}$$

hold in all three models. However, there also are some differences. The following formulas hold in models $\mathcal{M}_1$ and $\mathcal{M}_2$ but not in $\mathcal{M}_3$:

$$\neg\, \mathit{isZero}(\mathit{succ}(\mathit{succ}(\mathit{zero}))$$
$$\forall\, x.x \leq \mathit{succ}(x)$$

While the formula

$$\neg\, (\forall\, x.\neg\, (\mathit{isZero}(\mathit{succ}(x))))$$

holds in $\mathcal{M}_3$ but not in $\mathcal{M}_1$ or $\mathcal{M}_2$.

It is tempting to think that $\mathit{isZero}(\mathit{zero})$ holds in any $(\mathcal{F}, \mathcal{P})$-model. However, we can easily give a model in which $\mathit{isZero}(\mathit{zero})$ does not hold. For instance, if *isZero* is interpreted as the predicate that returns *false* on all its arguments, then that predicate will also yield *false* on the interpretation of *zero*. Remember that function and predicate symbols are just syntax that can be interpreted any way we like.

4.2.3 Satisfiability, validity, and entailment

Let φ be a closed formula over the symbols $\mathcal{F}$ and $\mathcal{P}$. φ is *satisfiable* if there exists a $(\mathcal{F}, \mathcal{P})$-model $\mathcal{M}$ such that φ holds in $\mathcal{M}$, that is, $\mathcal{M} \models \varphi$.

Formula φ is *valid* if φ holds in all $(\mathcal{F}, \mathcal{P})$-models. Like in propositional logic, a valid formula is also called tautology. Tautologies are general truths that hold independent of particular interpretations. For instance, all of the formulas mentioned in the example above are satisfiable. None of them is a tautology. Examples for tautologies are

$$\varphi_1 \vee \neg\, \varphi_1$$
$$\big(\forall\, x.\varphi_1 \wedge \varphi_2\big) \leftrightarrow \big(\forall\, x.\varphi_1\big) \wedge \big(\forall\, x.\varphi_2\big)$$
$$\big((\forall\, x.\varphi_1) \vee (\forall\, x.\varphi_2)\big) \rightarrow \big(\forall\, x.\varphi_1 \vee \varphi_2\big)$$
$$\big(\forall\, x.\forall\, x.\varphi\big) \leftrightarrow \big(\forall\, x.\varphi\big)$$

for any syntactically well-formed formulas φ_1 and φ_2.

Let $\varphi_1, \ldots, \varphi_n$ and ψ be closed formulas over the symbols $\mathcal{F}$ and $\mathcal{P}$. Entailment $\varphi_1, \ldots, \varphi_n \models \psi$ expresses that for all $(\mathcal{F}, \mathcal{P})$-models $\mathcal{M}$ and environments l, whenever $\mathcal{M}, l \models \varphi_i$ for all $1 \leq i \leq n$, then $\mathcal{M}, l \models \psi$.

Note that the symbol $\models$ is overloaded in predicate logic. It is used to denote satisfiability of closed formulas

$$\mathcal{M} \models \varphi$$

and semantic entailment

$$\varphi_1, \ldots, \varphi_n \models \psi$$

4.2.4 Existential quantification

Just as disjunction can be defined in terms of conjunction and negation, existential quantification can be defined in terms of universal quantification and

negation.

$$\exists x.\varphi \quad \stackrel{def}{=} \quad \neg\, \forall x.\neg\, \varphi$$

In the introduction to predicate logic we already mentioned that universal quantification over a domain with n objects is short for an n-fold conjunction. With the above abbreviation we see that existential quantification over the same domain is short for an n-fold disjunction. The proof rules for both quantifiers to be presented in the next section will reflect this correspondence.

To save parentheses, the quantifiers and connectives are assigned the following *binding priorities*:

$\neg\,, \forall, \exists$	negation and quantifiers bind most tightly
$\wedge, \vee$	conjunction and disjunction have medium binding power
$\rightarrow$	implication binds least tightly

Using the semantics to show that formulas are (not) equivalent

Consider the three formulas

$$\begin{array}{lcl} f_1 & \stackrel{def}{=} & \forall\, x.\, \exists\, y.\, \leq (x, y) \\ f_2 & \stackrel{def}{=} & \forall\, y.\, \exists\, x.\, \leq (y, x) \\ f_3 & \stackrel{def}{=} & \exists\, y.\, \forall\, x.\, \leq (x, y) \end{array}$$

Which one of these formulas are equivalent? How do we show this?

1. Consider f_1 and f_2. These two formulas are equivalent. Using the semantics, we show this as follows. Given some model $\mathcal{M}$ and some environment l, we have

   $$\mathcal{M}, l \models \forall\, x.\, \exists\, y.\, \leq (x, y)$$

   iff for all d_1 in $\mathcal{D}^{\mathcal{M}}$, there exists d_2 in $\mathcal{D}^{\mathcal{M}}$ such that

   $$\mathcal{M}, l[x \mapsto d_1][y \mapsto d_2] \models \leq (x, y)$$

   iff for all d_1 in $\mathcal{D}^{\mathcal{M}}$, there exists d_2 in $\mathcal{D}^{\mathcal{M}}$ such that

   $$\leq^{\mathcal{M}} (d_1, d_2)$$

   iff for all d_1 in $\mathcal{D}^{\mathcal{M}}$, there exists d_2 in $\mathcal{D}^{\mathcal{M}}$ such that

   $$\mathcal{M}, l[y \mapsto d_1][x \mapsto d_2] \models \leq (y, x)$$

   iff

   $$\mathcal{M}, l \models \forall\, y.\, \exists\, x.\, \leq (y, x)$$

   This example shows that consistently renaming bound variables in a formula does not alter its meaning.

2. Consider f_1 and f_3. These two formulas are not equivalent. Using the semantics, we show this as follows. Given model $\mathcal{M}_1$ (as defined above) and some environment l, we have

$$\mathcal{M}_1, l \models \forall\, x.\, \exists\, y.\, \leq (x, y)$$

iff for all d_1 in $\mathcal{D}^{\mathcal{M}}$, there exists d_2 in $\mathcal{D}^{\mathcal{M}}$ such that

$$\mathcal{M}_1, l[x \mapsto d_1][y \mapsto d_2] \models \leq (x, y)$$

iff for all d_1 in $\mathcal{D}^{\mathcal{M}}$, there exists d_2 in $\mathcal{D}^{\mathcal{M}}$ such that

$$\leq^{\mathcal{M}_1} (d_1, d_2)$$

To see that the last statement holds in $\mathcal{M}_1$, let $d_2 \stackrel{def}{=} d_1 + 1$ for every $d_1 \in \mathcal{D}^{\mathcal{M}}$. We conclude that f_1 holds in $\mathcal{M}_1$.

However, f_3 does not hold in $\mathcal{M}_1$. To see this, we observe

$$\mathcal{M}, l \models \exists\, y.\, \forall\, x.\, \leq (x, y)$$

for some l iff there exists d_1 in $\mathcal{D}^{\mathcal{M}}$ such that for all d_2 in $\mathcal{D}^{\mathcal{M}}$ we have

$$\leq^{\mathcal{M}_1} (d_2, d_1)$$

However, such a d_1 cannot exist. We show this by contradiction. Suppose we have a $d_1 \in \mathcal{D}^{\mathcal{M}_1}$ such that (*) for all d_2 in $\mathcal{D}^{\mathcal{M}_1}$, $\leq^{\mathcal{M}_1} (d_2, d_1)$. Then, if we choose $d_2 \stackrel{def}{=} d_1 + 1$, we get $\not\leq^{\mathcal{M}_1} (d_2, d_1)$ which contradicts (*).

Consequently, there is at least one model in which f_1 holds, but not f_3. So, the two formulas are not equivalent.

As another example, consider

$$\begin{array}{lcl} f_4 & \stackrel{def}{=} & \forall\, x.(\exists\, y.Q(x, y)) \rightarrow P(x) \\ f_5 & \stackrel{def}{=} & \forall\, x.\, \exists\, y.Q(x, y) \rightarrow P(x) \end{array}$$

These formulas differ only in the scope of the variable y. In f_4, the scope comprises $Q(x, y)$ only, whereas in f_5, y has $Q(x, y)) \rightarrow P(x)$ as its scope. Perhaps surprisingly, these two formulas are not equivalent; there is a model that distinguishes them. For instance, consider $\mathcal{M}_2$ with

- $\mathcal{D}^{\mathcal{M}_2} \stackrel{def}{=} \{a, b\}$,
- $\mathcal{P}^{\mathcal{M}_2} \stackrel{def}{=} \{P^{\mathcal{M}_2}, Q^{\mathcal{M}_2}\}$ where $P^{\mathcal{M}_2}(a) = P^{\mathcal{M}_2}(b) = \mathit{false}$, $Q^{\mathcal{M}_2}(a, a) = Q^{\mathcal{M}_2}(b, a) = \mathit{true}$ and $Q^{\mathcal{M}_2}(a, b) = Q^{\mathcal{M}_2}(b, b) = \mathit{false}$.

Then. $\mathcal{M}_2 \not\models f_4$, but $\mathcal{M}_2 \models f_5$.

4.3 Proof theory

The new syntactic constructs in predicate logic come with new proof rules. These rules augment the propositional proof rules presented in the previous chapter.

4.3.1 Universal quantification

The rules for universal quantification are

$$\frac{\boxed{\begin{array}{l} x_0 \\ \quad \vdots \\ \varphi[x_0/x] \end{array}}}{\forall x.\varphi}\ \forall\, x\ i \qquad\qquad \frac{\forall x.\varphi}{\varphi[t/x]}\ \forall\, x\ e$$

where x_0 is a fresh variable that does not occur free anywhere in the proof. The introduction rule uses a box to introduce a dummy variable x_0 that does not occur free anywhere. Consequently, we cannot make any assumptions about x_0 and it can thus be thought of as representing all terms. Thus, being able to derive $\varphi[x_0/x]$ under these circumstances means that φ is derivable for any value that x may take on. Consequently, it is sound to conclude $\forall x.\varphi$. To illustrate the rule, consider an informal proof of the statement

> *"For every natural number,*
> *there exists another natural number that is greater"* (4.1)

An informal proof of this statement could go like this:

> *"Let n be a natural number.*
> *Then, $n+1$ also is natural number.*
> *Moreover, $n+1$ is greater than n".*

The proof starts with fixing an arbitrary natural number and naming it n. Then, it proceeds to show the desired conclusion. Since no assumptions were made about n, it works for every natural number and thus establishes the universally quantified statement (4.1). The rule $\forall\, x\ i$ formalizes this kind of reasoning. The only minor difference is that it uses x_0 rather n as a place holder.

The elimination rule also is very intuitive. If we know that φ holds no matter what value x takes on, then it will clearly also hold if x takes on the value denoted by some term t. If we think of the universal quantification as a conjunction, the similarity between the above rules and the rules for conjunction becomes apparent.

4.3.2 Existential quantification

The rules for existential quantification are

$$\frac{\varphi[t/x]}{\exists x.\varphi}\ \exists\, x\ i \qquad\qquad \frac{\exists x.\varphi \qquad \boxed{\begin{array}{ll} x_0 & \varphi[x_0/x] \\ & \vdots \\ & \eta \end{array}}}{\eta}\ \exists\, x\ e$$

where x_0 is a fresh variable that does not occur free anywhere in the proof. Intuitively, if φ holds when x takes on the value denoted by t, then $\exists\, x.\varphi$ also holds. The rule $\exists\, x\ e$ performs a case analysis similar to rule $\vee\ e$. Due to the premise $\exists\, x.\varphi$ we know that φ is true for at least one value of x. Since we don't know which value that is, the second premise performs a case analysis over all possible values, writing x_0 as a generic value that represents them all. If assuming $\varphi[x_0/x]$ allows us to prove some η that does not mention x_0, then this η must be true whichever x_0 it was. Note that if x_0 is mentioned in a formula somewhere outside the box, it loses its ability to represent all possible values explaining the side condition on x_0. Thus, the box in $\exists\, x\ e$ controls the scope not only of the assumption $\varphi[x_0/x]$ but also of the variable x_0.

4.3.3 Examples

The following examples illustrate the use of the above rules. The first three are taken from [HR00].

1. The sequent

$$\forall\, x.\big(P(x) \to Q(x)\big), \forall x.P(x) \vdash \forall x.Q(x)$$

   can be proved by

	1	$\forall x.\big(P(x) \to Q(x)\big)$	premise
	2	$\forall x.P(x)$	premise
x_o	3	$P(x_0) \to Q(x_0)$	$\forall\, x\ e(1)$
	4	$P(x_0)$	$\forall\, x\ e(2)$
	5	$Q(x_0)$	$\to e(3,4)$
	6	$\forall x.Q(x)$	$\forall\, x\ i(3,5)$

2. A minister reasons:

> "*If all quakers are reformists and if there is a protestant who is also a quaker, then there must be a protestant who is also a reformist.*"

Formally,

$$\forall x.\big(Q(x) \to R(x)\big), \exists x.\big(P(x) \wedge Q(x)\big) \vdash \exists x.\big(P(x) \wedge R(x)\big).$$

Is he right? Yes, he is:

	1	$\forall x.\big(Q(x) \to R(x)\big)$	*premise*
	2	$\exists x.\big(P(x) \wedge Q(x)\big)$	*premise*
x_o	3	$P(x_0) \wedge Q(x_0)$	*assumption*
	4	$Q(x_0) \to R(x_0)$	$\forall\, x\ e(1)$
	5	$Q(x_0)$	$\wedge\ e_2(3)$
	6	$R(x_0)$	$\to e(4,5)$
	7	$P(x_0)$	$\wedge\ e_1(3)$
	8	$P(x_0) \wedge R(x_0)$	$\wedge\ i(7,6)$
	9	$\exists x.P(x) \wedge R(x)$	$\exists\, x\ i(8)$
	10	$\exists x.P(x) \wedge R(x)$	$\exists\, x\ e(2,3,9)$

3. The sequent

$$\exists x.P(x), \forall x.\forall y.\big(P(x) \to Q(y)\big) \vdash \forall y.Q(y)$$

is proved by

	1	$\exists x.P(x)$	*premise*
	2	$\forall x.\forall y.\big(P(x) \to Q(y)\big)$	*premise*
y_o	3		
x_o	4	$P(x_0)$	*assumption*
	5	$\forall y.\big(P(x_0) \to Q(y)\big)$	$\forall\, x\ e(2)$
	6	$P(x_0) \to Q(y_0)$	$\forall\, y\ e(5)$
	7	$Q(y_0)$	$\to e(6,4)$
	8	$Q(y_0)$	$\exists\, y\ e(1,7)$
	9	$\forall y.Q(y)$	$\forall\, y\ i(3,8)$

4. The last example will illustrate the importance of the side condition on rule $\exists\, x\ e$. Consider the sequent $\exists x.P(x) \vdash \forall x.P(x)$. This sequent is, obviously, not valid, that is, there must not be a proof for it. However, if we allow the variable x_0 introduced by rule $\exists\, x\ e$ to be used in a formula

outside the box, then the sequent is provable.

	1	$\exists x.P(x)$	*premise*
x_o	2		
x_o	3	$P(x_0)$	*assumption*
	4	$P(x_0)$	*copy(3)*
	5	$P(x_0)$	$\exists\, x\ e(1,3,4)$
	6	$\forall x.P(x)$	$\forall\, x\ i(2,5)$

Without the side condition the rule $\exists\, x\ e$ becomes unsound because it allows the variable x_0 to be used in a formula outside its scope.

Exercise 4.3.1. For each of the following sequents, either prove it using Jape or show a counter example, that is, a model in which the sequent is not valid.

1. $(\forall x.P(x)) \lor (\forall\, x.Q(x)) \vdash \forall x.(P(x) \lor Q(x))$

2. $\exists x.\big(P(x) \lor Q(x)\big) \vdash \big(\exists x.P(x)\big) \lor \big(\exists x.Q(x)\big)$

3. $\big(\exists x.P(x)\big) \lor \big(\exists x.Q(x)\big) \vdash \exists x.\big(P(x) \lor Q(x)\big)$

4. $\forall x.\neg\, Q(x), P \to \big(\exists x.Q(x)\big) \vdash \neg\, P$

5. $S(m,n), \forall x.P(x) \to \neg\, S(x,n) \vdash \neg\, P(m)$

6. $\exists x.P(x) \land \neg\, Q(x), \forall x.P(x) \to R(x) \vdash \exists x.R(x) \land \neg\, Q(x).$

4.3.4 Proof techniques II

Suppose we want to prove that $\varphi \vdash \forall x.\psi$. We want to revisit the techniques presented in the previous section, and also briefly discuss induction.

- Proof by induction. We have not introduced a proof rule for induction, so this technique is mentioned here only for the sake of completeness.

  If the premise φ defines some kind of inductive set S and the conclusion ψ makes a statement over S, then we may be able to prove the sequent by induction over the elements of S. For instance, consider

$$\begin{aligned}
&\forall x.\mathit{Even}(x) \leftrightarrow \big(x = 0 \lor \exists y.(x = s(y) \land \mathit{Odd}(y))\big),\\
&\forall x.\mathit{Odd}(x) \leftrightarrow \big(x = s(0) \lor \exists y.(x = s(y) \land \mathit{Even}(y))\big),\\
&\forall x.\mathit{Nat}(x) \leftrightarrow \big(x = 0 \lor \exists y.x = s(y)\big)\\
&\quad\vdash\quad \forall x.\mathit{Nat}(x) \to \mathit{Even}(x) \lor \mathit{Odd}(x)
\end{aligned}$$

Similarly, we could prove that every formula generated using the BNF on page 13 contains an even number of parentheses. Note, however, that we have to encode the inductive structure of the elements of S in the premise, so that it is available to us in when proving the conclusion.

- Proof by case study. Now we use the case analysis to group the set of values that x ranges over into several subsets. Suppose, we additionally know

$$\varphi \to \forall x.(\varphi_1 \vee \varphi_2).$$

In other words, we partitioned the domain of x into a subset characterized by φ_1 and into a subset characterized by φ_2. This allows us to prove $\forall x.\psi$ by considering two cases.

	1	φ			*premise*
	2	$\varphi \to \forall x.(\varphi_1 \vee \varphi_2)$			*premise*
	3	$\forall x.(\varphi_1 \vee \varphi_2)$			$\to\ e(1,2)$
x_0	4	$\varphi_1[x_0/x] \vee \varphi_2[x_0/x]$			$\forall\, x\ e(3)$
	5	$\varphi_1[x_0/x]$	*assumption*	$\varphi_2[x_0/x]$	*assumption*
	6	$\vdots$	...	$\vdots$	...
	7	$\psi[x_0/x]$	...	$\psi[x_0/x]$	...
	8	$\psi[x_0/x]$			$\vee\ e(4,7)$
	9	$\forall x.\psi$			$\forall\, x\ i$

For instance, we show that all moons of planet Saturn are smaller than the Earth, by considering each of Saturn's planets and comparing its size to that of the Earth.

- Proof by contradiction. Universally quantified formulas express that some property φ holds for all values and thus can be very hard to prove. Often, it is much easier to prove instead that the assumption that there exists an instance such that $\neg\,\varphi$ leads to a contradiction. Suppose, for instance, we want to prove that for all natural numbers x there exists a number that is greater than x, that is, $\Gamma \vdash \forall x.\exists y.x < y$ where Γ formalizes the natural numbers. We could attempt to prove the sequent with induction, but it is much easier by contradiction, because the negation $\exists x.\forall y.x \geq y$ is easily seen to be wrong, because no matter what x is, $x + 1$, for instance, will always be greater.

4.3.5 Bounded quantification

A quantified variable ranges over the entire semantic domain. Sometimes, however, it is convenient to make it explicit in the formula that a quantified variable ranges over a particular set S by using *bounded quantification* $\forall\, x : S.\varphi$ or

$\exists\, x : S.\varphi$. For instance, to formalize that there is no greatest natural number we could write $\neg\ \exists\, x : \mathbb{N}.\forall\, y : \mathbb{N}.x \geq y$. In this case it is apparent that the appropriate semantic domain for the interpretation of this formula is the set of natural numbers. Note that bounded quantifications do not add any expressive power to our logic and can actually be viewed as abbreviations of standard, unbounded quantifications:

$$\begin{aligned} \forall\, x : S.\varphi &\stackrel{def}{=} \forall\, x.inS(x) \rightarrow \varphi \\ \exists\, x : S.\varphi &\stackrel{def}{=} \exists\, x.inS(x) \wedge \varphi \end{aligned}$$

where inS is a unary predicate symbol that is to be interpreted by a predicate $inS^{\mathcal{M}}$ that returns true precisely when its argument is an element of the set S. Also note how the treatment of universal and existential quantification is slightly different.

We can also introduce an abbreviation that enumerates the values that the variables can take on. For instance, $\forall\, x : [1..5].\varphi$ means that φ has to be true for all natural numbers x between 1 and 5. The translation of these formulas are:

$$\begin{aligned} \forall\, x : [l..u].\varphi &\stackrel{def}{=} \forall\, x.(l \leq x \wedge x \leq u) \rightarrow \varphi \\ \exists\, x : [l..u].\varphi &\stackrel{def}{=} \exists\, x.(l \leq x \wedge x \leq u) \wedge \varphi \end{aligned}$$

where $\leq$ is a binary predicate symbol denoting "less than or equal" and l and u are natural numbers.

Other ways of bounding the quantifiers are possible. The point is that this does not increase the expressive power of the logic as they can all be expressed in pure predicate logic.

4.4 Soundness, completeness and decidability

Our presentation of predicate logic is sound and complete. The proofs are beyond the scope of this document.

Theorem 4. (Soundness)
For every sequent $\varphi_1, \ldots, \varphi_n \vdash \psi$ that is derivable using the presented proof rules we have $\varphi_1, \ldots, \varphi_n \models \psi$.

Theorem 5. (Completeness)
For every entailment $\varphi_1, \ldots, \varphi_n \models \psi$ that holds, there exists a proof of the sequent $\varphi_1 \ldots, \varphi_n \vdash \psi$ using the presented proof rules.

In the introduction we have argued that propositional logic is not capable of expressing all formulas of predicate logic. Now is the time to pay the price for that increase in expressiveness. There is an informal tradeoff between the expressiveness of a logic and its computational complexity. In other words, the more expressive a logic is, the harder it becomes to solve the decision problem.

In contrast to propositional logic, predicate logic is not decidable. Validity is a statement over all models of a formula. Intuitively, predicate logic is undecidable, because a predicate logic formula typically has *infinitely* many models. A propositional formula with n atomic propositions has 2^n, and thus only finitely many, models (truth assignments). Recall that the proof of the decidability of propositional logic (Theorem 3) relied crucially on this fact.

Theorem 6. (Decidability)
There is no mechanical procedure, that when given a sequent as input, always terminates, and outputs "yes" if the sequent is provable, and "no" otherwise.

Note, however, it can be shown that validity for predicate logic is *semi-decidable*, that is, there is a procedure that when given a valid predicate logic formula always terminates with output "Yes". However, when the input is not valid, the procedure may run forever. The proof of this property is beyond the scope of these notes. Informally, it relies on the fact that the set of all possible models of a predicate logic formula, although infinite, can be enumerated systematically.

The first person to show that predicate logic is sound, complete, and undecidable was the Austrian logician Kurt Gödel. Gödel is most famous for his Incompleteness Theorem (published in 1931) which says that in any logic strong enough to encode the basic arithmetic (that is, natural numbers with addition, subtraction, multiplication, and division), there are always formulas that cannot be proved or disproved with the rules of the logic. The Incompleteness Theorem ended a hundred years of attempts to establish logics in which the whole of mathematics could be dealt with.

Chapter 5

Theorem provers, SAT solvers, and proof checkers

The primary purpose of a theorem prover is to formally derive a new formula f from an existing set of formulas $\{f_1, \ldots, f_n\}$ using the rules of the logic. In other words, the prover checks if

$$f_1, \ldots, f_n \vdash f$$

is valid. Theorem provers usually work with predicate logic.

The overall result of the previous two chapters is that while propositional logic can be analyzed fully automatically, predicate logic cannot. Thus, for every tool P that attempts to determine the validity or satisfiability of predicate logic formulas, there will be a formula f such that the execution of P on f will either

- produce the wrong result (loss of soundness), or
- run forever without ever producing an answer (loss of termination), or
- not be fully automatic, that is, require user interaction (loss of automation).

In the face of this negative result, theorem provers typically maintain soundness and termination, but sacrifice automation. While the prover may be able to prove a small class of simple formulas automatically, it will require user guidance on more complex formulas. Despite the absence of full automation, theorem provers can be very useful. They relieve the user from the bookkeeping necessary to ensure soundness of the constructed proof. Moreover, the proof search mechanism of the prover may be able to construct large parts of the proof completely automatically, leaving only a few steps for the user to fill in. On the other hand, whenever the prover asks for help from the user, a lot of expertise and knowledge is required. Although there have been many successful uses of theorem proving reported in the literature (e.g., [BBFM99]), the use of theorem proving techniques is confined to a small group of experts.

A *SAT solver* checks the satisfiability of a formula. Given a formula, the solver will search for a model in which the formula holds. If such a model can be found, the solver will output it. If not, the solver will output an appropriate message. The field of SAT solving is very mature and thus many sophisticated and highly optimized SAT solvers exist. Signs of this maturity are the annual SAT competitions that aim to determine the best performing tools [SAT20], and their use in industry to, e.g., generate tests for software and hardware.

A *proof checker* (or *proof assistant*) is like a theorem prover without proof search. Its sole purpose is soley to assist the user in finding the proof by, e.g., displaying the status of the proof and applying user-selected proof rules. While this avoids, e.g., the erroneous application of a rule, the user has to find the entire proof manually.

The most popular theorem provers include PVS [SRI], Isabelle [Cam], HOL [GM93], ACL2 [KMM00b, KMM00a] and Never [ORA], the prover that the Z/Eves system is built on. Examples for SAT solvers are SPASS, CHAFF [MMZ01], and SATO [Zha97]. In class, we will see how, e.g., the SPASS theorem prover can be used to solve Sudoku puzzles.

An example for a proof checker is Jape [Oxf]. We will use Jape in class to see how a proof checker works.

Chapter 6

Bibliographic notes

The truth value semantics of propositional logic presented here and in every logic book today was invented only about 160 years ago, by G. Boole [Boo54]. The first account of predicate logic reasonably close to the ones used today was given by G. Frege in 1879 [Fre03]. Natural deduction was invented by G. Gentzen [Gen69], and further developed by D. Prawitz [Pra65].

The book by Huth and Ryan [HR00] gives a more detailed treatment of the material presented here and is highly recommended. It also discusses program verification, model checking and uses of modal logic.

Part III

Introduction to the Formal Specification of Software Systems (Week 4)

Chapter 7

Background

7.1 Terminology

The terms *development process* and *development methodology* refer to a particular set of guidelines or rules which describe how a development project should generally be carried out and what artifacts (e.g., documents, design models, prototypes, etc) should be produced and in what order.

A *model* is an abstract representation of a specification, a design or a system. Models are often subject to a particular point of view (e.g., the user view, the developer view, etc). It highlights aspects of the described entity important for that view and deemphasizes or ignores unnecessary detail.

The *modeling language* used to represent the models typically is also prescribed by the development process. A modeling language may be an informal notation like plain English, a formal textual notation like predicate logic, or a formal graphical notation like class diagrams, use cases, or interaction diagrams.

7.2 Development processes

What kind of information is essential for the development of good software? Which activities must be carried out and in which order? The literature is full of so-called software development processes that attempt to answer these questions by breaking software development down in various ways.

Perhaps the most famous, although now discredited, development process is the *waterfall process*. It divides software construction into a sequence of five phases called *requirements analysis*, *design*, *implementation*, *testing*, and *maintenance*. Informally, the requirements phase determines what the software is supposed to do, the design phase ends with a high-level description of the system, the implementation phase produces the code which is subjected to tests and maintenance in the final two phases. As Figure 7.1 indicates, each phase is carried out exactly once. Consequently, the waterfall process does not account for new information or insight that is gained during later phases and forces

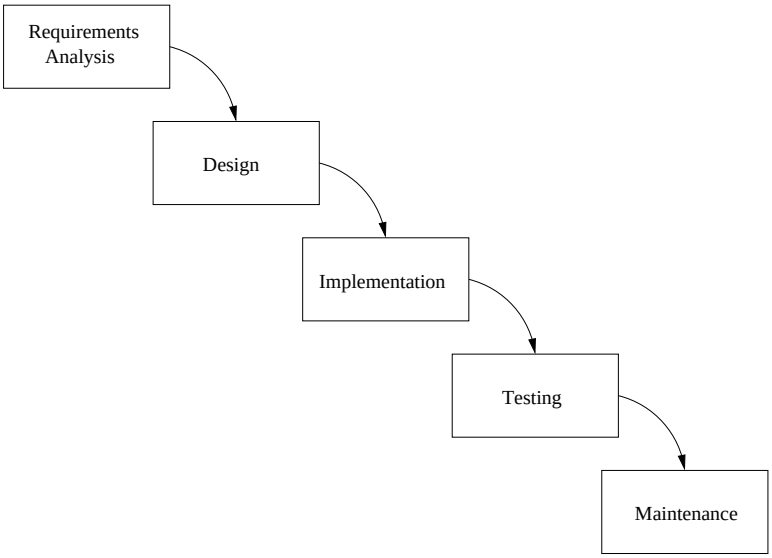


Figure 7.1: The waterfall process

earlier phases to be revisited. For instance, the testing phase may uncover serious shortcomings in the design and the code. Alternatively, the implementation phase may reveal that the requirements specifications are inconsistent, and, therefore, unimplementable. The waterfall process ignores the fact that software development often requires iteration. A number of alternative software development processes have been suggested such as:

- the *Iterative waterfall process*, in which earlier stages can be repeated,
- *Prototyping development*, in which prototypes are used for early evaluation and refinement of the requirements
- *Incremental development*, in which requirements are prioritized and assigned to increments and development proceeds in these increments.
- *Object-oriented development*, which encourages the developer to view the software to be developed as a system of cooperating, communicating, real-life objects. Figure 7.2 illustrates the object-oriented development process.
- *Extreme programming*, which is centered around user stories, small releases, early and frequent testing, and pair programming.
- *Agile development*, an evolution of extreme programming, which focusses on "requirements discovery and solutions improvement through the collaborative effort of self-organizing and cross-functional teams with their customer(s)/end user(s),[2] adaptive planning, evolutionary development, early delivery, continual improvement, and flexible responses to changes in requirements, capacity, and understanding of the problems to be solved" [Wik22a].

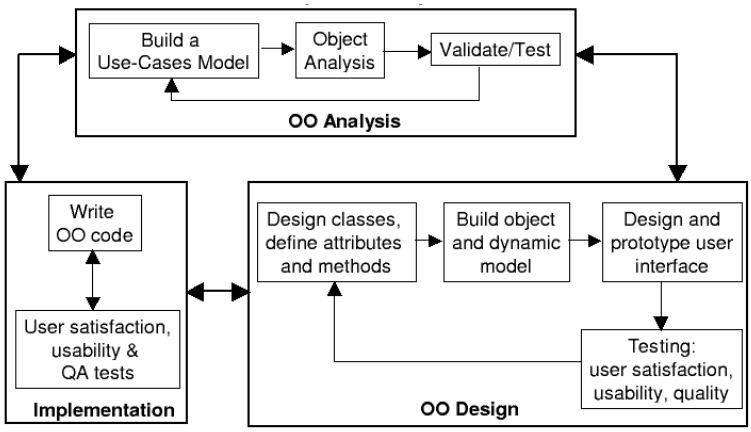


Figure 7.2: An object-oriented software development process [Bah99]

- *DevOps*, "a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development life cycle and provide continuous delivery with high software quality" [Wik22b].
- *Cleanroom engineering*, which combines the incremental process with an emphasis on formal specifications, testing, and statistical quality-control methods.
- *Model-driven development* (MDD), in which formal models of the software and its requirements form the primary artifacts from which code is automatically generated.

More details on some of these software development processes can be found in any software engineering textbook such as [Som04, Pre05]. The appendix contains introductory papers for Cleanroom engineering [CM90] and MDD [SMF03].

7.3 Requirements analysis

This course is concerned mostly with the requirements analysis, that is, the process of determining how the software is supposed to behave and documenting and analyzing this information. We therefore describe this process in more detail. Every software development process requires some kind of requirements analysis. According to [LG00], the purpose of the requirements analysis is to:

- identify the *functional requirements*, that is, to describe how a correctly functioning program responds to both correct and incorrect interactions with a user.
- identify the *performance requirements*, that is, to describe how fast certain actions must be, and how much primary and secondary storage the system

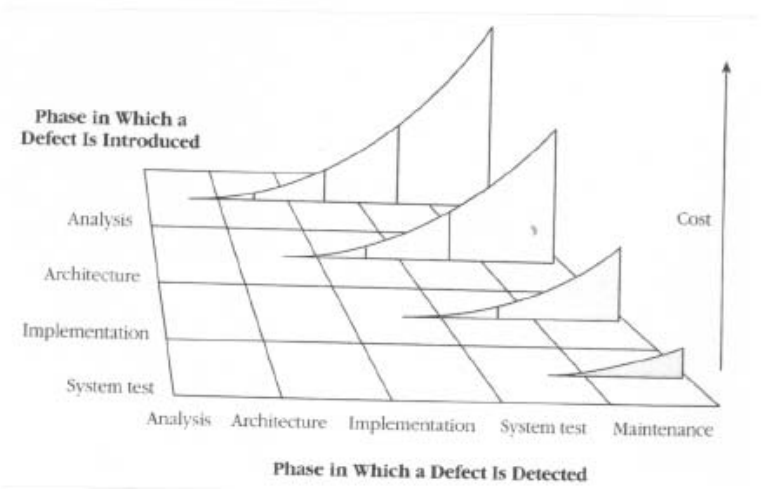


Figure 7.3: The cost of a development error increases the earlier it was introduced [McC93]

may use.

- identify *potential modifications*, that is, to describe which changes or extensions are likely in the future.
- pin down a *delivery schedule*.

The result of the requirements analysis is the *requirements document*. The requirements document contains:

- The *requirements specification* describing the functional requirements.
- A description of the performance requirements, potential modifications, and scheduling constraints.
- A discussion of alternatives that were considered and a rationale for the decisions made.

Getting the requirements right is extremely important for the success of a software development effort. Indeed, in its 1995 report, the Standish Group found that incomplete requirements and specifications were the second most important reason why projects failed [Gro95]. More recent results for safety-critical software reliant systems indicate that approximately 70% of defects are introduced during these phases and 80% of these defects are discovered late in the development cycle [RWCP10].

Problems with the requirements can render the entire development effort downstream useless, including, of course, the produced code itself. In other words, the cost of incorrect or incomplete requirements is extremely high. In [McC93],

McConnell summarizes this observation using the graph in Figure 7.3 which shows how the cost an error during the development increases the earlier it was introduced.

7.4 What is a specification anyway?

The American Heritage dictionary defines a specification as

> *"A detailed, exact statement of particulars, especially a statement prescribing materials, dimensions, and quality of work for something to be built, installed, or manufactured."*

In the software design context, such a statement can describe, for instance:

- the input-output behaviour of a method or procedure,
- a class invariant, that is, a property that all instances satisfy,
- the interaction necessary for the execution of a protocol,
- the structure and relationships between objects of a system as it executes.

The main desirable features of a specification probably are the following. A specification should be

- *as precise and detailed as necessary*, that is, it should describe the problem and its solution(s) with an appropriate amount of detail. The requirements for a word processor, for instance, do not have to specify the details of the file system the processor will be using. It does, however, have to specify which file formats the processor will be supporting. For another example, consider the following informal specification of the *pop* operation on stacks:

  > *"pop() returns the top element of the stack."*

  Strictly speaking, this specification is neither detailed nor comprehensive enough, since it neglects to say that the top element is also removed from the stack and what the result of popping an empty stack is. The point is that a specification should give the programmer enough information to design an implementation.

- *as abstract as possible*, that is, it should contain as little unnecessary detail as possible, since it may be the source of confusion, or inconsistency. Worse, it may restrict the implementer, the software or its use unnecessarily. In particular, a specification should, as much as possible, be implementation independent, that is, it should not prescribe which programming language or architecture the software is to use.

  The specification

  > *"pop() returns and removes the first element in the array that implements the stack",*

for instance, contains too much detail. The problem of determining and returning the top element of a stack has viable solutions that do not use an array in the way described above. For instance, the top element could be the last element in the array, or a vector or linked structure could be used instead of an array. The above specification constrains the programmer unnecessarily.

A high-calibre, industrial example for leaving unnecessary detail out of a specification comes from the Internet Protocol specification which, according to Vinton Cerf, vice president at Google [Cer17], was very beneficial for the development of the internet:

> *"The Internet Protocol (IP) specification does not contain any information about routing. It specifies what packets look like as they emerge from or arrive at the hosts at the edge of the Internet, but routing is entirely outside of that specificationa partly because it was not entirely clear what procedures would be used for Internet routing at the time the specification was developed and, indeed, a number of them have been developed over time. There is nothing in the specification that describes the underlying transmission technology nor is there anything in the specification that speaks to how the packet's payload (a string of bits) is to be interpreted. These matters are open to instantiation independent of the specification of packet formats."*

- *declarative rather than operational*, that is, it describes *what* problem the software is supposed to solve rather than describing *how* the problem is solved. Even experienced programmers with little background in specification may find this criterion the hardest to meet. The point is that a description of how to solve a problem is typically more detailed than a description of the problem alone, and thus also less abstract, contradicting the desire to maximize abstraction mentioned above.

  The specification of the Internet Protocol (IP) illustrates the benefits: By focussing what kind of interactions the protocol was to support, rather than how they are to be implemented, the specification allowed for decades of change and advances in communications technology to be leveraged without breaking the client applications, i.e., the upper layers.

  For a smaller, yet more concrete example, consider the following two specifications of the *pop* operation on stacks using pre- and post-conditions:

operation *pop(s,len):(s',elem,len')*

input
- s : *Array[1..n] of Data*
- $len : \mathbb{N}$

output
- s' : *Array[1..n] of Data*
- $elem$: *Data*
- $len' : \mathbb{N}$

pre-condition
- $len \leq n$ *and len holds number of elements in s*
- $len > 0$ *% s is not empty*

post-condition
- $len' = len - 1$
- $\forall\, 1 \leq i \leq len'.s'[i] = s[i+1]$
- $elem = s[1]$

and

operation *pop(s):(s', elem)*

input
- s : *Stack of Data*

output
- s' : *Stack of Data*
- $elem$: *Data*

pre-condition
- *s is not empty*

post-condition
- $s = push(elem, s')$
- $elem = s[1]$ *% elem holds first element in s*

Both specifications express properties that the input should satisfy and that the output will satisfy. So, both are, in that sense, declarative. However, the second specification is more declarative (i.e., the first specification is more operational), because it does not require the stack to be implemented as a (1-based) array giving the implementor more freedom to choose a suitable representation and even change it later. Also, to describe the effect of *pop*, it leverages the relationship that *pop* has to another stack operation: *push*. Note that the use of an array then also requires a length field *len* indicating which part of the array is actually used.

As another example, consider the following specification of a function returning the modulus of two natural numbers:

function $mod(m, n : \mathbb{N}) : \mathbb{N}$

pre-condition
- $n \neq 0$

post-condition
- $m = (m/n) \cdot n + mod(m, n)$

where m/n denotes integer division. The result returned is described in terms of the key property that relates the modulus with division, multiplication and addition.

For a final example, we consider a function computing a topological sort of a directed graph:

function $topSort(E, V) : ord$
pre-condition
$E \subseteq V \times V$ % *E is binary relation over set of vertices V*
post-condition
$ord : V \to \mathbb{N}$ % *ord is function mapping vertices to natural numbers*
$\forall\, v_1, v_2 \in V.\ (v_1, v_2) \in E \to ord(v_1) \leq ord(v_2)$
% *such that the source of an edge does not come after the target*

Note how the specification only states the property that the returned ordering is to satisfy, and contains no information about how it is to be computed. Also note how the specification does not define the result uniquely and allows for different orderings to be returned for the same input graph.

The point is that declarative specifications are usually more concise, because they contain less implementation bias. While they are typically harder for the inexperienced programmer to understand, they also tend to be less restrictive and thus more abstract.

- *correct*, that is, it should describe the problem to be solved accurately. However, specifications can be just as buggy as programs. Typos and incomplete case studies, for instance, are common sources of incorrectness.

- *consistent*, that is, it should not ask the implementer to solve an unsolvable problem. For instance, creating a state in which x is greater and less than 0 is just as impossible as deciding any problem that can be reduced to the Halting Problem.

Note that defects in the specification (ambiguity, incompleteness, incorrectness, or inconsistency) may or may not be noticed by the programmer. If they go unnoticed, there is a good chance the resulting program (and the entire development effort leading to it) may just be as useless as the specification. This highlights the need for specifications that can be analyzed to discover, for instance, the defects mentioned above. The size of modern software systems and their specifications demands that this analysis be carried out automatically, or at least, semi-automatically.

Typically, the requirements are obtained by the analyst from the potential users of the system through a process called requirements analysis or elicitation. Commercial requirements analysis tools such as IBM Rational Requisite Pro and Telelogic DOORS Enterprise RequirementsSuite facilitate the collection, documentation, and management of requirements.

7.4.1 Specification versus implementation

It is instructive to compare specifications and implementations. In contrast to specifications, implementations describe how a particular problem is to be solved. They contain so much operational detail that they are executable. Typically, specifications are not executable. For instance, the post-condition

$s = push(elem, s')$ in the specification of $pop(s)$ does not tell us how the removal of the top element from the stack can be implemented.

However, sometimes a specification can be executable despite its abstract, declarative nature and its lack of operational detail. The arrival of Prolog, for instance, created a lot of excitement, because it demonstrated that declarative specifications *can* be executable. Consider, for instance, the following Prolog program to determine whether or not an element occurs in a list.

```
member(X, [X|_]).
member(X, [Y|Ys]) :- X =/= Y, member(X, Ys).
```

The program is declarative, because it is a direct transcription of the definition of something being an element of a list:

> *"X is contained in list l, if and only if X is the first element of l, or X is contained in the rest of l."*

Just like the definition, it contains no unnecessary operational content. For instance, the order in which the clauses and goals are listed is irrelevant. Both can be permuted arbitrarily and the program still gives the correct answer. So, if only the "logical fragment" of Prolog is used, the resulting programs are executable, declarative specifications. The addition of non-logical operators like cut ("!") or input-output statements destroys this property. Prolog programs become implementations rather than specifications.

7.5 Formal specifications

7.5.1 What are they?

Formal specifications are specifications given in some formal, mathematical notation. The syntax and semantics of that notation is precisely and unambiguously defined. Propositional and predicate logic are both examples of notations that can (and are) used for formal specifications.

7.5.2 When and why use them?

Not every software development requires or merits the large-scale use of formal specifications. The design of some Unix or DOS shell script to test some application may benefit from the occasional informal comment, but most likely would not require the use of any formal notation except the code itself. If, however, the system has a non-trivial size and the correct functioning of the produced software is particularly important and malfunctioning very costly (e.g., safety-critical software, air-traffic control software) the use of a formal specification language probably is advisable. The main reasons are the following:

- Formal specification languages have a formal semantics. A formal semantics uniquely determines the meaning of every expression typically

using mathematical notations and concepts like sets, functions, or relations. Formal specification languages thus allow the precise, unambiguous expression of requirements.

- Even for relatively small systems the functional requirements can be quite complex. Most formal specification languages offer mechanisms like abstraction, modularity and reuse. These mechanisms aid the specifier in controlling large and complex requirements.
- The information available from customers and users is often lacking in, for instance, precision and quality. The process of expressing requirements formally helps uncovering ambiguities, inconsistencies, or incompletenesses. It may be important to be able to identify and correct such problems early, because flaws in early stages of the life cycle tend to be more costly than those made in later stages. More concretely, flaws in the requirements specification, if uncovered too late, can have severe consequences on the entire development effort. For instance, all implementation efforts to realize a specification that turns out to be inconsistent may be wasted.
- As already mentioned above, the size and complexity of typical specifications often makes tool support necessary. Only specifications with a precise semantics are amendable to an automatic analysis that, for instance, checks the specification for completeness or consistency.

7.5.3 Writing formal specifications

Writing formal specifications and programming are, at the same time, similar and different activities. They are similar, because they require the mastery of the syntax and semantics of a formal notation. Before you can use Java meaningfully and effectively, you must know what well-formed Java programs look like and precisely what, for instance,

```
((Employee) v.getElementAt[i+1]).pay();
```

means. A similar remark applies to predicate logic. Java and predicate logic are "tools" that allow you to achieve some goal such as expressing some algorithm or some information. Before you can use these tools you must learn their capabilities and limitations.

On the other hand, programming and specifying are also quite different. Both have different audiences and serve different purposes. Specifications are typically read by humans and describe the problem that is to be solved. Programs contain detailed instructions for a machine on how to solve this problem. As we have already discussed, programs tend to be more operational with (possibly) lots of implementation bias, while specifications tend to be more declarative with (ideally) little implementation bias.

CISC422 is meant to be an introduction to formal specification. It presents a variety of useful formal notations (e.g., propositional logic, predicate logic,

temporal logic), and discusses their uses, advantages and disadvantages, and associated analyses.

Detailed information on how to write formal specifications is given in [Win95].

7.5.4 Where are formal specifications used?

Formal specifications are used in, for instance, pre- and post-conditions, class invariants, test case and protocol descriptions. They can come in the form of, for instance, predicate logic, regular expressions, or state machines. Many software development processes employ formal specifications during their requirements analysis design phases. For instance, the Unified Modeling Language (UML) offers the Object Constraint Language (OCL) for expressing constraints in object-oriented development formally [WK99, Gro14]. Some methodologies are even based on formal specification such as Cleanroom engineering and Model-Driven Development (MDD). In MDD, formal models of the software and its requirements form the primary artifacts from which the code is automatically generated. MDD has been very successful for the development of embedded software and is supported through commercial tools such as IBM Rational Rose Technical Developer (formerly known as Rational Rose RealTime) and Telelogic Tau. Cleanroom engineering has been developed at IBM for the development of safety-critical software. See the accompanying papers in the appendix for more details on MDD [SMF03] and Cleanroom engineering [CM90].

However, formal specification and analysis can also be used during the implementation phase or quality assurance phase. Model checking, for instance, can be used to overcome the limitations of testing and to analyze models of code for correctness extremely rigorously. The last part of the course will introduce this technique. Recently, software model checking has been developed in which the code (rather than a model of it) is directly analyzed.

Exercise 7.5.1. Why are programming languages not well suited as a specification notation?

Chapter 8

Classical specification and development formalisms

8.1 Natural language

Natural language is the most common way to express requirements, because it offers three important advantages:

- It is very expressive.
- There is no need to learn a new formalism.
- It is sufficient for many purposes.

However, it is important to be aware of the limitations and disadvantages of natural language as a specification formalism.

- Natural language is often imprecise, ambiguous, or inconsistent. The following examples of ambiguous specifications are taken from [BKK03]:
  - *"For up to 12 aircraft, the small display format shall be used. Otherwise, the large display format shall be used."* Assuming that small and large display formats have been defined previously, the ambiguity lies in the phrase "up to 12". It is not clear whether it includes 12 or not.
  - *"Aircraft that are non-friendly and have an unknown mission or the potential to enter restricted air-space within 5 minutes shall raise an alert."* The problem here is that the relative precedence of "and" and "or" is unclear.
  - *"Process P1 negotiated with process P2 and process P3 and process P4 negotiated with P5."*

  - *"All members have a unique membership number".* The problem is that we do not know whether every member has its own unique number, or whether all members share the same unique number.
  - *"If $x > 0$ then check if $y > 0$. If yes, then set x to 1, else set x to 2".* Should x be set to 2 if $x > 0 \wedge y \leq 0$ or if $x \leq 0$?

  Another example is from [Jac98]:

  - *"Everybody likes a holiday."* We know what is intended here, but strictly speaking it is not clear whether everybody likes the same holiday ($\exists\, h.\, \forall\, x.likes(x, h)$), or if people may have different holidays that they like ($\forall\, x.\, \exists\, h.likes(x, h)$).

  The problem is that natural language is full of semantic ambiguities and subtleties.

- Attempts to make an informal specification more precise and unambiguous, may make it prohibitively long and verbose.

- Natural language cannot be analyzed automatically. Even if we had perfect grammars and dictionaries. One problem is that everything we say has a "context". Sometimes, that context is incredibly important for the correct interpretation of an English sentence. In fact, we are so used to automatically interpreting everything in context that it is even hard to see. In [Jac98], Michael Jackson offers the following real-life illustration of this. Suppose you enter an elevator containing two signs that read:

  Taken literally, every user of the elevator not only has to wear shoes, but also carry a dog. This is, however, not the intended meaning. We know that bringing dogs into public places can be problematic. So, we automatically read the second sentence as "If you have a dog, you must carry it".

  The following example is taken from [BEKV94]. Consider the following three sentences:

  1. *"She sang like her sister."*
  2. *"She sang like a nightingale."*
  3. *"He sang like a canary."*

  Note how the meaning of "sing" changes from one sentence to the other due to the subtleties of the English language and its use.

  A similar example illustrating ambiguity is due to Gause and Weinberg in [GW89]. Consider the familiar nusery rhyme *"Mary had a little lamb".* Armed with a dictionary, we find that "to have" and "lamb" can have

many meanings, some quite different from the intended one: E.g., "to have" can mean "to hold in possession as property", but also "to bear" (as in "to have a baby") and "to eat" (as in "to have lunch"); similarly, "lamb" can mean "a young sheep", but also, "a person as gentle or weak as a lamb" or "the young of various animals (e.g,. antelopes)". So, without any additional context knowledge *"Mary had a little lamb"* could be interpreted to mean, e.g., *"Mary ate a small, gentle person"* or *"Mary gave birth to a little young antelope"*.

Finally, the following joke shows the importance of punctuation:

> *"A panda walked into a cafe. He ordered a sandwich, ate it, then pulled out a gun and shot the waiter. 'Why?' groaned the injured man. The panda shrugged, tossed him a badly punctuated wildlife manual and walked out. And sure enough, when the waiter consulted the book, he found an explanation. 'Panda,' ran the entry for his assailant. 'Large black and white mammal native to China. Eats, shoots and leaves'."*

The joke illustrates how the presence or absence of a comma can change the meaning of a sentence dramatically. On the other hand, misplacing a comma is easy to do and hard to check for using grammar checking tools. So, the fact that misplaced commas have cost companies millions of dollars should not come as a surprise [SW18].

These examples illustrate vividly how subtle not only the semantics but also the syntax of natural language is and how challenging a task it is to give a computer enough smarts to deal with all of them.

Meyer gives a detailed critique of the use of natural language in specifications in [Mey85]. He concludes that every natural language specification will have flaws even after detailed review.

8.2 Predicate logic

Predicate logic is very expressive, well-studied and understood. A number of tools like editors, proof checkers, and theorem provers are available to facilitate writing and analyzing predicate logic specifications. Every computing undergrad is getting some introduction to logic. These things do make predicate logic an appealing choice as a formal specification notation.

Consider, for instance, an operation *add* on dictionaries modeled as elements of $\mathcal{P}(Key \times Value)$, i.e., sets of *Key*, *Value* pairs:

$$add : Key \times Value \times \mathcal{P}(Key \times Value) \rightarrow \mathcal{P}(Key \times Value)$$

The effect of executing *add* on an existing dictionary d using the key k and the value v can be captured by the following specification:

$$\begin{array}{l}\forall\, d : \mathcal{P}(Key \times Value).\, \forall\, d' : \mathcal{P}(Key \times Value).\, \forall\, key : Key.\, \forall\, val : Value. \\ \quad d' = add(k, v, d) \quad \leftrightarrow \\ \qquad \big((\neg\ \exists\, v' : Value. \langle k, v' \rangle \in d) \quad \rightarrow \quad d' = d \cup \{\langle k, v \rangle\}\big) \wedge \\ \qquad \big(\exists\, v' : Value. \langle k, v' \rangle \in d. \quad \rightarrow \quad d' = d - \langle k, v \rangle \cup \langle key, val \rangle\big)\end{array}$$

The specification says that a dictionary d' is the result of adding a key k and a value v to some other dictionary d if and only if d' is equal to $d \cup \{\langle key, val \rangle\}$ if k is not already used, and otherwise d' is just like d except that the old pair with k is removed and the new pair $\langle k, v \rangle$ added.

The pre- and post-condition notation makes this a bit more readable:

operation *add(k,v,d)*

input
- k : *Key*
- v : *Value*
- d : *Dictionary over Key* $\times$ *Value*

output
- d' : *Dictionary over Key* $\times$ *Value*

---

pre-condition

true — *% no constraints on the input*

post-condition

$\big((\neg\ \exists\, v' : Value\ .\ \langle k, v' \rangle \in d) \quad \rightarrow \quad d' = d \cup \{\langle k, v \rangle\}\big) \wedge$ — *% case 1*

$\big(\exists\, v' : Value\ .\ \langle k, v' \rangle \in d. \quad \rightarrow \quad d' = d - \{\langle k, v \rangle\} \cup \{\langle key, val \rangle\}\big)$ — *% case 2*

However, one problem with using plain predicate logic is its lack of support for modularity which is crucial for managing large specifications. In other words, large specifications in predicate logic usually are very hard to understand. The "classical" specification notations mentioned below (such as Z, VDM, B, Event-B) are based on logic but add a mechanism that allow them to break large specifications into smaller, manageable pieces.

Moreover, the expressiveness of predicate logic also has a down side. Predicate logic is undecidable. No tool will ever be able to determine, for instance, consistency or validity of predicate logic specifications in all generality and fully automatically. Some user interaction and thus expertise will always be required when using a theorem prover. For more details on predicate logic, see Part II.

8.3 Z

We briefly present the most characteristic features of Z. For more information, the reader is referred to [Spi89, Bow, Sch01].

The Z notation (pronounced as *zed* and named after the the German mathematician Ernst Zermelo) originated at Oxford University in the late 70s. The main goal was to develop a specification formalism that achieves precision, conciseness, and modularity without imposing unnecessary constraints. Since then it has evolved into a well-studied formal specification language with the following characteristics:

- Z is based on first-order predicate logic and set theory. Its built-in theory contains booleans, integers, reals, relations, functions, sets, bags (sets with duplicates), and sequences and the standard operations on them.
- Z is *typed*, that is, every entity in Z must have a type associated with it.
- *Schemas* encapsulate specifications of initial states, constants, types, functions, and operations.
- Schemas can be composed using Z's *schema calculus*.
- A *refinement calculus* allows the stepwise refinement of specifications into code.

8.4 Other formalisms

There are numerous other "classical" specification formalisms. The most popular include the Vienna Development Method (VDM) [Jon86], Larch [Ge93], Lotus [vEVD89], the B method [Sch01], the Event-B method [Abr10], and Abstract State Machines [BS03].

Part IV

Using Formal Specifications for Class Modeling and Analysis (Weeks 4-6)

Chapter 9

Class modeling (Week 4)

Since the mid 1990ties, object orientation has become very popular. There are many reasons for this development. A number of systems can naturally be thought of a collection of interacting objects. These objects often are related through a type hierarchy which is supported by inheritance and allows code reuse. Dynamic method dispatch (sometimes also called polymorphism) allows the invocation of different methods on different objects through the same piece of code and thus helps writing succinct code.

How can the specification and design of object-oriented systems best be supported? Lots of processes for object-oriented systems development have been suggested. Each of these processes is centered around their own set of models and employ their own notation. However, class models play a central role in most of them.

The class model is the key design representation in most object-oriented methods. It shows what objects there are and how they are connected. In the requirements analysis and specification phases, class models are essential for ensuring that the right domain notions have been identified, and that their relationships have been captured correctly. In the design phase, class models are essential for ensuring that the most relevant concepts (and only those) and their properties have been identified.

This document will discuss two different approaches to the specification and analysis of class models. The first approach is based on *Alloy*, a class modeling notation developed at MIT [Jac06b]. The second approach is based on UML and OCL and will be sketched very briefly in Chapter 12.

Chapter 10

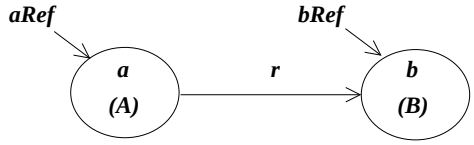
Class models vs state diagrams: Templates vs instances (Week 4)

Essentially, a *class model* (sometimes also called "object model") describes the runtime states a system is allowed to run through during execution. More precisely, it describes the structure of the states (e.g., What are the objects? What fields does an object have?), the relationships between state components (e.g., How are objects related to each other?), and the constraints the states must obey (e.g., Can the field of this object be null? Can this object be shared, that is, pointed to, by several other objects? Which invariants do the objects have to satisfy?).

Before we can discuss class models in more detail, we must first describe how a single state will be represented.

10.1 State diagrams

The runtime state of an object-oriented program can always be represented textually by listing the values of all attributes of all objects in that state. However, often a graphical notation is more concise. *State diagrams* (sometimes also called *object diagrams* or *instance diagrams*) provide a graphical description of a single runtime state. Suppose, for instance, we have a state s that contains exactly two objects, one object a of class A, and another object b of class B. Object a is accessed via variable *aRef* and object b via variable *bRef*. Furthermore, suppose that in s object a contains a field r whose value is object b. The state diagram of s looks as follows.



Objects are represented by ovals and arrows between ovals represent fields of objects. Field arrows are labeled with the name of the field. The diagram may also include the values of variables, shown as arrows from a variable name to an object. Optionally, ovals can be labeled with identifiers for talking about particular objects, and show the type of an object in parentheses.

Consider, for instance, the following Java code implementing parts of a simple file system.

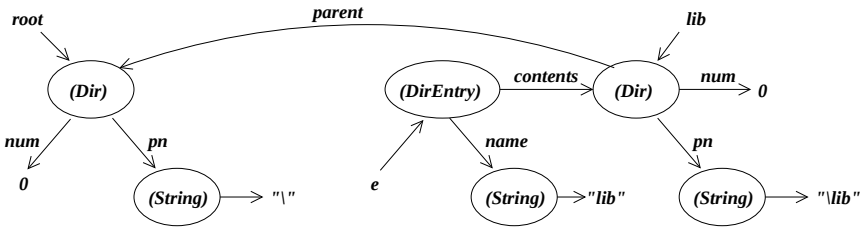
```
class DirEntry {
  Dir contents;
  String name;
  DirEntry (Dir c, String n) {
    contents = c;
    name = n;
  }
}

class Dir {
  String pn;
  Dir parent;
  Vector entries;
  int num;
  Dir (Dir p, String n) {
    parent = p;
    pn = n;
    entries = new Vector();
    num = 0;
  }
  void add (DirEntry c) {
    entries.addElement(c);
    num = num + 1;
  }
}
```

Note that for simplicity, the above code does not account for files. A directory can only contain other directories. We now can create directory objects using, for instance, the following code

```
Dir root = new Dir(null, "\");
Dir lib = new Dir(root, "\lib");
DirEntry e = new DirEntry(lib, "lib");
```

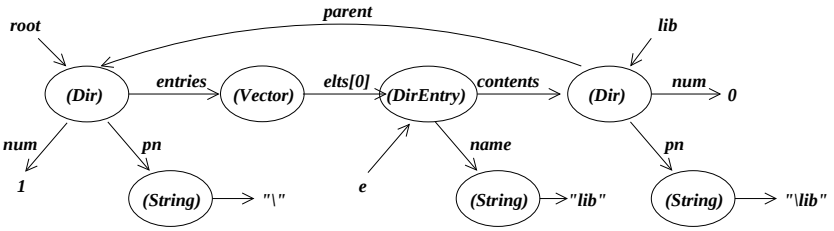
which results in the state:



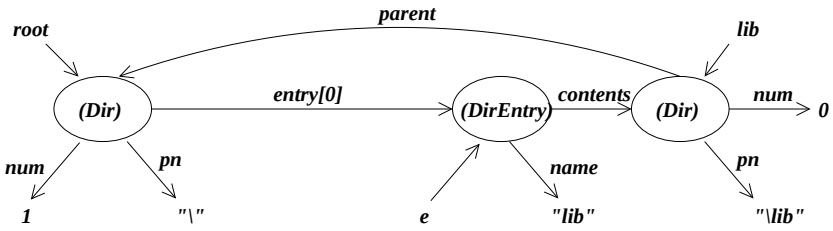
We add the directory entry e into the root directory.

```
root.add(e)
```

which results in the state

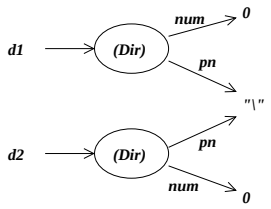


Primitive values are not considered objects. State diagrams distinguish primitive values from objects by not enclosing them in ovals. Often, we will want to elide objects such as the vector and only show the more interesting user defined objects. For instance, the vector is part of the representation of the directory object, and a client that only calls the methods of *Dir* doesn't see a *Vector* object. So, we might draw instead

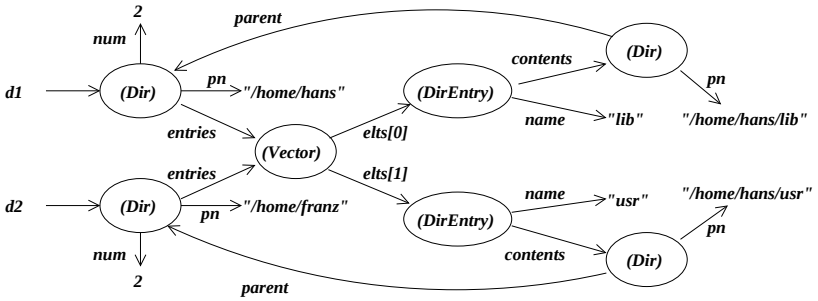


in which the *abstract field* arrow labeled *entry* from the *Dir* object to the *DirEntry* object hides the *Vector* object. Note that for the sake of brevity we also have elided the oval for *String* objects.

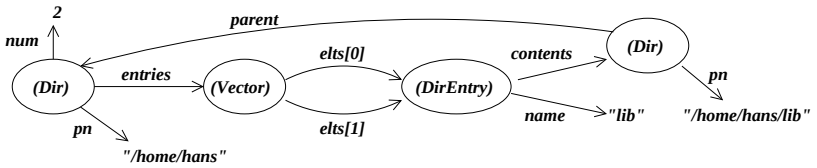
Many states can be created using the above code, but not all of them are desirable. In this state two directories share the same path name.



In the next diagram, we have a state in which all directory entries always belong to two directories.



In the next state, a directory contains the same entry twice.



Whether these are in fact problematic depends on the intent of the designer of these classes. As we will see, the class model describes a set of allowed state diagrams by succinctly summarizing some code features and adding textual constraints about potential clients of the code.

For instance, suppose we constrained instances of the directory class by the following representation invariant:

The value of num is equal to the number of elements in entries.

Now, the second state shown above exemplifies a problematic situation. Suppose two directories *d1* and *d2* share their entries. If an entry is added to *d1*, the *num* counter of *d1* is incremented and the *num* counter of *d2* is not. After the addition, directory *d2* will thus violate the representation invariant.

10.2 Class models

Programs manipulate state. This state often has a complex structure, even if we limit our attention to that part of the structure that is visible to the user. For

instance, a user of a file system needs to understand the purpose and structure of files and directories and the relationship between them. We use class models to describe the potentially infinite set of program states a system is allowed to exhibit. You can think of a class model as representing a set of state diagrams. Many problems involve states or configurations, so the use of class models is not confined to requirements specifications. In constructing a class model, one is forced to answer a variety of important questions — a process that often reveals inadequate understanding of the problem. More precisely, a class model consists of

- a declaration of the sets, subsets, and relations
- a description of the relationships between the sets and relations

The result is a reasonably precise definition of which components and relationships make up the program state. Moreover, object models form a good basis for program design, since the designer is more likely to understand what to build.

Chapter 11

Class model design and analysis using Alloy (Weeks 4-6)

Alloy is an object modeling notation developed at MIT [Jac06b]. This section serves as an introduction to class modeling in general, the Alloy notation, and the Alloy Constraint Analyzer. We will use a running example to illustrate the various concepts. The software can be obtained from `https://alloytools.org` where also additional documentation can be found. We will be using Alloy5 (v5.1.0).

11.1 Example: Graphs, trees, and lists (Week 4)

The Alloy GUI allows us to load Alloy specifications, and to edit, compile, and analyze them.

11.1.1 Signatures and fields

After selecting `New` in the `File` pull-down menu, we input the following Alloy specification into the editor window on the left (note that internal editor must be enabled in the `Options` menu).

```
module Graph
sig Node {}                       // signature Node
sig Name {}                       // signature Name
sig Graph {                       // signature Graph
        name : Name,              // every Graph has exactly one Name
        initial : set Node,       // every Graph has 0 or more initial Nodes
        curr : lone Node,         // every Graph has 0 or 1 current Nodes
        next : Node -> Node       // every Graph has a successor relation
}
```

It declares three *signatures* called *Node*, *Name* and *Graph*. Comments start with `//`. A signature stands for a set of atoms. We cannot make any assumptions about the interpretation, structure and order of the elements of a signature. Top-level signatures (i.e., those that do not extend any other user-defined signature) are mutually disjoint. The signature *Graph* signature declares four *fields*.

- The field *name* associates a name with each graph, or, more precisely, it associates an atom from signature *Name* with each atom from signature *Graph*. Formally, *name* is total function that returns a *Name* for every *Graph*.
- The field *initial* associates a set of nodes with each graph, or, more precisely, it associates a set of atoms from signature *Nodes* with each atom from signature *Graph*. The Alloy keyword *set* is the power set constructor. Formally, *initial* is an unconstrainted relation between *Graph* and *Node*.
- The field *curr* associates with each graph either the atom *null* or exactly one node. The notation *lone A* adds the atom *null* to the set of atoms A. Formally, *curr* is a possibly partial function from *Graph* to *Node*.
- The field *next* associates with each graph a binary relation *next* on nodes. The notation $A \rightarrow B$ denotes the *cross product* of the sets A and B, that is,

  $$A \rightarrow B \;=\; \{(a, b) \mid a \in A \wedge b \in B\}.$$

  Formally, *next* is function from *Graph* to (*Node*→*Node*). That is, given a graph *next* returns a binary relation between *Nodes*. Note that the cross product is also often denoted by $A \times B$.

11.1.2 Finding instances of an Alloy specification

An Alloy specification is a class model and thus stands for a set of states. More precisely, every state that satisfies the constraints expressed in a specification S is said to be a *state* or a *satisfying instance of* S. We can use Alloy to find sample states of a class model. By adding the lines

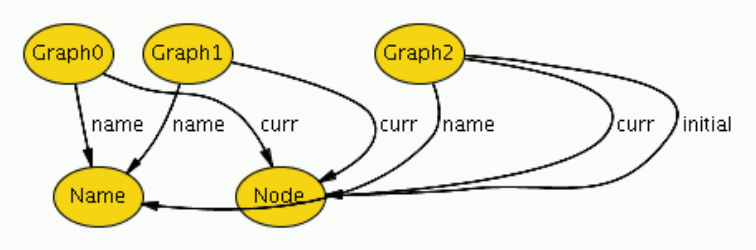
```
pred Show {}
run Show for 3
```

to our Alloy specification and save it (as *graphs.als*, for instance). The Alloy command *run* in *graphs.als* causes Alloy to find a state of the specification and is called with two arguments: the name of a predicate and a *scope*. Informally, the predicate specifies additional constraints the state has to satisfy, while the scope limits the number of candidate states Alloy has to consider. More precisely, the scope places an upper bound on the number of atoms in each signature in the specification. So, the command

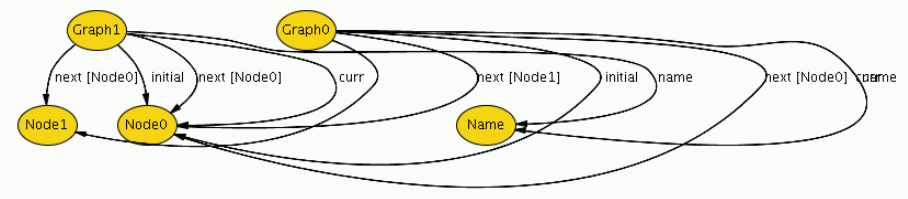
```
run Show for 3
```

in the above example causes Alloy to attempt to find a state of the specification in scope 3, that is, a satisfying instance that does not use more than three atoms from each signature.

We can ask Alloy to find a state of the above specification by clicking `Execute` and selecting `Run Show for 3`. Alloy will inform us that an instance (i.e., solution) was found. There are different formats to display the instance. The most interesting for us are `Viz` (visualization) and `Tree`. If we choose the visualization, we see[1]

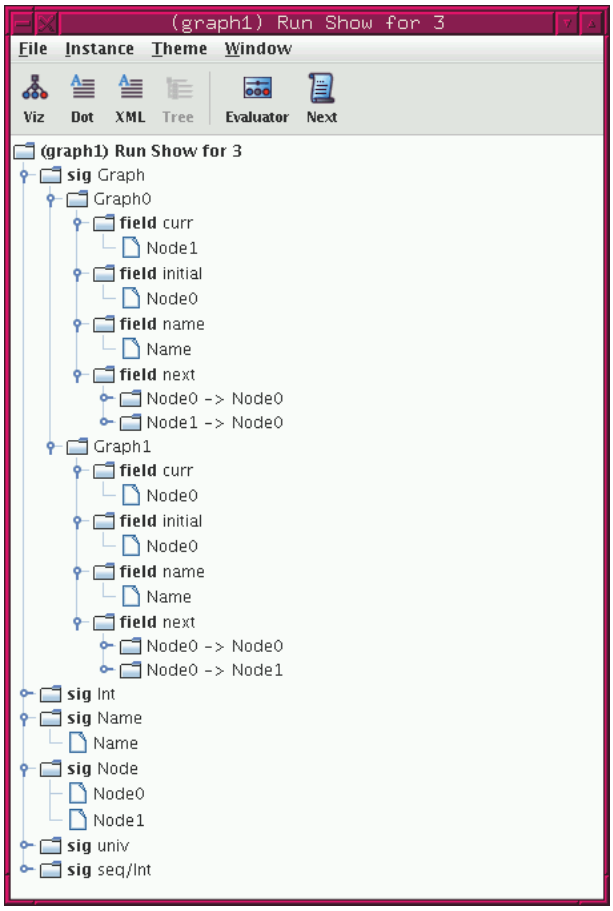


By clicking the `Next` button, more instances can be generated. Here is another instance found by Alloy:



This instance has two graph objects (*Graph0*, *Graph1*), one name object (*Name*) and two node objects (*Node0*, *Node1*). The successor relation for *Graph0* is such that *Node0* has itself as successor (indicated by the edge from *Graph0* to *Node0* labeled *next[Node0]*) and *Node1* has *Node0* as successor (indicated by the edge from *Graph0* to *Node0* labeled *next[Node1]*). The successor relation for *Graph1* says that *Node0* has two successors (*Node0* and *Node1*) and *Node1* has no successor. In the Alloy GUI "Themes" can be used to customize the layout of the instances and to make them more legible. Alternatively, an instance can be represented in terms of a tree. Here is the tree for the instance above:

[1] By default, Alloy displays objects as rectangles; for us, however, objects will always be shown by ovals (ellipses) and classes as rectangles; to change Alloy's visualization layout, choose `Theme` in the visualization window.



Note that signatures *univ*, *Int*, and *seq/Int* are always part of any model. Signature *univ* contains all atoms, and thus is similar to, e.g., the root class `Object` in Java. Signatures *Int* and *seq/Int* contain a subset of the integers. Both representations tell us that the instance consists of the following sets and relations:

Name of set	Atoms contained in it
Graph	{*Graph0*, *Graph1*}
Name	{*Name*}
Node	{*Node0*, *Node1*}

Name of relation	Tuples contained in it
curr	{(*Graph0*, *Node1*), (*Graph1*, *Node0*)}
initial	{(*Graph0*, *Node0*), (*Graph1*, *Node0*)}
name	{(*Graph0*, *Name*), (*Graph1*, *Name*)}
next	{ (*Graph0*, *Node0*, *Node0*), (*Graph0*, *Node1*, *Node0*), (*Graph1*, *Node0*, *Node0*), (*Graph1*, *Node0*, *Node1*) }

To force Alloy to find a state with only one graph we add a constraint to the predicate *Show*.

```
pred Show {
        one Graph
}
```

The formula

```
one Graph
```

expresses that the set *Graph* must contain exactly one atom where *one* is one of Alloy's existential quantifiers. Alloy has five quantifiers: *all*, *some*, *no*, *one*, and *lone*. Alloy quantifiers are used the same way as quantifiers in predicate logic. For instance,

```
one x : X | f
```

is true precisely if there exists exactly one atom in X such that replacing that atom for x in f makes f true. Given a set S and an Alloy quantifier *quant*, the notation

$$quant\ S$$

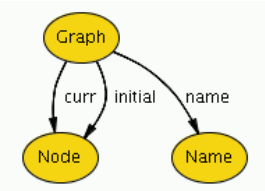
is short for

$$quant\ x : S \mid true.$$

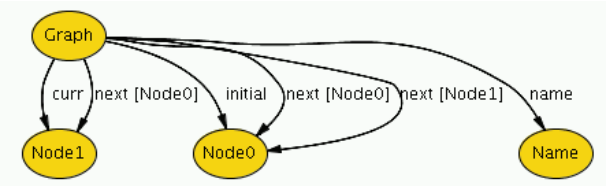
For instance, *one* S is true precisely if

$$quant\ x : S \mid true,$$

that is, if S is a singleton set. After adding `one Graph` to predicate `Show`, example states found by Alloy include:



and



where, e.g., the edge from *Graph* to *Node1* with label *next[Node0]* means that *Node1* is the successor of *Node0* in *Graph*, i.e., *(Graph, Node0, Node1)*$\in$*next*. Internally, the compilation translates the specification S and the given scope constraint into a predicate logic formula φ_S. Solving S means, in logical terms, finding an assignment of the symbols in φ_S to values that makes φ_S true. In Part II (Sections 3.2 and 4.2), we called such assignments models; in the context of Alloy, we will call them *satisfying instances*. A state of S is a satisfying instance of φ_S. Note that the constraints expressed in the specification can be contradictory, that is, the formula φ_S can be equivalent to false. In that case, the checker will not find a state and respond with

```
No instance found. Predicate may be inconsistent. 0ms.
```

For instance,

```
pred Show {
        one Graph          // signature Graph has exactly one element and
        no Name            // signature Name is empty
}
```

is contradictory, because in our specification, every graph has exactly one name. Thus, the moment we have at least one graph, we also have at least one name.

11.1.3 Checking properties of a specification

Assertions are used to check properties of a specification. For instance, after adding

```
assert CurrAtMostSingleton {
        // every graph has at most one current Node
        all g : Graph | lone g.curr
}
```

to *graphs.als* and

```
check CurrAtMostSingleton for 3
```

the analysis checks that in every instance of the specification in scope 3, the field *curr* contains at most one element. In general, *lone* $x : X \mid f$ is true precisely if either there is no x in X that makes f true, or there is exactly one such x in X. Consequently, given a set T, *lone* T holds if and only if T has at most one element. Note that in assertion *CurrAtMostSingleton* Alloy uses a "." to reference the fields of an object. For now, we can think of the dot-notation as navigation just like in, e.g., Java. The true meaning of "." will be discussed later.

When checking an assertion A, the checker attempts to find a *counter example* for φ_A, that is, a state of the specification within the scope that violates

φ_A. Assertion A is valid in the specification if and only if such a counter example does not exist. More precisely, given specification S and assertion A, the checker shows $\varphi_S \rightarrow \varphi_A$ by showing that $\varphi_S \wedge \neg\, \varphi_A$ is unsatisfiable, that is, that $\varphi_S \wedge \neg\, \varphi_A$ has no solution. Consequently, φ_A is valid in S if and only if $\varphi_S \wedge \neg\, \varphi_A$ does not have a satisfying instance.

In this case, Alloy responds with

```
No counterexample found. CurrAtMostSingleton may be valid. 20ms.
```

meaning that in the specified scope every instance of our *Graph* class model is such that a graph has at most one current node. Note that this is not surprising at all, because the declaration *curr : lone Node* enforces this.

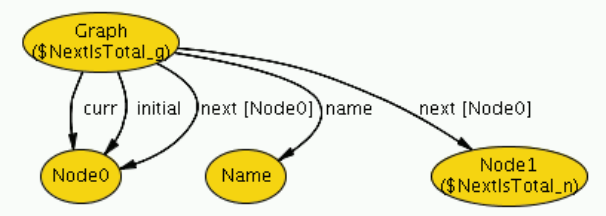
For another example, consider the assertion

```
assert NextIsTotal {
        all g : Graph | all n : Node | some n.(g.next)
}
```

which expresses that every node in every graph has at least one successor. More precisely, $n.(g.next)$ is the set of nodes that can be reached in one step from node n using g's transition relation, $g.next$. Thus, *NextIsTotal* requires this set to be non-empty for all graphs g and nodes n, that is, given any node n in any graph g, n always has at least one successor node. In general, given a relation $r : A \rightarrow B$ and an element $a : A$, $a.r$ denotes the *relational image* of a under r, that is, it is the set of $b : B$ that are related to a via r. Formally,

$$a.r \;=\; \{b \mid (a, b) \in r\}$$

The assertion *NextIsTotal* is not valid in *Graph* in scope 3 (or any scope). Alloy produces the following counterexample



in which *Node1* does not have a successor.

The assertion

```
assert AllNodesAreReachable {
        all n : Node | some g : Graph | n in (g.initial).*(g.next)
}
```

also is invalid. It expresses that for every node n, there exists a graph g, such that n can be reached from the initial nodes of g by following the *next* relation zero or more times. Formally, $*$ is a relational operator, that, when given a binary relation $r : A \rightarrow A$ builds the reflexive and transitive closure of r, that is, $*r$ is the relation

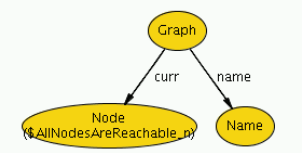
$$*r \;=\; \bigcup_{i=0} r^i$$

where r^0 is the identity relation over A and $r^{i+1} = r.r^i$ and $r.s$ denotes relational composition. Consequently,

$$(n_0, n_k) \in *(g.next)$$

if and only if n_k can be reached from n_0 by using *next* zero or more times, that is, if $n_0 = n_k$ or there exist $n_1, \ldots, n_{k-1}$ such that $(n_{i-1}, n_i) \in r$ for all $1 \leq i \leq k$. Thus, $(g.initial).*(g.next)$, for instance, stands for the set of all nodes that can be reached from one of g's initial nodes using $g.next$, g's transition relation. Finally, $n \in (g.initial).*(g.next)$ expresses that n is reachable from the initial states of g.

When analyzing assertion *AllNodesAreReachable*, the checker produces the following counter example:



In this instance, the graph has no initial node and an empty successor relation. Thus, *Node* is not reachable from any of the initial nodes of the graph. More precisely, we have

```
(Graph.initial).*(Graph.next) = {}
```

11.1.4 Invariants

We want to eliminate isolated nodes and force every node in a graph to be reachable from the initial nodes. To this end we add the following fact

```
fact {
        all g : Graph, n : Node | n in (g.initial).*(g.next)
}
```

to our specification. Facts, also called *invariants*, enforce all states of a specification to satisfy a given property, that is, the checker will only consider those states that satisfy the facts. Consequently, after adding the above fact, the assertion *AllNodesAreReachable* now is valid in the specification *Graph* in scope 3.

Facts can be incorporated into signature declarations. This allows a further notational simplification.

```
sig Graph {
        name : Name,
        initial : set Node,
        curr : lone Node,
        next : Node -> Node
}{
        all n : Node | n in initial.*next
}
```

Contraints attached to signature declarations need to be read slightly differently than the "stand-alone" facts that we have been considering so far. Given

```
sig A {...}{f}
```

constraint f is equivalent to

```
all this : A | f'
```

where f' is just like f except that each occurrence of a field g in f or one of its supertypes is replaced by *this.g*. To prevent a field name from being expanded like this, you can prefix it with "@". For example, the signature above is equivalent to

```
sig Graph {
        name : Name,
        initial : set Node,
        curr : lone Node,
        next : Node -> Node
}{
        all g : Graph | all n : Node | n in (g.@initial).*(g.@next)
}
```

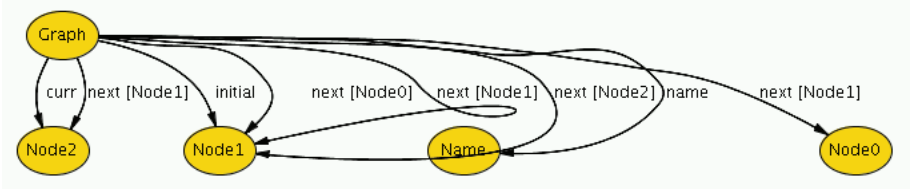
But, note that these signature-integrated facts can also be written as separate facts using the **fact** keyword. For example, the following specification of *Graph* is equivalent to the two preceding ones.

```
sig Graph {
        name : Name,
        initial : set Node,
        curr : lone Node,
        next : Node -> Node
}
fact {
   all g : Graph | all n : Node | n in (g.initial).*(g.next)
}
```

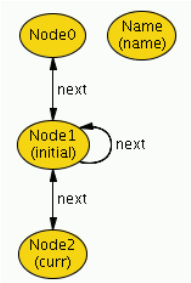
After also adding a fact saying that current nodes better be reachable

```
fact CurrentNodesAreReachable {
        all g : Graph | g.curr in (g.initial).*(g.next)
}
```

Alloy finds the following instance



After projecting on the object *Graph*, the connections between nodes in the instance become more legible:



11.1.5 Extension

A signature can extend another signature. For instance, the *Graph* signature can be extended into a signature for acyclic graphs.

```
sig AcyclicGraph extends Graph {
}{
        no n : Node | n in n.^next
}

pred Show {
        one Graph
        one AcyclicGraph
}
```

where the quantifier *no* expresses that no node satisfies the formula that follows and the relational operator $\hat{}next$ denotes the *transitive closure* of *next*. Given a binary relation $r : A \rightarrow A$, the transitive closure $\hat{}r$ is the relation

$$\hat{}r \;=\; \bigcup_{i=1} r^i$$

Thus,

$$n.\hat{}next$$

is the set of all nodes that can be reached from n in one or more steps using the *next* relation. The fact

$$no\ n : Node \mid n.\hat{}next$$

expresses that no node is its own successor. In general, the formula $no\ x : X \mid f$ holds if and only if there is no atom in X that makes f true.

Given two signatures, S and $subS$, the clause $subS$ *extends* S expresses a subset relationship. More precisely, all atoms in $subS$ also are atoms in S. For instance, every atom in *AcyclicGraph* also is an atom in *Graph*. However, not every atom in *Graph* also is in *AcyclicGraph*. Note that subsignatures of a signature are mutually disjoint (just like "top-level" signatures). So, in, e.g.,

```
sig A {}
sig B {}
sig A1 extends A {}
sig A2 extends A {}
```

A and *B* are disjoint, and so are *A1* and *A2*, but we cannot conclude that *A1 + A2 = A*. A check of this new specification with the assertion

```
assert Acyclic {
        all d : AcyclicGraph, n : Node | n !in n.^(d.next)
}
```

where ! denotes negation, shows that the specification is correct and that all states of the specification are indeed acyclic.

We now extend acyclic graphs further into trees using

```
sig Tree extends AcyclicGraph {
        root : initial
}{
        all n : Node | lone n.~next
}
```

where $\sim r$ denotes the inverse of r, that is,

$$\sim r \;=\; \{(b, a) \mid (a, b) \in r\}.$$

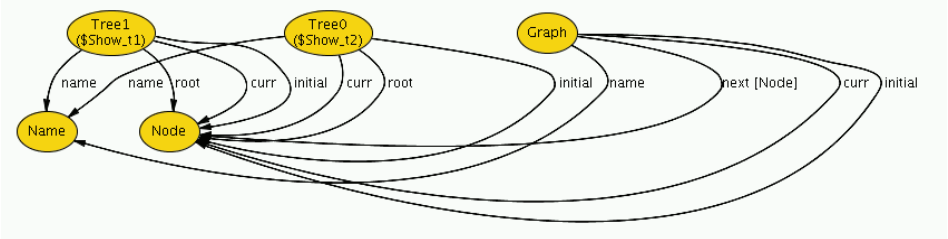
The specification says that trees are acyclic graphs that have one distinguished initial node called root, and whose nodes have at most one predecessor, that is,

```
lone n.~next
```

has at most one element.

11.1.6 An inconsistent specification

In the current specification, two trees may share nodes. For instance, the following is an instance of our specification:



In other words, while a node can have at most one predecessor in any given tree, it can have additional predecessors in other trees. To remedy this situation, we add a subsignature with a constraint saying that no node is reachable from the roots of two trees.

```
sig TreeNoSharing extends Tree {
}{
        all t1, t2 : TreeNoSharing | no (t1.@root).*(t1.@next) &
                                        (t2.@root).*(t2.@next)
}
```

where & denotes set intersection. There is a problem with the above specification. When we ask the checker to find a satisfying instance using

```
pred Show {
        some TreeNoSharing
}
run Show for 3
```

we find that our specification now is contradictory, because the checker responds with

```
No instance found. Predicate may be inconsistent. 39ms.
```

The problem is that if $t1$ and $t2$ are the same there *are* nodes that are reachable from the root of $t1$ and $t2$. We forgot to say that $t1$ and $t2$ must be different.

```
all t1, t2 : TreeNoSharing | t1 != t2 =>
                                    no (t1.@root).*(t1.@next) &
                                       (t2.@root).*(t2.@next)
```

or

```
all t1 : TreeNoSharing, all t2 : TreeNoSharing - t1 |
                                    no (t1.@root).*(t1.@next) &
                                       (t2.@root).*(t2.@next)
```

where => and − denote implication and set difference respectively.

11.1.7 Functions

Functions allow us to state the above property more legibly and succinctly.

```
fun reachableNodes[g : Graph] : set Node {
        {n : Node | n in (g.initial).*(g.next)}
}
sig TreeNoSharing extends Tree {
}{
        all t1, t2 : TreeNoSharing |
                t1 != t2 => no (reachableNodes[t1] & reachableNodes[t2])
}
```

The notation $\{a : A \mid f\}$ in function *reachableNodes* denotes the set of all atoms in A that satisfy f and is commonly called *set comprehension*. While set comprehension is very useful in general to construct sets of elements satisfying a certain property, in the above case it is not necessary. The definition

```
fun reachableNodes[g : Graph] : set Node {
        (g.initial).*(g.next)
}
```

is equivalent yet shorter.

Rather than defining a function that returns the nodes reachable in a graph, we could have defined a predicate *reachable(g, n)* that returns true if and only if node n is reachable in graph g.

```
pred reachable[g : Graph, n : Node] {
        n in (g.initial).*(g.next)
}
sig TreeNoSharing extends Tree {
}{
        all t1, t2 : TreeNoSharing |
            t1 != t2 => no n : Node | reachable[t1, n] && reachable[t2, n]
}
```

where `&&` denotes logical conjunction. Note the predicate above does not contain a declaration of a result type.

The whole Alloy specification for the running example used in this section is shown in Figure 11.1.

```
module Graph

sig Node {}

sig Name {}

sig Graph {
        name : Name,            // every Graph has exactly one Name
        initial : set Node,     // every Graph has 0 or more initial Nodes
        curr : lone Node,       // every Graph has 0 or 1 current Nodes
        next : Node -> Node,    // every Graph has a successor relation
}

fact GraphFacts {
        // all nodes belong to at least one graph
        all n : Node | some g : Graph | n in (g.initial).*(g.next)
        // current nodes belong to a graph
        all g : Graph | g.curr in (g.initial).*(g.next)
}

sig AcyclicGraph extends Graph {
}{
        no n : Node | n in n.^next
}

sig Tree extends AcyclicGraph {
        root : initial
}{
        all n : Node | lone n.~next
}

sig TreeNoSharing extends Tree {}{
        all t1, t2 : TreeNoSharing | t1 != t2 =>
                no (reachableNodes[t1] & reachableNodes[t2])
}

fun reachableNodes [g : Graph] : set Node {
        (g.initial).*(g.next)
}

assert CurrAtMostSingleton {
        all g : Graph | lone g.curr
}

assert NextIsTotal {
        all g : Graph | all n : Node | some n.(g.next)
}

assert AllNodesAreReachable {
        all n : Node | some g : Graph | n in (g.initial).*(g.next)
}

assert Acyclic {
        all d : AcyclicGraph, n : Node | n !in n.^(d.next)
}

pred Show() {
        some TreeNoSharing
}

run Show for 4
check CurrAtMostSingleton for 4
check NextIsTotal for 4
check AllNodesAreReachable for 4
check Acyclic for 4
```

Figure 11.1: Alloy specification used as running example

Functions and predicates can also be used for the specification of operations. Here is the specification of an operation that adds a new node as the root of a tree; the old root becomes the one and only child; additionally, the new root becomes the current node.

```
fun addNodeAtRoot[before, after : Graph, n : initial] {
        after.root = n
        after.curr = n
        (after.root).(after.next) = before.root
        after.name = before.name
}
```

11.1.8 Scalars are singletons

Next, let us specify lists as trees without sharing that have a last node and whose nodes have at most one successor.

```
sig List extends Tree {
        last : Node
}{
        all n : Node | lone n.next
//      last = {n : Node | no n.next}          // explicit definition
        no last.next                           // implicit definition
}
```

The specification shows an explicit and implicit definition of *last*. Both definitions are equivalent. However, the implicit definition is shorter and thus preferable. The explicit definition is surprising. The field *last* is declared as a variable ranging over nodes. However, the explicit definition assigns a singleton set to it. Alloy's type checker does not complain, because Alloy does not distinguish between the two. In other words, scalars are singleton sets in Alloy. More precisely, the declaration $a : A$ declares a field named a containing singleton sets of atoms of A. If the specification forces a to take on a non-singleton set as a value, the specification will be contradictory and thus have no satisfying instances. Consider, for example, the following specification.

```
module Test

sig A {
}{
        some a1, a2 : A | a1 != a2             // A has at least 2 elements
}

sig B {
        a1, a2, a3, a4 : A
}{
        a1 = a2                                // ok
        a2 = {a : A | a in a1}                 // ok
//      a3 = A                                 // contradiction
//      a4 = {a : A | a in a1 && a !in a1}     // contradiction
}

pred Show {
        some B                  // show state with at least one atom in B
}

run Show for 3
```

The declaration $b : set\ B$ removes the singleton constraint and allows b to range over sets of atoms of B of all sizes. Similarly for a declaration $c : lone\ C$.

We can now properly explain the meaning of `in`: `in` always denotes the subset relation. The constraint

```
reachableNodes[t1] in reachableNodes[t2]
```

for instance, says that the reachable nodes of tree `t1` are a subset of the reachable nodes of tree `t2`. Consider the constraint

```
all n : Node | all t : Tree | n in reachableNodes[t]
```

Here, `in` can be read as "is element of". However, since the scalar `n` denotes a singleton set containing `n`, it also denotes subset.

The main advantage of treating scalars as singletons is notational uniformity. For instance, the relational image $e.r$ of an expression e under a relation r can be treated the same way regardless of whether e evaluates to a scalar or a set.

11.1.9 Multiplicity constraints

Finally, doubly linked lists are lists with a partial function *prev* that is the inverse of the *next* function.

```
sig DoublyLinkedList extends List {
        prev : Node lone -> lone Node
}{
        prev = ~next
}
```

where *lone* denotes a multiplicity constraint. In general, the source end and the target end of every binary relation $r : A \rightarrow B$ can be decorated with multiplicity constraints as follows $r : A\ m \rightarrow n\ B$. The meaning of the constraint m on the source end is that every element in B is reached through r from m elements in A. The meaning of the constraint n on the target end is that every element in A is related to n elements in B through r. There are four different choices for m and n: *set*, *lone*, *one*, and *some*. Their meaning is as follows:

set	*zero or more*
lone	*zero or one*
one	*exactly one*
some	*one or more*

Multiplicity *set* is the default and does not need to be mentioned explicitly. In other words, $r : A\ set \rightarrow set\ B$ is the same as $r : A \rightarrow B$. Multiplicities *lone* and *one* can be used to, e.g., turn a relation into a function, or to make it injective or surjective. For instance, the constraints in $r : A \rightarrow one\ B$ and $s : A \rightarrow lone\ B$ make r and s total and partial functions respectively. $t : A\ one \rightarrow B$ is an injective (one-to-one) relation. The declaration $pred : Node\ lone \rightarrow lone\ Node$ expresses that every node has at most one predecessor (*lone* at target end) and that every node is the predecessor of at most one node (*lone* at source end).

11.1.10 Integer expressions

Alloy has some limited support for the use of integers through the built-in signature *Int*. The most important ways of constructing integer expressions are

- constants (e.g., 0, 1, 42, -13),
- the cardinality operator (#) as in, e.g., $\#Node$ and $\#\{n : Node \mid no\ n.next\}$, and
- the standard arithmetic operators:
  - *plus[a,b]*: returns the sum of a and b
  - *minus[a,b]*: returns the difference between a and b
  - *mul[a,b]*: returns the product of a and b
  - *div[a,b]*: returns the number of times b divides a

Given two integer expressions e_1 and e_2, a boolean expression can be built using the standard comparison operators: $e_1 = e_2$, $e_1 < e_2$, $e_1 > e_2$, $e_1 <= e_2$, and $e_1 >= e_2$.

Note that the notion of scope has a different meaning for *Int* and does not restrict the number of integers available. Instead, it restricts the bitwidth of the integers considered. From the Alloy Language Reference [Jac06a]:

> "For example, if the scope specification assigns 4 to *Int*, there are four bits for every integer and integer-valued expression, and the set *Int* may contain values ranging from -8 to +7, including zero. All integer computations are performed within the given bitwidth, and if, for a given instance, an expression's evaluation would require a larger bitwidth to succeed without overflow, the instance will not be considered by the analysis."

By default, *Int* has scope 4. This scope can be changed with an appropriate scope specification.

11.1.11 Scope specifications

So far, the scope to be used for a `run` and the `check` command was given by a single integer. In these cases all signatures defined in the Alloy model are given the same scope. It is possible to give different signatures different scopes:

```
module Test
sig A {...}
sig B {...}
sig C {...}
run1: run {...} for 2
run2: run {...} for 3, but 4 C, 5 Int
```

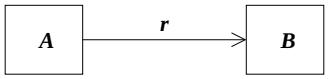
In the above specification, command `run1` will give all user-defined signatures scope 2, i.e., it will consider instances in which the interpretations of signatures `A`, `B`, and `C` have at most 2 instances and in which all integer expressions evaluate to a number between -8 and 7 (since scope 4 is used by default for signature *Int*). With command `run2`, however, signatures `A` and `B` are given scope 3, signature `C` gets scope 4, and integer expressions can range over -16 to 15 (scope/bitwidth 5).

11.1.12 How to represent Alloy specifications graphically

Certain aspects of an Alloy specification can easily be represented graphically by using a notation familiar from UML class diagrams. Extension is shown using an arrow with a hollow head

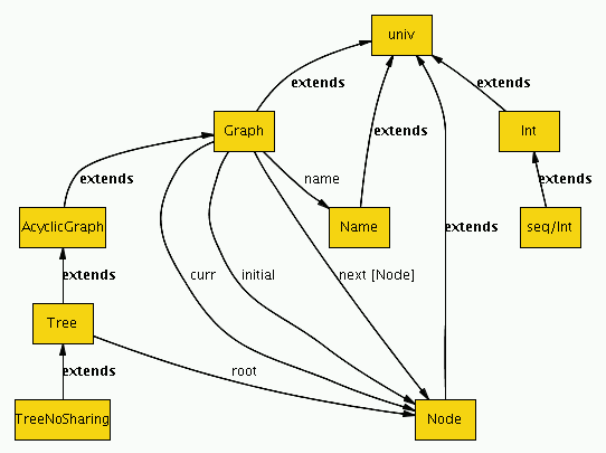

```
sig B {}
sig A extends B {}
```

whereas binary relations are shown using arrows with a standard head.

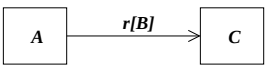



```
sig B {}
sig A {
      r : set B
}
```

The Alloy GUI does not quite stick to this convention and always uses a filled arrow head with extension arrows marked by a label. For instance, selecting `Execute | Show metamodel` in the GUI, Alloy generates the following graphical representation of the specification in Figure 11.1.



It is not obvious how to display relations of arity greater than 2. Alloy adopts the convention that a relation $r : A_1 \to (A_2 \to \ldots (A_{n-1} \to A_n) \ldots)$ is shown as an arrow from A_1 to A_n with label $r[A_2, \ldots, A_{n-1}]$ as in, for example,




```
sig C {}
sig B {}
sig A {
      r : B -> C
}
```

and some of the previous specifications.

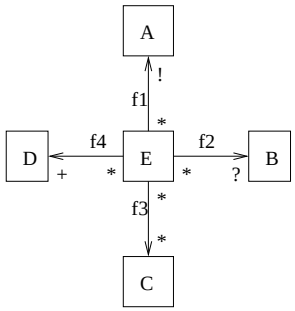
While we cannot represent all textual constraints graphically, multiplicity constraints can easily be shown (as in, e.g., UML class diagrams). For brevity, we adopt the following notation:

Symbol	Meaning
$*$	*zero or more*
?	*zero or one*
!	*exactly one*
$+$	*one or more*

11.1.13 How to represent graphical multiplicity constraints textually

Sometimes, we are given a graphical representation of a class model (in, e.g., Alloy or UML notation), and then need to create the corresponding textual representation in Alloy. While this is straight-forward for the most part, dealing with multiplicities requires some care. We will show the translation by example.

- **Dealing with multiplicities on the target end of a relation**. The keywords `lone` and `set` help us here. The Alloy specification

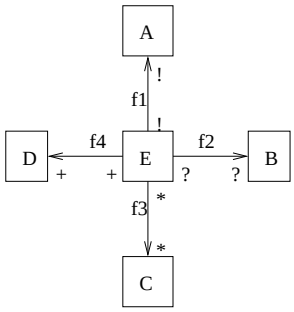


  is represented as

```
sig A {}
sig B {}
sig C {}
sig D {}
sig E {
        f1 : A            // 'one' on target of f1
        f2 : lone B       // 'lone' on target of f2
        f3 : set C        // 'set' on target of f3
        f4 : some D        // 'some' on target of f4
}
```

  Note that the encoding of $+$ in the declaration of `f4` requires an additional fact that prevents `f4` to be null.

- **Dealing with multiplicities on the source end of a relation**. All non-trivial multiplicity constraints at the source end of a relation need to be expressed by facts. The Alloy specification



is represented as

```
sig A {}
sig B {}
sig C {}
sig D {}
sig E {
        f1 : A            // 'one' on target of f1
        f2 : lone B       // 'lone' on target of f2
        f3 : set C        // 'set' on target of f3
        f4 : some D       // 'some' on target of f4
}
fact Multiplicities {
        all a : A | one a.~f1      // 'one' on source of f1
        all b : B | lone b.~f2     // 'lone' on source of f2
        all d : D | some d.~f4     // 'some' on source of f4
}
```

11.2 Summary of Alloy language (Week 5)

11.2.1 Expressions

Every Alloy expression r denotes (evaluates to) a (possibly empty) relation, that is, a (possibly empty) set of tuples $(a_0, \ldots, a_n)$ with $n \geq 0$. The following special cases arise: The relation denoted by r may contain

- nothing. In that case, r represents the empty relation $\emptyset$.
- a single unary tuple, e.g., (a_0). In that case, r can be thought of just the atom (or, scalar, value) a_0.
- more than one unary tuple, e.g., (a_0), (a_1), (a_2). In that case, r can be thought of as the set $\{a_0, a_1, a_2\}$.
- one or more non-unary tuples, e.g., $(a_1, \ldots, a_n)$ for $n \geq 2$. In that case, r represents an n-ary relation. Every tuple in a relation has the same length.

Alloy expressions are built from constants, variables and operators. Constants are the names of signatures and fields; signature `Node`, for instance, stands for a set of nodes, and field `next` stands for a set of pairs of nodes. Alloy offers

the standard operators (union, intersection, difference) on expressions (that is, sets):

+	*union*
&	*intersection*
–	*difference*

We have already seen how to obtain the *relational image* $s.r$ of a set $s : set\ A$ under a relation $r : A \rightarrow B$:

$$s.r \quad = \quad \{b : B \mid \exists\, a : s \;.\; (a, b) \in r\}$$

Note that $s.r$ can also be written as $r[s]$ in Alloy.

Turns out, relational image is a special case of a more general operation on relations: *relational composition*. Two tuples

$$(a_1, a_2, \ldots, a_{n-1}, a_n)$$

and

$$(b_1, b_2, \ldots, b_{m-1}, b_m)$$

can be composed, if $a_n = b_1$. In that case, their composition is the tuple

$$(a_1, a_2, a_{n-1}, b_2, \ldots, b_{m-1}, b_m).$$

The relational composition $s.r$ of two relations s and r is the set of all compositions of tuples in s and r. If s is a set of atoms and r is a relation, $s.r$ specializes to relational image discussed above. If, however, s is a relation and r is a set of atoms, $s.r$ denotes the *relational pre-image* of r under s, that is,

$$s.r \quad = \quad \{a : A \mid \exists\, b : r \;.\; (a, b) \in s\}$$

We have seen two closure operations: *transitive closure* and *reflexive and transitive closure*. The transitive closure $\hat{}\,r$ of a binary relation $r : A \rightarrow A$ is the smallest relation that contains r and is transitive. The reflexive and transitive closure $*r$ is the smallest relation that contains r and is reflexive and transitive.

$$\begin{array}{lcll} \hat{}\,r & = & r + r.r + r.r.r + \ldots & = \bigcup_{i=1} r^i \quad \textit{transitive closure} \\ *r & = & id_A + \hat{}\,r & = \bigcup_{i=0} r^i \quad \textit{reflexive and transitive closure} \end{array}$$

where $r : A \rightarrow A$ for some A and id_A is the identity relation on A, that is, $id_A = \{(a, a) \mid a \in A\}$.

The *transpose* (or *inverse*) of a relation r reverses the order of all tuples in r:

$$\tilde{}\,r \quad = \quad \{(a_n, \ldots, a_1) \mid (a_1, \ldots, a_n) \in r\}$$

The *product* $s \rightarrow r$ of two relations s and r is obtained by taking every tuple in s and concatenating it with every tuple in r. That is,

$$s \rightarrow r \quad = \quad \{(a_1, \ldots, a_n, b_1, \ldots, b_m) \mid (a_1, \ldots, a_n) \in s \wedge (b_1, \ldots, b_m) \in r\}$$

A special operator to build expressions is *set comprehension*:

$$\{x : r \mid f\}$$

where x is a variable, r is an expression, and f is a formula (as defined below). For instance,

```
{n : Node | some g : Graph | n in (g.initial).(g.next)}
```

evaluates to a set containing all nodes n which, in some graph g, are the successor of one of the initial nodes of g.

11.2.2 Formulas

The smallest boolean expressions, i.e., atomic formulas, are obtained by comparing two expressions: r *in* s is true if and only if every tuple in r also is in s. $r = s$ is true if and only if r *in* s and s *in* r.

Let r, s denote expressions:

r `in` s	*subset: every element of r also is an element of s*
r `=` s	*set equality: r and s contain the same elements*

Moreover, Alloy contains the standard boolean connectives familiar from predicate logic. It only uses ASCII symbols to represent them.

Let f and g denote formulas:

`!`f	*negation: not f*
f `&&` g	*conjunction: f and g*
f `||` g	*disjunction: f or g*
f `=>` g	*implication: if f then g*
f `<=>` g	*equivalence: f if and only if g*
`all` $x : r \mid f$	*universal quantification: f true for all x in r*
`some` $x : r \mid f$	*existential quantification: f true for at least one x in r*
`one` $x : r \mid f$	*f true for exactly one x in r*
`lone` $x : r \mid f$	*f true for at most one x in r*
`no` $x : r \mid f$	*f true for no x in r*

Note that quantifiers can be applied to expressions, too.

`some` r	*r contains at least one element*
`one` r	*r contains exactly one element*
`lone` r	*r contains at most one element*
`no` r	*r contains no element*

11.3 Syntax and semantics of the Alloy kernel (Week 6)

To finish off the discussion of the Alloy language itself, we now present the syntax and semantics of the Alloy kernel language (i.e., its smallest core language) formally.

11.3.1 Syntax of Alloy's kernel language

Recall that Propositional and Predicate Logic contain some redundancy, in the sense that some connectives can be defined in terms of others. The notion of *adequate connectives* (Section 3.2.3 in Part II) then allowed us to identify the smallest possible core of each language, from which the whole language can be bootstrapped.

To make the language more user-friendly, Alloy contains a very similiar kind of redundancy. To simplify the definition of the semantics, we identify Alloy's *kernel language* below, i.e., the language with the smallest number of constructs from which all other constructs of the language can be derived.

Syntax of signatures in Alloy's kernel language

Let *Sigs* denote a set of signature declarations of the form

```
sig s_i {
    ...
    a_i : s_{i_1} -> ... -> s_{i_k}
    ...
}
```

where s_i is the name of the signature and a_j is the name of an attribute in signature s_i with arity $k+1$.

Let $\mathcal{F}_{Sig}$ and $\mathcal{F}_{Attr}$ denote the sets of signature names and attribute names in *Sigs* respectively. Formally, the syntax of a signature declaration is

sig	::=	`sig` s `{}` | `sig` s `{` *attrs* `}`	signature declaration
attrs	::=	*attr* | *attr attrs*	attribute declaration
attr	::=	a `:` *type*	
type	::=	`set` s | `set` s `->` *type*	

where $s \in \mathcal{F}_{Sig}$ and $a \in \mathcal{F}_{Attr}$.

Attributes names have a type which indicates which signature they are part of and which kinds of values they can take on. We assume the existence of a mapping

$$type : \mathcal{F}_{Attr} \quad \rightarrow \quad \big(\mathcal{F}_{Sig} \times \mathcal{F}_{Sig}\big) \cup \ldots \cup \underbrace{\big(\mathcal{F}_{Sig} \times \ldots \times \mathcal{F}_{Sig}\big)}_{maxArity \ times}$$

where *maxArity* is the largest arity of any of the attributes, i.e.,

$$maxArity = max\{arity(a) \mid a \in \mathcal{F}_{Attr}\}$$

Example Let *Sigs* contain the signature

```
sig s_1 {
    ...
    a : set s_2-> set s_3
    ...
}
```

Then, $s_1 \in \mathcal{F}_{Sig}$, $a \in \mathcal{F}_{Attr}$, and $arity(a) = 3$. Also, $type(a) = (s_1, s_2, s_3)$, indicating that a is an attribute of the signature s_1 and can take on values of type $s_2 \times s_3$.

Syntax of formulas in Alloy's kernel language

Assuming a set of variables $\mathcal{V}$, a set of signature names $\mathcal{F}_{Sig}$ and a set of attribute names $\mathcal{F}_{Attr}$, the syntax of formulas in the Alloy kernel is:

φ	::=	*expr* `in` *expr* $\mid$	atomic formulas
		`!`$\varphi \mid \varphi$ `&&` $\varphi \mid$	composed formulas
		`all` *var* : *expr* $\mid \varphi$	quantified formulas

Expressions are defined by:

expr	::=	*name* $\mid$ *var* $\mid$ `none` $\mid$ *expr binOp expr* $\mid$ *unOp expr*
binOp	::=	`+` $\mid$ `&` $\mid$ `-` $\mid$ `.` $\mid$ `->`
unOp	::=	`~` $\mid$ `^`

where $name \in \mathcal{F}_{Sig} \cup \mathcal{F}_{Attr}$ and $var \in \mathcal{V}$.

Symbols in Alloy's kernel language

Remember that the syntax of Predicate Logic was given in terms of three sets of symbols: A set $\mathcal{V}$ of *variables* which are used for quantification, a set $\mathcal{F}$ of *functions symbols* of different arity greater or equal to zero, and a set $\mathcal{P}$ of predicate symbols also of different arity greater or equal to 1. Variables and function symbols are used for building *terms* which always denote elements in the semantic domain $\mathcal{D}^{\mathcal{M}}$. Predicate symbols are used to build atomic formulas.

Before we can start defining the semantics of Alloy's kernel language, we need to determine and classify the symbols it uses.

- The set $\mathcal{V}$ of *variables* again contain variables used for expressing quantification only. Note that elements of $\mathcal{V}$ do not represent attribute names used in signatures.
- The set $\mathcal{F}$ of *functions symbols* contains function symbols for *relational expressions* and names, respectively.

  $$\mathcal{F} = \mathcal{F}_{Op} \cup \mathcal{F}_{Name}$$

  where

  $$\mathcal{F}_{Op} = \{\texttt{none}\} \cup \{\texttt{~}, \texttt{\^{}}\} \cup \{\texttt{+}, \texttt{\&}, \texttt{-}, \texttt{.}, \texttt{->}\}$$

  and `none` has arity zero, `~` and `^` have arity one, and the symbols in {`+`, `&`, `-`, `.`, `->` } all have arity two. The symbols in $\mathcal{F}_{Name}$ represent names of signatures and attributes.

  $$\mathcal{F}_{Name} = \mathcal{F}_{Sig} \cup \mathcal{F}_{Attr}$$

As mentioned above, each attribute has an arity and a type, which can be determined using the mapping *type*.

- The set $\mathcal{P}$ of *predicate symbols* only contains `in` which has arity 2:

$$\mathcal{P} = \{\texttt{in}\}$$

11.3.2 Semantics of Alloy's kernel language

Let *Spec* be an Alloy specification with signatures *Sigs*, facts *Facts*, run predicate P, and assertion A. Let $\mathcal{V}$ be the set of variables *var* mentioned in *Facts*, P, or A. Let $\mathcal{F}_{Sig}$ and $\mathcal{F}_{Attr}$ be the set of signature names and attribute names used in *Sigs*. We define the following formulas over $\mathcal{V}$, $\mathcal{F}_{Sig}$, and $\mathcal{F}_{Attr}$:

φ_{Facts}	formula representing the conjunction of all facts in *Facts*
φ_P	formula representing run predicate P
φ_A	formula representing assertion A

Definition of instances

In Predicate Logic, the notion of a *model* $\mathcal{M}$ was used to identify a *semantic domain* $\mathcal{D}^{\mathcal{M}}$ and to assign mathematical functions over $\mathcal{D}^{\mathcal{M}}$ to the function symbols and predicate symbols, so that the truth or falsehood of a formula in $\mathcal{M}$ could be determined. In Alloy, we have been calling these models *instances*. We will now define these instances formally. The definition will be similar to the definition of models in Sections 3.2 and 4.2.

The 3-tuple

$$\mathcal{I} = (\mathcal{D}^{\mathcal{I}}, \mathcal{F}^{\mathcal{I}}, \mathcal{P}^{\mathcal{I}})$$

is called an *instance of the specification Spec* if it interprets the symbols in $\mathcal{F}$ and $\mathcal{P}$ as shown below.

Definition of semantic domain $\mathcal{D}^{\mathcal{I}}$ For each signature name s_i in $\mathcal{F}_{Sig}$ we assume an infinite set of unique atoms as given. Unique here means that every such atom is associated with exactly one signature name. We can think of these atoms as representing the objects of the class represented by the signature:

$$\mathcal{D}_{s_i} = \textit{infinite set of unique atoms for interpretation of signature } s_i$$

for all signature names s_i in $\mathcal{F}_{Sig}$. To allow for the uniform interpretation of the navigation operation "." as relational composition mentioned in Section 11.1.8, we need to make relational composition work on sets of atoms. To this end, we need to turn each atom d in $\mathcal{D}_{s_i}$ into a 1-tuple (d):

$$\mathcal{D}^{\mathcal{I}}_{s_i} = \{(d) \mid d \in \mathcal{D}_{s_i}\}$$

Informally, the semantic domain $\mathcal{D}^{\mathcal{I}}$ is given by the set of

(1) all (unary) relations over $\mathcal{D}^{\mathcal{I}}_{s_i}$ for all signature names $s_i \in \mathcal{F}_{Sig}$, and

(2) all relations over $\mathcal{D}^{\mathcal{I}}_{s_1} \times \ldots \times \mathcal{D}^{\mathcal{I}}_{s_n}$ (i.e., subsets of $\mathcal{D}^{\mathcal{I}}_{s_1} \times \ldots \times \mathcal{D}^{\mathcal{I}}_{s_n}$) for all attribute names $a \in \mathcal{F}_{Attr}$ with $type(a) = (s_1, \ldots, s_n)$.

Formally, we have

$$\begin{aligned} \mathcal{D}^{\mathcal{I}} &= \left\{ d \subseteq \mathcal{D}^{\mathcal{I}}_{s_i} \mid s_i \in \mathcal{F}_{Sig} \right\} \cup & (1) \\ &\quad \left\{ d \subseteq \mathcal{D}^{\mathcal{I}}_{s_1} \times \ldots \times \mathcal{D}^{\mathcal{I}}_{s_n} \mid \exists\, a \in \mathcal{F}_{Attr}.type(a) = (s_1, \ldots, s_n) \right\} & (2) \end{aligned}$$

Part (1) expresses that for every signature s_i every possible interpretation of s_i as a unary relation over $\mathcal{D}^{\mathcal{I}}_{s_1}$ is a semantic value, i.e., an element in the semantic domain $\mathcal{D}^{\mathcal{I}}$. Part (2) says that for every attribute a with type $(s_1, \ldots, s_n)$, every possible interpretation of a as an appropriately typed relation (i.e., subset of $\mathcal{D}^{\mathcal{I}}_{s_1} \times \ldots \times \mathcal{D}^{\mathcal{I}}_{s_n}$) also is a semantic value. As mentioned before, we can see that every element in $\mathcal{D}^{\mathcal{I}}$ is a relation; thus, every expression in Alloy will evaluate to one of these relations.

Example: Assume that *List* and *Node* are signature names in $\mathcal{F}_{Sig}$ and that $\mathcal{F}_{Attr}$ contains attribute names *head* and *next* with arity two and types

$$\begin{aligned} type(head) &= (List, Node) \\ type(next) &= (Node, Node) \end{aligned}$$

and that φ_{Facts} and φ_P are formulas using these names. Also assume that the atoms in $\mathcal{D}_{Node}$ and $\mathcal{D}_{List}$ are denoted by $N0, N1, N2, \ldots$ and $L0, L1, L2, \ldots$, respectively. The semantic domain $\mathcal{D}^{\mathcal{I}}$ will, e.g., contain the following elements:

$\{(N0), (N1), (N2)\}$	possible interpretation of signature *Node*
$\{(N0)\}$	possible interpretation of signature *Node*
$\emptyset$	possible interpretation of signature *Node*
$\{(L0)\}$	possible interpretation of signature *List*
$\{(L0), (L1), (L2), (L3)\}$	possible interpretation of signature *List*
$\{((N0), (N1)), ((N1), (N2))\}$	possible interpretation of attribute *next*
$\{((L0), (N0))\}$	possible interpretation of attribute *head*
$\emptyset$	possible interpretation of attribute *head*

The semantic domain $\mathcal{D}^{\mathcal{I}}$ will, e.g., not contain the following elements:

$\{((N0), (N1), (N2)), ((N1), (N2), (N3))\}$	no attribute of matching type
$\{((N0), (L0)), ((N1), (L1))\}$	no attribute of matching type

Definition of interpretation $\mathcal{F}^{\mathcal{I}}$ of function symbols As for Predicate Logic, $\mathcal{F}^{\mathcal{I}}$ interprets the function symbols in the Alloy specification *Spec*. To this end, it will contain an interpretation $f^{\mathcal{I}}$ for every symbol $f \in \mathcal{F} = \mathcal{F}_{Op} \cup \mathcal{F}_{Name}$. If $f \in \mathcal{F}_{Op}$, then $f^{\mathcal{I}}$ is a function on $\mathcal{D}^{\mathcal{I}}$ with the expected arity and definition. If $f \in \mathcal{F}_{Name}$, then $f^{\mathcal{I}}$ is a type-consistent element from $\mathcal{D}^{\mathcal{I}}$:

Case 1: $f \in \mathcal{F}_{Op}$**, i.e.,** $f \in \{$`none`, `~`, `^`, `+`, `&`, `-`, `.`, `->`$\}$
The interpretation of all these symbols is *fixed*:

$$\begin{aligned}
\texttt{none}^{\mathcal{I}} &= \emptyset \in \mathcal{D}^{\mathcal{I}} \\
\texttt{\~{}}^{\mathcal{I}} &= \text{the unary function that reverses the input relation } d \in \mathcal{D}^{\mathcal{I}} \\
&\quad \text{i.e., } (d_k, d_{k-1}, \ldots, d_2, d_1) \in \texttt{\~{}}^{\mathcal{I}}(d) \text{ iff } (d_1, d_2, \ldots, d_{k-1}, d_k) \in d \\
\texttt{\^{}}^{\mathcal{I}} &= \text{the unary function that builds the } \textit{transitive closure} \text{ of} \\
&\quad \text{binary relations } d \text{ for all } s \in \mathcal{F}_{Sig} \text{ with } \textit{type}(d) = (s, s) \\
\texttt{+}^{\mathcal{I}} &= \text{the binary function that returns the } \textit{union} \text{ of its input} \\
\texttt{\&}^{\mathcal{I}} &= \text{the binary function that returns the } \textit{intersection} \text{ of its input} \\
\texttt{-}^{\mathcal{I}} &= \text{the binary function that returns the } \textit{difference} \text{ of its input} \\
\texttt{.}^{\mathcal{I}} &= \text{the binary function that returns the } \textit{relational composition} \text{ of} \\
&\quad \text{its input} \\
\texttt{->}^{\mathcal{I}} &= \text{the binary function that returns the } \textit{cartesian product} \text{ of} \\
&\quad \text{its input}
\end{aligned}$$

Case 2: $f \in \mathcal{F}_{Name} = \mathcal{F}_{Sig} \cup \mathcal{F}_{Attr}$
For all symbols representing names of signatures or attributes the interpretation $f^{\mathcal{I}}$ of f will *vary from instance to instance.*

If f is a signature name, i.e., $f \in \mathcal{F}_{Sig}$, then the interpretation $f^{\mathcal{I}}$ of f will be given by a subset of $\mathcal{D}^{\mathcal{I}}_f$, i.e., the set of atoms associated with f:

$$f^{\mathcal{I}} \subseteq \mathcal{D}^{\mathcal{I}}_f \in \mathcal{D}^{\mathcal{I}}, \qquad \text{for all } f \in \mathcal{F}_{Sig}$$

If f is an attribute name, i.e., $f \in \mathcal{F}_{Attr}$ with $\textit{type}(f) = (s_1, \ldots, s_k)$, then the interpretation $f^{\mathcal{I}}$ of f is given by a relation respecting the type of f, i.e.,

$$f^{\mathcal{I}} \subseteq (\mathcal{D}^{\mathcal{I}}_{s_1} \times \ldots \times \mathcal{D}^{\mathcal{I}}_{s_k}) \in \mathcal{D}^{\mathcal{I}}, \qquad \text{for all } f \in \mathcal{F}_{Attr} \text{ with } \textit{type}(f) = (s_1, \ldots, s_k)$$

Definition of interpretation $\mathcal{P}^{\mathcal{I}}$ of predicate symbols The interpretation of the predicate symbols $\mathcal{P} = \{$`in`$\}$ is also fixed. The predicate symbol `in` is interpreted as subset check, i.e.,

$$\texttt{in}^{\mathcal{I}} : \mathcal{D}^{\mathcal{I}} \times \mathcal{D}^{\mathcal{I}} \to \mathbb{B}$$

such that for all $d_1, d_2 \in \mathcal{D}^{\mathcal{I}}$

$$\texttt{in}^{\mathcal{I}}(d_1, d_2) \quad = \quad \begin{cases} \textit{true}, & \text{if } d_1 \subseteq d_2 \\ \textit{false}, & \text{otherwise} \end{cases}$$

Definition of a satisfaction relation

Just like in Predicate Logic, the satisfaction relation $\models$ is intended to relate an instance $\mathcal{I}$ and a formula φ as defined above

$$\mathcal{I} \models \varphi$$

iff φ holds in $\mathcal{I}$. Let $\mathcal{V}$ denote the variables in φ. The definition of the satisfaction relation is similar to that for Predicate Logic in Section 4.2.

$$\mathcal{I} \models \varphi \;\text{ iff }\; \mathcal{I}, \emptyset \models \varphi$$

where $\mathcal{I}, l \models \varphi$ for mappings (we also called them environments) $l : \mathcal{V} \to \mathcal{D}^{\mathcal{I}}$ is defined inductively by

$$\begin{array}{ll}
\mathcal{I}, l \models p(e_1, \ldots, e_n) & \text{iff } p^{\mathcal{I}}(evalE_l^{\mathcal{I}}(e_1), \ldots, evalE_l^{\mathcal{I}}(e_n)) = true \quad \text{for all } p \in \mathcal{P} \\
\mathcal{I}, l \models \texttt{!}\,\varphi & \text{iff it is not the case that } \mathcal{I}, l \models \varphi \\
\mathcal{I}, l \models \varphi \;\texttt{\&\&}\; \psi & \text{iff } \mathcal{I}, l \models \varphi \text{ and } \mathcal{I}, l \models \psi \\
\mathcal{I}, l \models \texttt{all}\ x\texttt{:}\,e \texttt{|}\,\varphi & \text{iff } \mathcal{I}, l[x \mapsto d] \models \varphi \text{ for all } d \in evalE_l^{\mathcal{I}}(e)
\end{array}$$

In the above, $evalE_l^{\mathcal{I}} : expr \to \mathcal{D}^{\mathcal{I}}$ is the evaluation function for expressions:

$$\begin{array}{rcl}
evalE_l^{\mathcal{I}}(var) & = & l(var) \;\text{ for all } var \in \mathcal{V} \\
evalE_l^{\mathcal{I}}(name) & = & name^{\mathcal{I}} \;\text{ for all } name \in \mathcal{F}_{Name} \\
evalE_l^{\mathcal{I}}(\texttt{op}(e_1, ..., e_n)) & = & \texttt{op}^{\mathcal{I}}(evalE_l^{\mathcal{I}}(e_1), \ldots, evalE_l^{\mathcal{I}}(e_n)) \;\text{ for all } \texttt{op} \in \mathcal{F}_{Op}
\end{array}$$

Definition of an evaluation function

We can also define an evaluation function for formulas. More precisely, given an instance $\mathcal{I}$ we define a function

$$evalF^{\mathcal{I}} : Formula \to \mathbb{B}$$

such that

$$evalF^{\mathcal{I}}(\varphi) = evalF_{\emptyset}^{\mathcal{I}}(\varphi)$$

where for mappings $l : \mathcal{V} \to \mathcal{D}^{\mathcal{I}}$, $evalF_l^{\mathcal{I}}(\varphi)$ is defined by

$$\begin{array}{rcl}
evalF_l^{\mathcal{I}}(p(e_1, \ldots, e_n)) & = & p^{\mathcal{I}}(evalE_l^{\mathcal{I}}(e_1), \ldots, evalE_l^{\mathcal{I}}(e_n)) = true \;\text{ for all } p \in \mathcal{P} \\
evalF_l^{\mathcal{I}}(!\varphi) & = & \neg\, evalF_l^{\mathcal{I}}(\varphi) \\
evalF_l^{\mathcal{I}}(\varphi_1 \;\texttt{\&\&}\; \varphi_2) & = & evalF_l^{\mathcal{I}}(\varphi_1) \wedge evalF_l^{\mathcal{I}}(\varphi_2) \\
evalF_l^{\mathcal{I}}(\texttt{all}\ x : e \texttt{|}\,\varphi) & = & evalF_{l[|x \mapsto d]}^{\mathcal{I}}(\varphi) \text{ for all } d \in evalE_l^{\mathcal{I}}(e)
\end{array}$$

11.4 Using Alloy to specify operations (Weeks 5,6)

Predicates can be used to formalize the effect of an operation by capturing its pre-conditions and post-conditions. Consider, for instance, the Alloy specification of a stack in Figure 11.2. The signatures define a stack to have the following structure: a stack has an attribute *head* that is either null or points to a node. A node has an attribute *val* that carries exactly one value and an attribute *succ* that is either null (node has no successor and thus is the 'bottom' of the stack)

```
1  module Stacks
2
3  sig Value {}
4
5  sig Node {
6      val : Value
7      succ : lone Node
8  }
9
10 sig Stack {
11     head : lone Node
12 }
13
14 pred isEmpty[s : Stack] {
15     no s.head
16 }
17
18 fact StackFacts {
19    all s : Stack | no n : (s.head).*succ | n in n.^succ
20 }
21
22 pred push[before : Stack, v : Value, after : Stack] {
23    after.head.val = v &&
24    after.head.succ = before.head
25 }
26
27 pred top[s : Stack, v : Value] {
28    !isEmpty[s]
29    v = s.head.val
30 }
31
32 pred pop[before, after : Stack, v : Value] {
33    !isEmpty[before] &&
34    after.head = before.head.succ &&
35    v = before.head.val &&
36 }
37
38 assert NeverEmptyAfterPush {
39    all s1, s2 : Stack | all v : Value | push[s1, v, s2] => !isEmpty[s2]
40 }
41
42 assert PopAfterPush {
43    all s1, s2 : Stack | all v : Value | push[s1, v, s2] => pop[s2, s1, v]
44 }
45
46 pred Show {
47    some s1, s2, s3 : Stack | some v1, v2 : Value | push[s1, v1, s2] &&
48                                                   push[s2, v2, s3]
49 }
```

Figure 11.2: Textual constraints of a stack

or points to another node (the node below this node in the stack). The fact *StackFacts* ensures the successor relation *succ : Node → Node* does not contain any cycles.

Predicates *push*, *top*, and *pop* define the usual stack operations. Each of these operations have parameters and the effect of each of these operations is captured in terms of Alloy constraints that use these parameters. Take *top[s : Stack, v : Value]*, for instance. Given an Alloy instance containing stack *s1* and a value *v1*, *top[s1, v1]* will hold precisely when *!isEmpty[s1]* and *v1 = s1.head.val*, i.e., when the given parameters make the constraints true. The key thing to remember when specifying operations in Alloy is that the predicates that make up the defintion of the operation do not bind values to any of these parameters; all parameters need to be bound to values already and the predicate just checks that the supplied parameters satisfy the constraints in the predicate.

The situation is similar for specification of *push* and *pop*. Given an instance with two stacks *s1* and *s2* and a value *v*, *push[s1, v, s2]* will hold precisely when *s1*, *s2*, and *v* satisfy the constraints in the body of *push*, i.e., when *s2* has a head with datum *v* and the successor of the head of *s2* is equal to the head of *s1*. So, unlike in imperative programming, *push* does not assign a new value to its third parameter — it just checks that all given parameters satisfy the given constraints. Note that, in terms of imperative programming, these constraints can be thought of as pre-conditions (e.g., *!isEmpty[before]*) and post-conditions (e.g., *after.head.val = v*).

The assertions express, respectively, that a stack is never empty after a push and that a pop immediately after a push yields the original stack and the pushed data object. Both of these assertions should hold.

Apart from assertion checking, Alloy can now also be used to find instances satisfying the operation constraints i.e., instances that illustrate possible applications of the stack operations:

```
stackScenario1: run {some s1, s2 : Stack, some v : Value | push[s1,v,s2]}
```

These illustrations can be made more interesting via additional constraints on any of the parameters:

```
stackScenario2: run {some s1, s2 : Stack, some v : Value | !isEmpty[s1] &&
                                                           push[s1,v,s2]}
```

Note that constraints on parameter representing the resulting stack effectively allow computations to be run "backwards" by asking how the input would have to look like such that the output satisfies certain conditions:

```
stackScenario3: run {some s1, s2, s3 : Stack, some v1, v2 : Value |
                     pop[s1,v1,s2] && pop[s2,v2,s3] && !isEmpty[s3] && v1!=v2}
```

As another example, consider the 'River Crossing' puzzle, in which a farmer has to bring a cabbage, a goat, and a wolf from one river shore to the other river shore using a boat that seats only him and one other object while avoiding that (1) the goat and the cabbage, and (2) the wolf and the goat are ever without him together on one shore (because without the farmer present the first object will eat the second). Devise a sequence of river crossings that allow the farmer bring all objects safely from one shore to the other.

```
1  // farmer and his possessions are objects
2  abstract sig Object {
3     eats: set Object
4  }
5  one sig Farmer, Fox, Chicken, Grain extends Object {}
6
7  // defines what eats what
8  fact {
9     eats = Fox->Chicken + Chicken->Grain
10 }
11
12 // at most one item to move from 'from' to 'to'
13 pred crossRiver [from, from', to, to': set Object] {
14   some x: from | {
15     from' = from - x - Farmer - from'.eats &&
16     to' = to + x + Farmer
17   }
18 }
```

Figure 11.3: Part of river crossing model from Alloy online tutorial

The Alloy tutorial[2] contains a model of this problem. The part that describes a river-crossing operation is reproduced in Figure 11.3. The operation *crossRiver* takes four parameters. Parameters *from* and *to* denote the set of objects on each of the two shores before the crossing. Similarly, parameters *from'* and *to'* denote the set of objects on each of the two shores after the crossing.

Alloy can now be used to find instances satisfying the constraints in *crossRiver*, i.e., instances that illustrate possible river crossing scenarios:

```
crossRiverScenario1: run {some from', to' : set Object |
                                  crossRiver[Object, from', none, to']}
```

As expected, Alloy will find a total of four satisfying instances (one for the farmer taking nothing and three more for the farmer taking one of the three objects) in response to the run command above.

However, to make our formalization truly useful, we need to be able to ask Alloy to find *sequences* of *crossRiver* operations connecting a given initial state with a given final state.

11.5 Using Alloy to specify sequences of operations (Week 6)

To express constraints on sequences of operations (and let Alloy find sequences of operation applications that satisfy certain constraints), Alloy's built-in library is useful. Alloy's `util` library offers a parametric (generic) signature `ordering` which allows elements of a signature to be put into a total linear order. Let S be some signature. The line

```
open util/ordering[S]
```

[2]Available at `http://alloytools.org/tutorials/online/`

then adds constraints to your Alloy specification that forces all elements of *S* into a total linear order, i.e., into a structure very similar to a linked list. A number of functions and predicates are then available on the elements in *S*. For instance, in every instance of the specification, *S* either has no elements, or

- there is a unique first element in S (denoted with key word *first*),
- there is a unique last element in S (denoted with key word *last*),
- every element *s* in *S*, except *last*, has a unique successor (denoted with *s.next*).

Also, there are predicates to compare two elements *s1* and *s2* in *S*:

- *lt[s1,s2]* holds if *s1* is less than *s2* in the order,
- *lte[s1,s2]* holds if *s1* is less than or equal to *s2*,
- *gt[s1,s2]* holds if *s1* is greater than *s2*, and
- *gte[s1,s2]* holds if *s1* is greater than or equal to *s2*.

Coming back to our stack example, if we now impose such a total linear ordering on signature *Stack* and define the successor of a given stack as the result of either the *push*, *top*, or *pop* operation (see Figure 11.4), the satisfying instances produced by Alloy will then contain sequences of stacks that are the result of applying a sequence of these operations to some initial stack.

The command below creates a sequence of 7 stacks each created from the previous by one of the three operations and starting and ending with the empty stack:

```
stackScenario4 : run {isEmpty[last]} for 7
```

Again, arbitrary constraints can be put on any of the stacks in the sequence. For instance, the following run command prompts a search for a sequence of length 8 such that (1) the first two elements of the first stack have some values *v1* and *v2* in this order and (2) the first two elements of the last stack also hold these values, but in reverse order:

```
stackScenario5 : run {some disj v1, v2 : Value |
                      first.head.val=v1 && first.head.succ.val=v2 &&
                      last.head.val=v2 && last.head.succ.val=v1} for 8
```

The exact same technique can now be used to find solutions to the the River Crossing puzzle. Since the *crossRiver* operation changes both the objects on the near and the far shore and both shores are relevant for the description of the desired final state, we wrap these two as attributes into an element of a signature *State* whose elements are totally ordered and whose attribute values change according to the *crossRiver* operation.

Now,

```
solvePuzzle: run {first.near=Object && first.far=none && last.far=Object} for 8
```

```
module StacksAndSequencesOfOperations

open util/ordering[Stack]

sig Value {}

sig Node {
val : Value,
succ : lone Node
}
sig Stack {
head : lone Node
}
pred isEmpty[s : Stack] {
no s.head
}
pred push[s1 : Stack, v : Value, s2 : Stack] {
   // as above
}
pred pop[s1 : Stack, s2 : Stack, v : Value] {
   // as above
}
fact StackFacts {
  all s : Stack | no n : (s.head).*succ | n in n.^succ  // no cycles
  isEmpty[first]  // we're always starting with the empty stack
  all s1,s2 : Stack | s1.next=s2 =>
                        (some v : Value |
                          push[s1,v,s2] || (top[s1,v]&&s1=s2) || pop[s1,s2,v])
}
```

Figure 11.4: Alloy specification allowing the creation of sequences of operations

```
open util/ordering[State]   // impose an ordering on states

abstract sig Object {
   eats: set Object
}
one sig Farmer, Fox, Chicken, Grain extends Object {}

sig State {
    near, far: set Object    // use states to record who is where
}
fact {
   eats = Fox->Chicken + Chicken->Grain
}
pred crossRiver[from, from', to, to': set Object] {
  some x: from | {
    from' = from - x - Farmer - from'.eats &&
    to' = to + x + Farmer
  }
}
fact {  // operation crossRiver transitions allows between states
  all s: State, s': s.next {
    (Farmer in s.near => crossRiver[s.near, s'.near, s.far, s'.far]) &&
    (Farmer !in s.near => crossRiver[s.far, s'.far, s.near, s'.near])
  }
}
```

Figure 11.5: Complete river crossing model from Alloy online tutorial

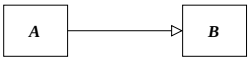
will find the shortest solutions to the puzzle (i.e., a scope of at least 8 is required). Note how the run command describes both the initial and final states (*first.far=none* is necessary because we have not said that an object always is at exactly one shore.

11.6 Example: A simple file system

We now return to the file system example used in Section 10.1. We will use this example to introduce the graphical representation of Alloy specifications. This example is taken from [Jac06b].

11.6.1 Sets and subsets

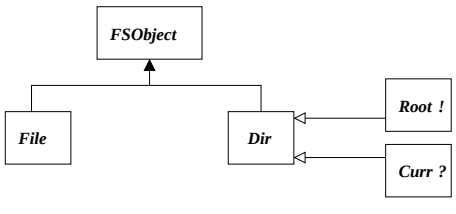
Each box in a graphical Alloy specification denotes a set of objects. If the specification describes a computer system to be developed, each box typically is implemented by a class. In that case, the set of objects denoted by the box coincides with the objects of that class and we will use the terms "set" and "class" synonymously. That is, when a box labeled A in a specification is referred to as class A, we really mean the set of objects implemented by class A. In the graphical specification, an arrow with a closed head from a box labeled A to a box labeled B, denotes set inclusion: every object of set A also is contained in set B. A is a subset of B. If the sets are implemented with two classes A and B, A is a subclass of B.




```
sig B {}
sig A extends B {}
```

The arrowhead indicates whether the subset contains all elements of its superset (the arrowhead is filled) or just some elements of the superset (the arrowhead is unfilled). Subsets can share an arrow; in this case, they are mutually disjoint, and the arrowhead indicates whether or not their union exhausts the superset. Subsets that don't share an arrow are not necessarily disjoint. The cardinality of a set can be constrained. This can be done by writing the cardinality of the set within the corresponding box. For example, a "!" means that the set has exactly one object. A "?" expresses that the set contains at most one object.

Consider, for example, the figure below showing a class model of a part of a file system. In contrast to the code in Section 10.1, this file system will allow directories to contain not only other directories but also files.




```
sig FSObject {}
sig Name {}
sig File extends Object {}
sig Dir extends Object {}
fact FileDirPartition {
        no File & Dir
        File + Dir = Object
}
sig Root extends Dir {
}{      one Root
}
sig Cur extends Dir {
}{      lone Cur
}
```

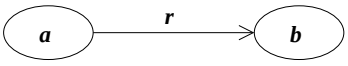
The box labeled *File* represents the set of all objects of a class called *File*. The class *Dir* has two subclasses *Root* and *Curr*. The subclass *Root* contains exactly one object representing the root directory. Classes with exactly one object are called *singletons*. The subclass *Curr* contains at most one object representing the current directory. Note that this means that there may be no current directory. Also, since *Root* and *Curr* do not share an arrow, they need not be disjoint. Consequently, the root directory may also be the current directory. An *FSObject* can either be a file or a directory. These two subclasses are disjoint and exhaust their superclass.

11.6.2 Relations

Depending on whether we consider a single state or a set of states, relations either connect objects or classes.

Between objects

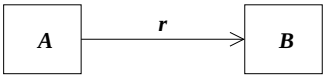
In Section 10.1 we have already seen how a field of an object gives rise to a relation and how that relation is represented graphically. More precisely, the field arrows in state diagrams express a relation between the object containing the field and the object contained in the field. The name of the field is the name of the relation. For instance, consider again the following state diagram



The field r relates the object a to object b, that is, we have a relation r that connects a to b, or, more formally, $(a, b) \in r$.

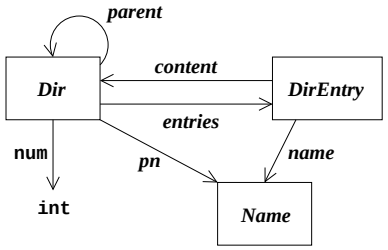
Between classes

Suppose that in every execution state, every object of class A contains a field r whose value is an object of class B. This relationship between the classes is expressed by a relation r that relates A to B, that is, $(A, B) \in r$. Relations between classes are described in the specification. Graphically,



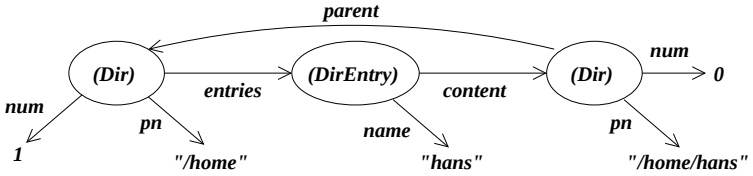

```
sig B {}
sig A {
      r : set B
}
```

While a state diagram describes a single state, a class model thus describes a set of states. More precisely, the class model specifies a set of states by imposing the constraint that each object in the state belong to one of the boxes, and that the field arrow in the state diagram connect objects from the appropriate boxes. If part of the requirements specification, a class model thus summarizes the set of permissible states. Consider, for instance, the specification

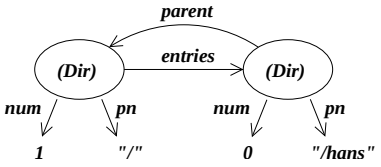



```
sig Name {}
sig Dir {
      num : set int,
      parent : set Dir,
      entries : set DirEntry,
      pn : set Name
}
sig DirEntry {
      content : set Dir,
      name : Name
}
```

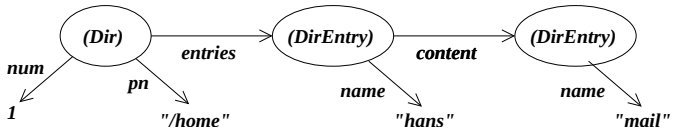
The arrow labeled *pn* from *Dir* to *Name* expresses that every object of the *Dir* class has a field called *pn* whose value is an object of class *Name*. Note that Alloy does not contain any build-in datatypes, that is, the *Name* class must be declared as a domain like *Dir* and *DirEntry*. For instance, this specification allows this state



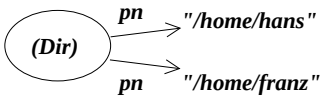
but not this state



or this state



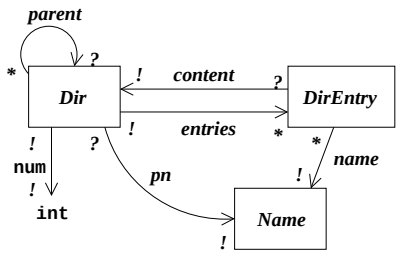
Note that the specification above does not place constraints on the relative number of objects and thus also allows this state



which the code in Section 10.1 clearly rules out (*pn* would have to be an array or vector for this state to be feasible).

11.6.3 Multiplicities

So far, the specification does not allow us express that, for instance, a directory object may contain several entries or that every entry belongs to exactly one directory. Let's add multiplicities to our file system specification.




```
sig Name {
}{    lone this.~pn
}
sig Dir {
      num : int,
      parent : lone Dir,
      entries : set DirEntry,
      pn : Name
}{    lone this.~content
}
sig DirEntry {
      content : Dir,
      name : Name
}{    one this.~entries
}
```

The meaning of the multiplicities at the target end of the relations is as follows:

- From *Dir* to *DirEntry*: The $*$ at the target end of the *entries* arrow tells us that in any legal state, there may be zero or more *DirEntry* objects associated with each *Dir* object, that is, a directory can contain zero or more directories. Note that the multiplicity $*$ is chosen by default.

- From *Dir* to *Name*: The ! at the target end of the *pn* arrow tells us that in any legal state, there is exactly one *Name* object associated with each *Dir* object, that is, every directory has exactly one path name. Note that this implies that the *pn* field in a *Dir* object can never be null. Similarly for the *name* relation.

- From *Dir* to *Dir*: The ? at the target end of the *parent* arrow tells us that in any legal state, there is zero or one *Dir* object associated with each *Dir* object. Thus, the *parent* field in a *Dir* object may be null.

The meaning of the multiplicities at the source end of the relations is as follows:

- From *Dir* to *DirEntry*: The ! at the source end of the *entries* arrow tells us that in any legal state, every *DirEntry* object is contained in exactly one directory. Consequently, directory entries cannot be shared by several directories, that is, different directory objects must have different elements in *entries*.

- From *Dir* to *Name*: The ? at the source end of the *pn* arrow means that a *Name* object will represent the name of at most one *Dir* object. Consequently, a *Name* object can represent the path name of at most one directory. In other words, path names cannot be shared.

- From *Dir* to *Dir*: The $*$ at the source end of the *parent* arrow tells us that in any legal state, a directory can be the parent of zero or more directories.

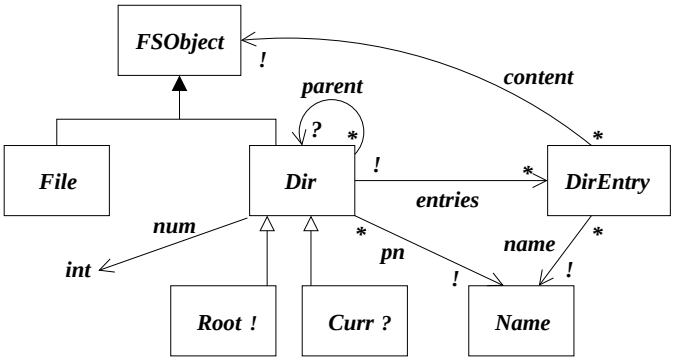


Figure 11.6: Graphical constraints of a simple file system in Alloy [Jac06b]

- From *DirEntry* to *Name*: The $*$ at the source end of the *name* arrow tells us that in any legal state, each *Name* object may be the name of zero or more *DirEntry* objects, that is, different directory entries can share the same name (for instance, many user directories contain a directory with the name "mail").

Multiplicity constraints may or may not be enforced by code. For instance, if the code allows the name field of a directory entry to be set to null, the multiplicity constraint on the name relation would be violated. To enforce this constraint in code, we could, for instance,

- declare the *name* field as private. This would ensure that the name field is only set by the *DirEntry* constructor and never by code outside the *DirEntry* class.
- ensure that the *DirEntry* constructor never sets *name* to null.

Note, however, that not every constraint in the specification can be enforced in the code. Consider, for instance, the non-sharing requirement of directory entries, that is, every directory entry must belong to exactly one directory (the ! at the source end of the *entries* relation). Assuming the code in Section 10.1, this constraint cannot easily be enforced, because we cannot prevent the user from creating two directories that share an entry:

```
Dir d1 = new Dir(null, "d1");
Dir d2 = new Dir(null, "d2");
DirEntry e = new DirEntry(null, "e");
d1.add(e);
d2.add(e)
```

To guarantee this constraint, we could, for instance, make entries cloneable and modify the `add` method such that it always insert clones of the argument entries.

Figure 11.6 contains the graphical constraints of the file system.

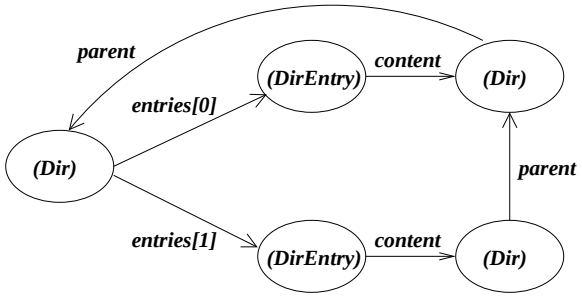
11.6.4 Invariants

We will now list constraints on our simple file system that cannot be expressed graphically.

The parent of a directory is the directory that contains an entry for it:

```
sig Dir extends FSObject {...}{
    ...
    // A directory's parent is the directory that has an entry for it
    parent = this.~@content.~@entries
    // or parent = {d : Dir | this in (d.@entries).@content}
    ...
}
```

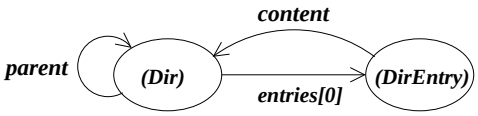
This constraint rules out, for instance, the following state which is allowed by the specification and also the code presented in Section 10.1.



The next constraint ensures that every directory except the root has exactly one parent.

```
fact OneParent {
        // all directories besides root have one parent
        all d: Dir - Root | one d.parent
}
```

So far, there is nothing in our specification that prevents a directory from containing itself:



Thus, we want to express that the root directory is an ancestor of every directory except itself and that there are no cycles in an ancestor chain.

```
sig Dir extends FSObject {...}{
    ...
    // A directory is not its own ancestor and every directory
    // but the root has the root as ancestor
    this !in this.^@parent
    this != Root => Root in this.^@parent
    ...
}
```

This constraint implies that every directory in the file system either is the root directory or is accessible from the root. A similar constraint limits the files in the system:

```
sig File extends FSObject {...}{
    ...
    // Every file has an entry in some directory
    some d : Dir | this in (d.@entries).@content
    ...
}
```

Together, the two constraints imply that every file system object is accessible from the root. For a user this means that files not reachable from the root cannot be used. For the implementor this means that, for instance, unreachable objects can be garbage collected.

Here's another constraint that ensures that directory entries are uniquely identified by their name.

```
sig Dir extends FSObject {...}{
    ...
    // A directory contains at most one entry with a given name
    all e1, e2 : entries | e1.name = e2.name => e1 = e2
    ...
}
```

The textual constraints of the final Alloy specification of a simple file system are summarized in Figure 11.7.

The constraints that we've written do not contain everything that is true about the system. Some facts are implied by the constraints. For instance, the fact that the root has no ancestor is implied by the above constraints.

11.7 Example: Family tree

Our running example of a file system models a computer system. In this example, an Alloy specification is used to describe family relationships (Figure 11.8). The set *Person* has three subsets: *Married*, *Man*, and *Woman*. The sets *Man* and *Woman* are partition *Person*: Every person is either a man or a woman. A person may be associated with other persons through the *parents* and *siblings* relations. Every person has exactly one name. Every man has at most one wife and every woman is the wife of at most one man. Every woman has at most one husband and every man is the husband of at most one woman.

To obtain an adequate specification of family trees, the graphical constraints in Figure 11.8 need to be complemented with the textual constraints in Figure 11.9.

11.8 Example: Air traffic control

In this example an Alloy specification is used to describe an organizational structure. It describes how airspace is divided into sectors, and how controllers are associated with aircraft.

```
1
2         module FileSystem  // model of a simple file system
3         abstract sig FSObject {}
4         sig Name {
5         }{  lone this.~pn // path names are not shared
6         }
7         sig File extends FSObject {
8             fn : Name
9         }{  some this.~content  // every file is contained in some entry
10        }
11        sig Dir extends FSObject {
12            entries : set DirEntry,
13            parent : lone Dir,
14            pn : Name
15        }{  // parent is the dir that contains this dir as entry
16            parent = this.~@content.~@entries
17            // equivalently, parent = {d : Dir | this in (d.@entries).@content}
18            all e1, e2 : entries | e1.name = e2.name => e1 = e2
19            this !in this.^@parent  // dirs don't contain themselves
20            // root is ancestor of every dir, except root
21            this != Root => Root in this.^@parent
22            lone this.~content // dirs are not shared
23        }
24        sig DirEntry {
25            name: Name,
26            content : FSObject
27        }{  one this.~entries // every entry belongs to one directory
28        }
29        sig Root extends Dir {
30        }{  one Root
31            no parent
32        }
33        sig Cur extends Dir {
34        }{  lone Cur
35        }
36        fact FileDirPartition {
37            no File & Dir    // not necessary b/c subsignatures always disjoint
38            File + Dir = FSObject
39        }
40        fact OneParent {
41            // all directories besides root have one parent
42            all d: Dir - Root | one d.parent
43        }
44        fun NonTrivial () {
45            some DirEntry
46        }
47        assert NoDirAliases {
48            // Directories are the content of at most one directory entry. Valid!
49            all o: Dir | lone o.~content
50        }
51        assert NoFileAliases {
52            // Files are the content of at most one directory entry. Invalid
53            // I.e., files may have 'symbolic links'
54            all f: File | lone f.~content
55        }
56        assert OneRoot {
57            lone Root  // there is at most one root. Valid!
58        }
59        run NonTrivial for 3
60        // check NoDirAliases for 3
61        // check NoFileAliases for 3
62        // check OneRoot for 3
63
```

Figure 11.7: Textual constraints of a simple file system in Alloy [Jac06b]

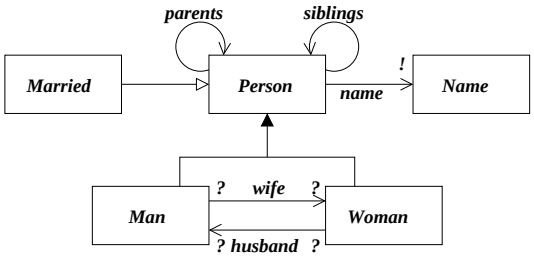
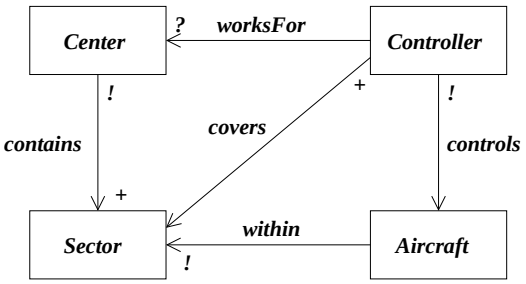


Figure 11.8: Graphical constraints for family tree in Alloy



The specification shows that there are centers, each of which contains one or more sectors. Air traffic controllers work for centers and cover sectors. An aircraft is controlled by a controller and flies within a sector. A controller works for at most one center. A center contains at least one sector and each sector is contained by exactly one center. An aircraft is always exactly in one sector, that is, sectors don't overlap or have gaps between them. Each aircraft is controlled by exactly one controller. A sector is covered by at least one controller.

Invariants

Our graphical specification does not express that

- if an aircraft is flying in a sector, then it is being controlled by a controller that covers that sector.
- controllers only cover sectors contained by centers they work for.

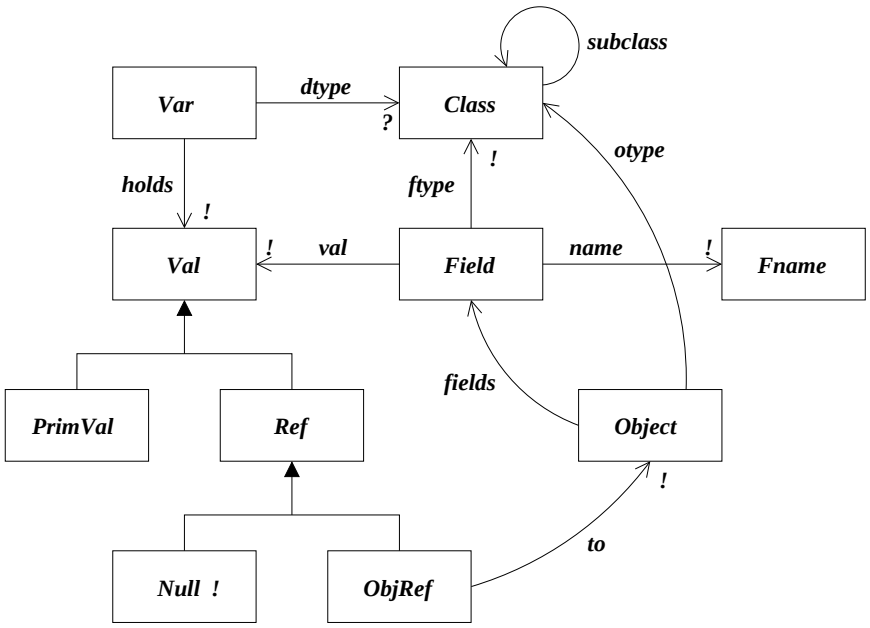
Appropriate textual constraints would have to be formulated and added to the specification.

11.9 Example: Java

This example shows how an Alloy specification can be used to describe how variables, references and objects are related in Java programs. The specification could thus form a useful part of a Java language description.

```
1  // Partial model of a family tree
2  module Family
3
4  sig Name {}
5
6  sig Person {
7      parents : set Person,
8      siblings : set Person,
9      name : Name
10 }
11
12 sig Man extends Person {
13     wife : lone Woman
14 }
15
16 sig Woman extends Person {
17     husband : lone Man
18 }
19
20 fact Partition {
21     Man + Woman = Person
22 }
23
24 fact ParentsBasics {
25     all p : Person | lone (p.parents & Man) && lone (p.parents & Woman)
26     no p : Person | p in p.^parents
27 }
28
29 fact WifeIsInverseOfHusband {
30     wife = ~husband
31 }
32
33 fact DefSiblings {
34     all p, q : Person | q in p.siblings <=> (p.parents = q.parents)
35 }
36
37 sig Married extends Person {}
38
39 fact MarriedBasics {
40     Married = Man.wife + Woman.husband
41     all p : Person | some p.wife iff p in (Man & Married)
42     all p : Person | some p.husband iff p in (Woman & Married)
43 }
44
45 pred Marry[wife, wife': Man - > Woman, m : Man, w : Woman] {
46     m not in Married && w not in Married
47     m.wife' = w
48     all p : Man | p != m => p.wife' = p.wife
49 }
50
51 fun Show () {
52     some Man
53     some Woman
54 }
55
56 run Show for 4
57 // run Marry for 4
```

Figure 11.9: Textual constraints for family tree in Alloy



Java programs have variables, which have declared types that are classes. Classes are related by subclassing. Variables hold values, divided into primitive values and references. References are further divided into the null reference and references to objects. An object has fields, each of which has a name and a value. Objects have types, too. Every variable and every field have a value (although it may be null). Every object reference points to an object. Note that these are fundamental properties of a safe programming language.

The declared type of a variable cannot change. The type of an object cannot change either. Objects cannot migrate between classes. An object cannot change what fields it has, and the name of a field cannot change.

Invariants

The fundamental property guaranteed by the Java type system can be expressed as a textual constraint: If a variable or field holds a reference to an object, the type of the object is the same class or subclass of the declared type of the variable or field.

11.10 Analyzing Alloy specifications (Weeks 5,6)

We now discuss in more detail how the Alloy Constraint Analyzer works. Given a specification S, we can use the Alloy Constraint Analyzer for the animation of S, checking consistency of S, and checking assertions of S. Each task can be reduced to a satisfiability problem. Let the constraints expressed in S be given

by the predicate logic formula φ_S and let φ_P be the formula representing the run predicate

- Animation: The analyzer shows us a state that satisfies $\varphi_S \wedge \varphi_P$, that is, it shows us an instance that makes $\varphi_S \wedge \varphi_P$ true.
- Checking consistency: The analyzer tells us whether or not $\varphi_S \wedge \varphi_P$ is consistent, that is, satisfiable.
- Checking assertions: Let φ_A be an assertion and The analyzer tells us whether φ_A is true, whenever φ_M is, that is, whether $\varphi_S \rightarrow \varphi_A$ is valid. $\varphi_S \rightarrow \varphi_A$ is valid if and only if $\varphi_S \wedge \neg \varphi_A$ is unsatisfiable. If $\varphi_S \wedge \neg \varphi_A$ is true in some state s, then $\varphi_S \rightarrow \varphi_A$ is not valid and s is a counter example. More formally,

$$\begin{array}{ccc} \varphi_S \rightarrow \varphi_A & \textit{valid} & \textit{iff} \\ \neg\,(\varphi_S \wedge \neg\,\varphi_A) & \textit{valid} & \textit{iff} \\ \varphi_S \wedge \neg\,\varphi_A & \textit{unsatisfiable} & \end{array}$$

Unfortunately, it turns out that the satisfiability problem for Alloy formulas is undecidable. To see this, remember that validity for first-order predicate logic is undecidable.

Theorem 7. (Validity in first-order predicate logic is undecidable) The validity problem for first-order predicate logic is undecidable.

The undecidability of validity directly implies the undecidability of satisfiability.

Theorem 8. (Satisfiability in first-order predicate logic is undecidable) The satisfiability problem for first-order predicate logic is undecidable.

Proof: By contradiction. Suppose there was a procedure P that, when given a first-order predicate logic formula f, always terminates and outputs "Yes", if f is satisfiable and "No" otherwise.

$$P(\varphi) \quad = \quad \begin{cases} \textit{"Yes"}, & \textit{if } f \textit{ satisfiable} \\ \textit{"No"}, & \textit{otherwise} \end{cases}$$

Using P, we can construct a procedure P' as follows.

$$P'(f) \quad = \quad \begin{cases} \textit{"Yes"}, & \textit{if } P(\neg\, f) = \textit{"No"} \\ \textit{"No"}, & \textit{otherwise} \end{cases}$$

Figure 11.10 illustrates the construction. Remember that f is a theorem if and only if $\neg\, f$ is not satisfiable. So, P' decides the validity problem. Contradiction to Theorem 8. ■

Since Alloy contains quantifiers it is as expressive as first-order predicate logic. Therefore, satisfiability in Alloy is also undecidable.

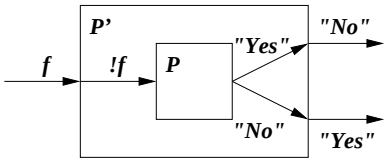


Figure 11.10: Validity versus satisfiability

Corollary 11.10.1. (Satisfiability in Alloy is undecidable)
The satisfiability problem for Alloy is undecidable. ■

The above result means that satisfiability in Alloy cannot be automated. For every procedure P for finding satisfying states there will always exist a formula φ such that the execution of P on φ either does not terminate, produces the wrong output, or requires user interaction. If we implement P such that it always terminates with the correct result, but requires user interaction, P is essentially a theorem prover. However, we want our analysis tools to hide the details and complexities of the underlying logic and analysis techniques from the user. To this end, it is imperative that our tools require as little user interaction as possible. So, rather than accepting user interaction and analyzing Alloy formulas by means of a theorem prover, we break the impasse by trading expressiveness of the analysis for automation. We do this by limiting the analysis to a finite subset of the entire state space.

11.10.1 Bounded satisfiability

The Alloy Constraint Analyzer achieves full automation despite the expressiveness of the Alloy logic and its resulting undecidability by explicitly keeping the search space finite. Whenever we ask the checker to find a satisfying instance, we have to define the scope of the search. We call the problem of deciding whether a formula has a satisfying instance within a given scope the *bounded satisfiability problem*. While general satisfiability is not decidable for Alloy, bounded satisfiability is. In a finite scope, every Alloy formula φ has only finitely many candidate instances that could make φ true. This means that the bounded satisfiability problem for Alloy is decidable.

Theorem 9. (Decidability of bounded satisfiability problem)
Given an Alloy formula φ and a scope $n \in \mathbb{N}$, the problem of deciding whether φ has satisfying states in scope n is decidable.

Proof idea: The number of possible instances for φ is finite. Thus, we can construct a procedure P that exhaustively generates all instances and for each instance checks whether it makes φ true. P will always eventually terminate with the correct answer. ■

Essentially, the Alloy checker is built using the brute force method described in the above proof. First, it translates the specification S into one large first-

order predicate logic formula φ_S. Then, φ_S is passed to an off-the-shelf satisfiability solver for predicate logic called *Satlab*. If Satlab finds a satisfying instance for φ_S, it is output by the Alloy checker. Otherwise, Alloy answers "No solution found".

11.10.2 Interpreting Alloy analysis results

To achieve decidability and thus full automation, the Alloy checker thus sacrifices informativeness of the analysis. Instead of answering the question

Q1: "Is this formula satisfiable?"

it answers the question

Q2: "Is this formula satisfiable in scope n?"

where n is some natural number. A positive answer of the Alloy checker to *Q2* provides as much information as a positive answer to the question *Q1*. However, a negative answer does not. If the checker says that specifcation φ_S is not satisfiable in scope n, then φ_S could be satisfiable in some larger scope, or it could not be satisfiable at all.

Theorem 10. (Bounded satisfiability I)
If φ satisfiable in scope n, then φ satisfiable in general. If φ not satisfiable in scope n, then φ may or may not be satisfiable in general. ■

Since the scope is an *upper bound* on the size of the underlying sets, the states of a specification S in scopes $n' < n$ are a subset of the states of S in scope n. This has the consequence that if assertion A is valid in S in scope n, then A also is valid in S in all scopes $n' < n$. In other words, if no state of S in scope n was a counter example for A, then no state of S in n' can be a counter example either.

However, A may or may not be valid in S in scopes larger than n. Intuitively, this is because a counter example for A may require a scope larger than n.

Theorem 11. (Bounded satisfiability II)
If assertion φ_A does not have a counter example in scope n,

- then it does not have a counter example in scopes less than n.
- then it may or may not have a counter example in scopes greater than n.

■

11.10.3 Soundness and completeness of Alloy's analysis

In Part II, we saw how formal proofs can be used to reason about formulas and derive new information about them. We saw that it was important to know that

- what has been derived with a proof does indeed follow semantically (soundness), and
- what follows semantically can be also be derived with a proof (completeness).

Alloy's analysis allows us to reason about Alloy specifications. Is this analysis sound and complete? Since we have a formal semantics for Alloy, we can formalize these notions.

Soundness

Alloy's consistency analysis can be said to be sound iff for every Alloy specification *Spec* with run predicate P, the instance $\mathcal{I}$ produced by Alloy's analysis in response to the command

```
run P() for n
```

for some scope n does indeed make all the constraints in *Spec* and P true, i.e.,

$$\mathcal{I} \models \varphi_{Spec} \wedge \varphi_P$$

Similarly, Alloy's assertion check is sound iff for every Alloy specification *Spec* and assertion A, the counter example $\mathcal{I}$ produced by Alloy's analysis in response to the command

```
check A for n
```

for some scope n does indeed make all the constraints in *Spec* true, but not A, i.e.,

$$\mathcal{I} \models \varphi_{Spec} \wedge \neg\, \varphi_A$$

To prove soundness, we would have to consult the implementation to show that it is a sound implementation of Alloy's semantics.

Completeness

Alloy's consistency analysis can be said to be complete iff for every instance $\mathcal{I}$ with

$$\mathcal{I} \models \varphi_{Spec} \wedge \varphi_P$$

for some specification *Spec* and predicate P Alloy's analysis in response to the command

```
run P() for maxSize
```

where *maxSize* is the size of the signature interpretation in $\mathcal{I}$ with the most elements, i.e.,

$$maxSize = max\{|\ s^{\mathcal{I}}\ |\ |\ \ s \in \mathcal{F}_{Sig}\}$$

will eventually (using the "Next instance" command) produce an isomorphic instance of $\mathcal{I}$. Similarly for assertion analysis.

To prove completeness, we would have to consult the implementation to show that it is exhaustive, i.e., that in scope *maxSize* it will not overlook satisfying isomorphic instances.

11.11 Summary

In summary, the Alloy notation allows the succinct specification of class models and their properties. The Alloy checker allows states of the specification to be displayed and properties of the specification to be verified with respect to the given scope. The counter examples produced by the checker allow the refinement of the specification. Due to the scope restriction, the checker only has to perform a bounded, finite search. This allows all interactions with the checker to be fully automatic and always terminating. However, bounded unsatisfiability is a weaker property than general unsatisfiability. Consequently, the absence of a counter example in some scope does not imply the absence of a counter example in all scopes. Figure 11.11 summarizes the usage of the Alloy checker and illustrates the iterative nature of the process of designing and analyzing Alloy specifications.

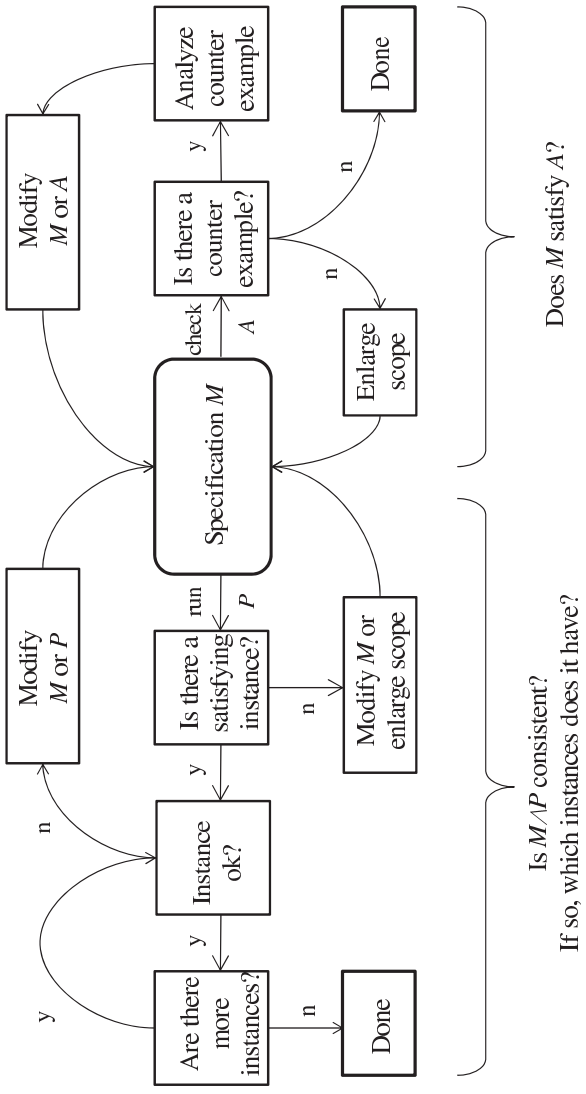


Figure 11.11: Design and analysis of Alloy specifications

Chapter 12

Class model design and analysis using UML and OCL

12.1 Background on UML

As object-orientation became more and more popular in the late 1980s and early 1990s a large number of methods for object-oriented software development were proposed including, for instance,

- Object-oriented Analysis method (OOA) by Coad and Yourdon [CY89],
- Object-oriented Modeling technique (OMT) by Rumbaugh et al [RBP+91],
- Jacobson Method by Jacobson et al [JCJG92], and
- Booch's method (ODD) [Boo91].

However, as the world expected a "methods war", Rumbaugh and Jacobson joined Booch's company Rational Software and declared in 1995 that they intended to merge their methods. Initial efforts led to drafts of the so-called *Unified Method*. Then, however, the focus shifted from the development of a unified method to the development of a unified modeling language which led to UML [FS97]. UML thus is a combination of the notations and models of the methods of Rumbaugh, Booch, and Jacobson. It includes different models (like class models, use case models, interaction and collaboration diagrams, state and activity diagrams, and implementation diagrams), their (informal) semantics, and an interchange format for CASE tools. UML has the backing of a large consortium that includes Microsoft, Oracle, HP, and IBM and was made standard by the Object Management Group (OMG).

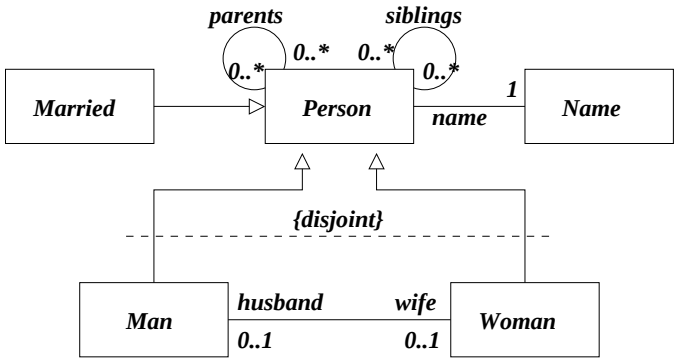


Figure 12.1: Graphical constraints for family tree in UML

12.2 Class modeling in UML

UML class models contain all the features of Alloy object models. Classes may have attributes (fields), operations, and subclasses. Associations show dependencies between classes. Multiplicities put restrict these dependencies quantitatively. UML represents these features graphically in very much the same way as Alloy does. Consider, for instance, the UML class diagram in Figure 12.1 for the family tree example presented in Section 11.7. The most notable difference probably is the use of *rolenames* in UML which name the ends of associations. For instance, in Alloy, we have two relations *husband* and *wife* between *Man* and *Woman*. In UML, however, the association itself remains unnamed and the two rolenames *wife* and *husband* are used instead. As we will see, the rolenames of an association can be used to refer to the objects connected by the association. Also note that the disjointness of the types *Man* and *Woman* is expressed graphically in UML, whereas it is expressed textually in Alloy.

UML class models offer a few features that Alloy does not support. For instance, UML class models may contain so-called *association classes*, that is, classes that are attached to an association. Moreover, in UML, class models may also be related through *aggregation* or *composition*. For more information on class modeling with UML see [PS99], for instance.

12.3 The Object Constraint Language

Just like in Alloy, a class model in UML consists of graphical and textual constraints. While Alloy chooses predicate logic to express all constraints, UML uses the Object Constraint Language (OCL) for textual constraints. OCL is an expression language that enables one to restrict the values of (parts of) an object-oriented model or system using invariants and pre- and post-conditions OCL originated in the Syntropy method and has been developed by IBM. It is used in the semantics document of UML (the so-called *meta model*) to place

constraints on which UML models are well-formed, as well as being available to users of UML to place constraints on their own models. OCL intends to combine the rigor and precision of mathematics with the ease of use of natural language. In other words, it is meant to be formal, precise and unambiguous and to be used by average software developers. To this end, OCL was made a strongly typed, functional (i.e., side-effect free) language with the syntax of an object-oriented programming language. However, the developers of OCL have achieved their goals only partially. OCL does not have a formal semantics yet, so it is not really a formal language. Moreover, it features all the complexities of a full-fledged object-oriented programming language which in many cases makes it hard to use.

12.4 Summary

Alloy was designed to be as compatible as possible with UML class models. Alloy also intends to overcome the flaws of OCL.

- Both Alloy and UML express the graphical constraints of a class model similarly. However, in contrast to UML, Alloy does not support role names, aggregation, composition, or association classes.
- Alloy has a mathematical semantics based on basic concepts from discrete mathematics: sets and relations.
- Alloy's precise semantics allows the development of analysis tools. The concept of bounded satisfiability allows Alloy's checker to perform non-trivial analyses fully automatically. However, the scope restricts the search space somewhat artificially, and implies that an Alloy analysis typically is partial and cannot be used to prove that a specification always guarantees some property.

Chapter 13

Bibliographic notes and acknowledgments

The background material in Chapter 7 follows the presentation in [PS99]. The introduction to state diagrams, class modeling, and Alloy in Chapters 10 and 11 used [Jac99, LG00, Jac00, Jac01] as sources. The file system example used in Chapter 10 and Section 11.6 is taken from [Jac00]. Chapter 12 was prepared using [WK99] and [DHH01].

A number of articles report on the successful application of formal techniques in large projects. These include [MDL90, CM90, BBFM99].

Part V

Using Formal Specifications for Testing (Weeks 7-8)

Chapter 14

Property-based Testing

While regular testing is typically based on checking that specific inputs yield specific expected outputs, property-based testing (PBT) emphasizes the properties that the inputs and outputs are expected to satisfy. Paired with automation to generate suitable test inputs automatically it aims to facilitate effective, high-volume testing. Before we introduce PBT in more detail, we briefly review example-based testing using examples involving *topological sort* implemented with JUnit 5.

14.1 Motivation

Testing typically means that the software under test (SUT) is executed on specific inputs and for each such execution, the actual output produced is compared with the expected output. Consider, for instance, the routine

$$\mathit{topSort}(\mathit{Integer}\ n,\ \mathit{List}\langle\mathit{List}\langle\mathit{Integer}\rangle\rangle\ \mathit{deps})\colon\ \mathit{List}\langle\mathit{Integer}\rangle\ \mathit{ord}$$

which computes a ordering of the nodes in $\{0, \ldots, n-1\}$ such that none of the dependencies in *deps* are violated. So, e.g., $\mathit{deps} = [[1, 0], [2, 1]]$ indicates that node 1 depends on node 0 and that node 2 depends on node 1. So, we would expect the execution of $\mathit{topSort}(3, [[1, 0], [2, 1]])$ to produce the output $[0, 1, 2]$. Similarly, in case of cyclic dependencies, no ordering can be found and we would expect the routine to inform us of that by, e.g., throwing a specific exception. As illustrated in Figure 14.1, test frameworks such as JUnit provide good support for this.

This kind of *example-based testing* is commonplace. However, it also presents the following challenges:

> **1. Input-related challenges:** Exactly which inputs should the SUT be tested on? How many? Too few or poorly chosen inputs can lead to an insufficiently tested SUT.

```
// test1: acyclic dependencies: [[1,0], [2,1]]
@Test
void topSortTest1() {
   List<List<Integer>> deps = new ArrayList<>(Arrays.asList(
                                Arrays.asList(1,0),
                                Arrays.asList(2,1)
   ));
   System.out.println("dependencies: " + toStringSorted(deps));
   List<Integer> ord = topSort(3, deps);
   System.out.println("ordering: " + ord);
   List<Integer> expected = new ArrayList<>(Arrays.asList(0,1,2));
   Assertions.assertThat(ord).isEqualTo(expected);
}

// test2: cyclic dependencies: [[1,0], [2,1], [0,2]]
@Test
void topSortTest2() {
   List<List<Integer>> deps = new ArrayList<>(Arrays.asList(
                                Arrays.asList(1,0),
                                Arrays.asList(2,1),
                                Arrays.asList(0,2)
   ));
   System.out.println("dependencies: " + toStringSorted(deps));
   Assertions.assertThatExceptionOfType(CyclicDependenciesException.class).
  isThrownBy(() -> {
      List<Integer> ord = topSort(3, deps);
   }).withMessageContaining("Cyclic dependencies");
}
```

Figure 14.1: Example-based tests for topological sort in JUnit 5

2. Output-related challenges: For each input, what exactly is the expected output? Determining and capturing the expected output can be challenging, especially for large inputs (e.g., how exactly does an AVL-tree look like after some long sequence of insert and delete operations have been performed on it?). An even more significant problem is that for some problems the expected output may not be unique, i.e., for a single input there may not be a single expected output. As an example consider the invocation of $topSort(3, [[1,0],[2,0])$ which can result in one of two correct orderings: $[0,1,2]$ and $[0,2,1]$.

Some of these challenges can be mitigated. For instance, JUnit 5 features *parametrized tests* which facilitate the expression of several test cases into a single test. Parametrized tests allow for inputs and associated expected outputs to be enumerated explicitly:

```
@ParameterizedTest
@CsvSource({"test,TEST", "tEst,TEST", "Java,JAVA"})
void toUpperCase_ShouldGenerateExpectedUppercase(String inp, String expected) {
   String actualValue = inp.toUpperCase();
   assertEquals(expected, actualValue);
 }
```

or programmatically:

```
@ParameterizedTest
@MethodSource("dependenciesAndOrderingProvider")
void topSortTest3(List<List<Integer>> deps, List<Integer> expected) {
    System.out.println("dependencies: " + toStringSorted(deps));
```

```
        List<Integer> ord = topSort(3, input);
        System.out.println("ordering: " + ord);
        Assertions.assertThat(ord).isEqualTo(expected);
    }
    static Stream<Arguments> dependenciesAndOrderingProvider() {
       List<Integer> d10 = Arrays.asList(1,0);
       List<Integer> d21 = Arrays.asList(2,1);
       List<Integer> d20 = Arrays.asList(2,0);
       return Stream.of(
                 arguments(Arrays.asList(d10,d21), Arrays.asList(0,1,2)),
                 arguments(Arrays.asList(d10,d21,d20), Arrays.asList(0,1,2)),
                 arguments(Arrays.asList(d10,d20), Arrays.asList(0,1,2))
       );
    }
```

Note how the `dependenciesAndOrderingProvider` method computes the stream of inputs and outputs.

Also, sometimes it is possible to automatically generate suitable test input. As seen above, a convenient data structure for this purpose are *streams* with lazy evaluation as supported in many functional languages such as Haskell

```
filter odd [1..]    -- infinite stream of natural numbers starting with 1
```

and object-oriented languages such as Java

```
List<Integer> num = Arrays.asList(2,3,5,7);   // finite stream
List<Integer> square = num.stream().map(x -> x*x).collect(Collectors.toList());
Stream<Integer> str = Stream.iterate(0, n -> n+2).limit(20);
                                                 // bounded infinite stream
```

However, even these techniques don't help us deal with situations in which multiple different outputs are correct. They also don't free us from having to determine and capture the expected output for every input.

To tackle these problems, we need specifications. In particular, specifications of what makes the output for some input correct, i.e., which properties the output is expected to have. These specifications can then be used for *property-based testing*.

14.2 Introduction to Property-based Testing

Typically, property-based testing (PBT) means the following:

1. **Automatic test input generation:** Test inputs are automatically generated using possibly user-defined generators that examine a customizable search space to generate input satisfying certain properties the input can be assumed to have (i.e., *pre-conditions*). The generators support a broad range of input types (primitive, composite, user-defined), and can examine sufficiently small search spaces exhaustively, but use randomization otherwise.

2. **Property checking:** The produced output is checked for expected properties (i.e., *post-conditions*) using assertions.

3. **Shrinking:** Input that produced output violating an expected property will be automatically "shrunk", i.e., the PBT tool will attempt to find the smallest input causing the violation.

PBT facilitates testing by leveraging automation (test input generation, shrinking) and specifications (to determine valid inputs and outputs). It thus addresses the problems listed above: neither inputs nor associated expected outputs need to be provided explicitly anymore. All that is needed are specifications of the properties valid inputs and outputs are expected to have. In other words, we don't need to pre-compute expected outputs or commit to one particular output. As long as the output satisfies all properties it it expected to satisfy, it will pass the test.

So, in the context of *topSort*, this means that with PBT we can automatically generate lists of dependencies and check that 1) the routine produces an ordering iff these dependencies are cycle-free or that 2) every ordering produced respects all dependencies.

PBT thus supports high-volume testing with large complex input and an emphasis on the requirements the SUT is expected to satisfy. PBT support is available for a broad range of languages, including Python, Java, JavaScript, Golang, C/C++, C#, and Haskell. That support builds on concepts and techniques known from testing such as unit testing and assertions, but also from programming in general such as stream-based generators and design patterns such as factory and fluent interfaces.

Before we discuss specific PBT libraries (such as Jqwik, the most widely used PBT library for Java) we will introduce the concepts that are key to PBT and common to most implementations. A small amount of mathematical notation will be used for this.

14.2.1 Generators

Ignoring the implementation-side of things, we can think of a *generator factory* GF as a higher-order function that takes as arguments a type T and boolean-valued function $p : T \to \mathbb{B}$ and returns a generator capable of generating all those values $x \in T$ for which $p(x)$ holds. We call $GF(T, p)$ *a generator for* T *with condition* p, and use

$$gen(GF(T, p)) = \{x : T \mid p(x)\}$$

to refer to the set of values it can generate.

Examples of generators are given below. To facilitate the definition of generators, we will use the notation $[(x_1, \ldots, x_n) \to bExp]$ to represent boolean-valued functions where $bExp$ is a boolean expression containing the variables x_1 through x_n. This notation allows us to define anonymous functions (i.e., functions without names) and corresponds directly to *lambda expressions* in Java.

$$\begin{array}{lcl}
G_1 & = & GF\big(\mathbb{N},\ [(x)\ \to\ 0 \le x \le 99 \wedge x\%4 = 0]\big) \\
G_2 & = & GF\big(\mathbb{N} \times \mathbb{N},\ [(x, y)\ \to\ x < y]\big) \\
G_3 & = & GF\big(\mathit{List}(\mathbb{N}) \times \mathbb{N},\ [(l, n)\ \to\ 3 \in l \wedge \mathit{len}(l) < 5 \wedge n = \mathit{len}(l)]\big) \\
G_4 & = & GF\big(\mathit{String} \times \mathit{String},\ [(s_1, s_2) \to s_1 \textit{ is valid name and } \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad s_2 \textit{ is valid street address}]\big)
\end{array}$$

Generator G_1 generates all natural numbers between 0 and 99 divisible by 4. G_2 generates the set of all pairs of natural numbers such that first is less than the second. G_3 generates the set of all pairs (l, n) where l is a list of natural numbers that contains at least one 3 and whose length is less than 5, and n is the length of l. G_4 generates pairs of strings representing valid addresses.

Generators can be composed. In the examples below, we use $x \in GF(T, p)$ to abbreviate $x \in gen(GF(T, p)$.

$$
\begin{array}{lcl}
G_5 & = & GF\big(\mathbb{N} \times \mathbb{N},\ [(x, y)\ \rightarrow\ x \in G_1 \wedge y = x + 2]\big) \\
G_6 & = & GF\big(\mathbb{N} \times \mathit{String},\ [(x, s)\ \rightarrow\ \mathit{len}(s) = x \wedge \exists (s_1, s_2) \in G_4.\ s = s_1 + \, , \ + s_2]\big) \\
G_7 & = & GF\big(\mathit{List}(\mathbb{N}),\ [(l)\ \rightarrow\ \forall\, n : \mathbb{N}.\ n \in \mathit{elems}(l) \Rightarrow n \in G_1]\big)
\end{array}
$$

Generator G_5 uses generator G_1 to produce pairs of natural numbers (x, y) such that x is between 0 and 99 and divisible by 4 and y is equal to $x + 2$. E.g., $(4, 6) \in G_1$, but $(4, 5) \notin G_1$. Generator G_6 uses G_4 to produce address strings s and then pairs them up with a natural number representing its length. G_7 produces lists containing the numbers generated by G_1.

Let students be represented by tuples as follows:

$$
\begin{array}{rcl}
\mathit{Student} & = & \{(\mathit{name}, \mathit{plan}, \mathit{yearEnrolled}) \mid \mathit{name} \in \mathit{String} \wedge \\
& & \quad \mathit{plan} \in \{\mathit{Comp}, \mathit{Math}, \mathit{Bio}, \ldots\} \wedge \mathit{yearEnrolled} \in \mathbb{N}\}
\end{array}
$$

Then, G_8 is a generator for Computing students enrolled in 2020:

$$
G_8 \ = \ GF\big(\mathit{Student},\ [(s) \rightarrow s.\mathit{plan}{=}\mathit{Comp} \wedge s.\mathit{yearEnrolled}{=}\mathit{2020}]\big)
$$

Similarly, assuming that binary search trees are expressed by

$$
\begin{array}{rcl}
\mathit{BST} & = & \{(k, v, \mathit{null}, \mathit{null}) \mid k \in \mathbb{N} \wedge v \in \mathit{String}\} \cup \\
& & \{(k, v, l, r) \mid k \in \mathbb{N} \wedge v \in \mathit{String} \wedge l \in \mathit{BST} \wedge r \in \mathit{BST}\}
\end{array}
$$

then G_9 is a generator for balanced binary trees of height at least 3:

$$
G_9 \ = \ GF\big(\mathit{BST},\ [(bt) \rightarrow \mathit{balanced}(bt) \wedge \mathit{height}(bt) > 2]\big)
$$

As we will see, PBT tools have built-in, default generators for basic types (such as numeric, character, string), but also for enumerations, collections, arrays, maps, and iterators. These generators can be customized via constraints. However, it is also possible to create user-defined generators.

Generators in Jqwik

Built-in generators In Jqwik, the built-in generators can be used in the property definition. Below is an example of a property that constrains the built-in generator for integers so that it generates positive integers without ever generating any integer more than once. As we can see, Jqwik makes heavy use of annotations to customize generators.

```
@Property
@Report(Reporting.GENERATED)
public void aProp1(@ForAll @Positive @UniqueElements Integer i) {
   Assertions.assertThat(i).isLessThan(100000);
}
```

Checking of a property causes a number of test executions (also called *tries* in Jqwik). In each test execution of the property above, an integer is generated, assigned to parameter `i` and the code in the body of the property is run. A test execution stops with a failure in case of an assertion violation or an uncaught exception. Unless Jqwik is configured otherwise, the check stops with the first failure. If the execution of the body completes without failure, the test execution is complete and the next test execution starts, unless the upper limit of test executions has been reached. This upper limit can be configured via

```
jqwik.tries.default=1000
```

in file `junit-platform.properties`.

The next example composes the integer generator with the list generator to generate lists of integers that have exactly 5 unique elements all of which are between 0 and 9. Note how the integer generator is used with a suitable constraint (`@IntRange(min=0,max=9)`):

```
@Property
@Report(Reporting.GENERATED)
void myProp2(@ForAll @Size(5) @UniqueElements
                              List<@IntRange(min=0,max=9) Integer> aList) {
   Assertions.assertThat(aList).doesNotHaveDuplicates();
   Assertions.assertThat(aList).allMatch(anInt -> anInt>=0 && anInt<=9);
}
```

User-defined generators Below is an example of a user-defined generator similar to generator G_1 above. You can think of class `Arbitraries` as a generator factory:

```
@Property
@Report(Reporting.GENERATED)
public void myProp3 (@ForAll("leapYear") Integer y) {
   Assertions.assertThat(y%2==0 && (i<91 || i>95)).isTrue();
}
@Provide
public Arbitrary<Integer> leapYear() {
   return Arbitraries.integers()
         .between(0,3000)
         .filter(n -> (n%4==0 && n%100!=100 && n%400==0));
}
```

Finally, note that the definition of a property can use several generators:

```
@Property
@Report(Reporting.GENERATED)
public void myProp3 (@ForAll("leapYear") Integer year,
                     @ForAll @IntRange(min=0,max=2) Integer i,
                     @ForAll @CharRange(from='a',to='z')
                             @StringLength(3) String str) {
   Assertions.assertThat(...);
}
```

14.2.2 Testing individual operations

Let $opn(x : T_1) : T_2$ be an operation (method). Suppose that according to the requirements, *opn* can be considered to work correctly if when invoked on input x satisfying p, it returns a result satisfying q. In other words, if it satisfies the following Hoare triple (pre- and post-condition statement):

$$\{p(x)\}$$
$$res = opn(x)$$
$$\{q(res)\}$$

To use PBT, we need to build a generator $G(T_1, p)$ and express $q(x)$ as an assertion. Then, with the help of PBT, we can automatically

1. generate a user-specified number of inputs x that satisfy $p(x)$,
2. execute $opn(x)$, and
3. check that the result returned satisfies $q(res)$.

Also, should one of the generated inputs x cause $opn(x)$ to produce a result that does not satisfy $q(res)$, then PBT will attempt to shrink x, i.e., it will try to find the smallest input x_s that satisfies $p(x_s)$ but causes *opn* to produce a result violating $q(res)$. Supported notions of input size typically include 'less than' on the integers or the length of a list or array, but can often also be user-defined.

Example: Topological Sort

Let's use PBT for our $topSort(m, deps)$ routine mentioned above. But, what are the properties that we can use? Which properties is the output expected to have? Which properties does the input need to have so that the routine can work? In other words, what are the relevant pre- and post-conditions? In the following, for natural numbers i and j, let $\{i..j\}$ abbreviate the set $\{n \in \mathbb{N} \mid i \leq n \wedge n \leq j\}$.

Pre-conditions: The first argument m defines the set of nodes $\{0..m-1\}$ and the second argument *deps* indicates the dependencies between these nodes, i.e., $(i, j) \in deps$ means that node i depends on node j. So, we can assume $m \in \mathbb{N}$ and $m > 0$ and $deps \subseteq \{0..m-1\} \times \{0..m-1\}$.

Post-conditions: Suppose m and *deps* satisfy these conditions and let $ord = topSort(m, deps)$. What are the properties that *ord* is expected to have?

> $P_1(ord, deps)$: *"ord respects the dependencies in deps"*. Formally, let $ord = [n_0..n_k]$ for some $k \in \mathbb{N}$ (note that the routine may return an empty list). Then, for any node n_j that occurs after some other node n_i in the order (i.e., $j > i$), n_i must not depend on n_j:
>
> $$\forall\, n_i \in order.\ \forall\, n_j \in ord \mid j > i \Rightarrow (n_i, n_j) \notin deps$$

Or, equivalently[1], for any two nodes n_i and n_j in the ordering, if n_i depends on n_j, then n_j must occur before n_i:

$$\forall\, n_i \in \mathit{ord}.\ \forall\, n_j \in \mathit{ord} \mid (n_i, n_j) \in \mathit{deps} \Rightarrow j < i$$

Note that P_1 is necessary for *ord* to be a correct topological ordering, but not sufficient, because it allows for, e.g., the ordering to be empty.

$P_2(m, \mathit{ord})$: *"ord contains every node exactly once"*. Formally,

$$\mathit{asSet}(\mathit{ord}) = \{0..m-1\}$$

where $\mathit{asSet}(l)$ returns the elements of list l as a set.

Together, P_1 and P_2 ensure that any ordering produced is correct. However, they still allow for the routine to not produce any ordering due to non-termination or a runtime exception.

$P_{3a}(\mathit{deps})$: *"If deps is acyclic, then topSort will not throw any runtime exception"*.

$P_{3b}(\mathit{deps})$: *"If deps is acyclic, then topSort will eventually terminate"*.

We are not quite done. We still need a property to capture that if the dependencies are cyclic, then a `CyclicDepsException` is thrown (and no other exception).

$P_4(\mathit{deps})$: *"If deps is cyclic, then topSort throws a* `CyclicDepsException`*"*.

Defining suitable generators: The next step is to define suitable generators. The following generators could be used to test *topSort* with respect to properties P_1 and P_2:

$$\begin{aligned}
G_{10} &= \mathit{GF}\big(\mathbb{N} \times \mathit{Pow}(\mathbb{N} \times \mathbb{N}),\ [(n,d) \ \rightarrow\ n = 4 \land d \subseteq \{0..3\} \times \{0..3\}]\big)\\
G_{11} &= \mathit{GF}\big(\mathbb{N} \times \mathit{Pow}(\mathbb{N} \times \mathbb{N}),\ [(n,d) \ \rightarrow\ n = 5 \land d \subseteq \{0..2\} \times \{1..4\}]\big)\\
G_{12} &= \mathit{GF}\big(\mathbb{N} \times \mathit{Pow}(\mathbb{N} \times \mathbb{N}),\ [(n,d) \ \rightarrow\ n = 4 \land d \subseteq \{0..3\} \times \{0..3\} \land\\
&\qquad\qquad \forall\, 0 \le i \le 3.\ (i,i) \notin d]\big)\\
G_{13} &= \mathit{GF}\big(\mathbb{N} \times \mathit{Pow}(\mathbb{N} \times \mathbb{N}),\ [(n,d) \ \rightarrow\ n = 100 \land d \subseteq \{0..99\} \times \{0..99\} \land\\
&\qquad\qquad 10 \le |\,d\,| \le 20]\big)
\end{aligned}$$

For property P_{3a}, generator G_{14} allows focussing on acyclic dependencies:

$$G_{14} = \mathit{GF}\big(\mathbb{N} \times \mathit{Pow}(\mathbb{N} \times \mathbb{N}),\ [(n,d) \ \rightarrow\ n = 10 \land d \subseteq \{0..9\} \times \{0..9\} \land \mathit{acyclic}(d)]\big)$$

assuming a helper predicate *acyclic* that checks if a set of dependencies is acyclic.

The number of inputs generated by G_{10} through G_{12} is small enough to allow for *exhaustive enumeration* within a reasonable amount of time, whereas *random generation* would have to be used for generators G_{13} and G_{14}.

[1] See discussion of the contrapositive of an implication during the logic review on page 25

In Jqwik The code below captures property P_1 in Jqwik:

```
@Property
@Report(Reporting.GENERATED)
void propCheckComputedOrdering1b (
   @ForAll("dependencyListsCyclesPossible1") List<List<Integer>> deps) {
      try {
         List<Integer> ord = topSort(numNodes, deps);
         System.out.println("dependencies: " + deps);
         System.out.println("order: " + ord);
         Assertions.assertThat(checkOrdering(ord,deps)).isTrue();
      } catch (CyclicDependenciesException e) {
         System.out.println("CyclicDepsException thrown");
      }
   }
```

The code assumes a global variable `numNodes`, a helper method `checkOrdering`, and a generator `dependencyListsCyclesPossible1`:

```
@Provide
public Arbitrary<List<List<Integer>>> dependencyListsCyclesPossible1() {
   final int maxPairs = numNodes;
   Arbitrary<Integer> nodes = Arbitraries.integers().between(0,numNodes-1);
   Arbitrary<List<Integer>> pairs = nodes.list().ofSize(2);
   return pairs.list()
               .ofMinSize(0)
               .ofMaxSize(maxPairs)
               .uniqueElements();
}
```

The following expresses property P_2:

```
@Property
@Report(Reporting.GENERATED)
void propCheckComputedOrdering2 (
   @ForAll("dependencyListsCyclesPossible1") List<List<Integer>> deps) {
      try {
         List<Integer> ord = topSort(numNodes, deps);
         boolean res=false;
         for (int i=0; i<numNodes; i++) {
            res = ord.remove((Object) i);
            Assertions.assertThat(res).isTrue();
         }
         Assertions.assertThat(ord).isEmpty();
      }
      catch (CyclicDependenciesException e) {
         System.out.println("CyclicDepsException thrown");
      }
}
```

To focus the testing on acyclic dependencies, a different generator could be used

```
@Provide
public Arbitrary<List<List<Integer>>> dependencyListsWithoutCycles() {
   final int maxPairs = numNodes*numNodes;
   final int minPairs = 0;
   Arbitrary<Integer> num = Arbitraries.integers().between(0,numNodes-1);
   Arbitrary<List<Integer>> tuples = num.list().ofSize(2)
                                               .uniqueElements()
                                               .filter(t-> t.get(0)!=t.get(1));
   return tuples.list()
                .ofMinSize(minPairs)
                .ofMaxSize(maxPairs)
                .uniqueElements()
                .filter(ll -> aCyclic(ll));
}
```

where `aCyclic` is a helper method checking the list of dependencies for cycles.

Property P_{3a} is easy to capture because in Jqwik any uncaught exception causes a test to fail.

```
@Property
@Report(Reporting.GENERATED)
void propCheckProperTermination1 (
            @ForAll("dependencyListsWithoutCycles") List<List<Integer>> deps) {
   topSort(numNodes, deps);
}
```

Due to the undecidability of the Halting Problem, non-termination cannot be detected in general and a timeout mechanism would have to be used to implement property P_{3b}. JUnit 5 offers support for this:

```
@Test
@Timeout(value = 5000, unit = TimeUnit.MILLISECONDS)
void propCheckProperTermination2 () {
   ...
   List<Integer> ord = topSort(numNodes, deps);
   ...
}
```

14.2.3 Testing sequences of operations

Sometimes, the data that the operation *opn* to be tested is subject to modification via some other operations. The impact of these modification operations on the output of *opn* can be a great source of properties for PBT. For instance, what is the impact of changing the weight of an edge on the shortest paths in a directed, weighted graph? How would we expect the height of an AVL-tree to change by the removal of a node? How would the strengthening of the conditions on getting a mortage at a bank influence any eligibility checking the bank has in place? We want to capture these relationships between operations as properties so that we can use them to test the operations.

In the context of our *topSort* example, consider two operations:

$$\mathit{addDep}(\mathit{Integer}\ i,\ \mathit{Integer}\ j,\ \mathit{List}\langle\mathit{List}\langle\mathit{Integer}\rangle\rangle\,\mathit{deps}) \ :\ \mathit{List}\langle\mathit{List}\langle\mathit{Integer}\rangle\rangle$$

adds a dependency (i, j) to a list of dependencies *deps*, and

$$\mathit{remDep}(\mathit{Integer}\ i,\ \mathit{Integer}\ j,\ \mathit{List}\langle\mathit{List}\langle\mathit{Integer}\rangle\rangle\,\mathit{deps}) \ :\ \mathit{List}\langle\mathit{List}\langle\mathit{Integer}\rangle\rangle$$

removes the dependency (i, j). Clearly, performing these operations on a list of dependencies *deps* impacts the problem of finding a topological sort for *deps*.

For *remDep*, the problem becomes easier in the sense that the removal would not invalidate any ordering already found:

$$P_{rem_1}(m, \mathit{deps}, \mathit{ord}) \ = \\ \quad \forall\, i, j \in \{0..m-1\}.\ P_1(m, \mathit{deps}, \mathit{ord}) \Rightarrow P_1(m, \mathit{remDep}(i, j, \mathit{deps}), \mathit{ord})$$

or

$$P_{rem_1}(m, \mathit{deps}, \mathit{ord}) \ = \\ \quad \forall\, i, j \in \{0..m-1\}.\ \mathit{ord} = \mathit{topSort}(m, \mathit{deps}) \Rightarrow P_1(m, \mathit{remDep}(i, j, \mathit{deps}), \mathit{ord})$$

Note that

$$\forall\, i, j \in \{0..m-1\}.\ ord = topSort(m, deps) \Rightarrow \\ ord = topSort(m, remDep(i, j, deps))$$

is too strong and might fail even for a correct implementation because the removal might cause the routine to produce a different (yet still correct) topological ordering.

On the other hand, the addition of a dependency may invalidate any previously found ordering:

$$\begin{array}{l} P_{add_1}(m, deps, ord)\ = \\ \quad \forall\, i, j \in \{0..m-1\}.\ acyclic(deps) \Rightarrow \\ \qquad (cyclic(addDep(i, j, deps)) \vee before(j, i, topSort(m, addDep(i, j, deps)))) \\ P_{add_2}(m, deps, ord)\ = \\ \quad \forall\, i, j \in \{0..m-1\}.\ (ord = topSort(m, deps) \wedge before(i, j, ord) \Rightarrow \\ \qquad \bar{}P_1(m, addDep(i, j, deps), ord) \end{array}$$

where $before(Integer\ i, Integer\ j, List\langle Integer\rangle\ ord) : \mathbb{B}$ returns *true* if i occurs before j in ord. Note, however, that an added edge (i, k) may not add an actual new dependency. In that case, a previously found ordering would still be valid:

$$\begin{array}{l} P_{add_3}(m, deps, ord)\ = \\ \quad \forall\, i, j, k \in \{0..m-1\}. \\ \qquad (i, j) \in deps \wedge (j, k) \in deps \wedge ord = topSort(m, deps)\ \Rightarrow \\ \qquad\quad P_1(m, addDep(i, k, deps), ord) \end{array}$$

Finally, there is an idempotency relationship between adding a new dependency and then removing it:

$$\begin{array}{l} P_{add_4}(m, deps, ord)\ = \\ \quad \forall\, i, j, k \in \{0..m-1\}. \\ \qquad (i, j) \notin deps \wedge ord = topSort(m, deps)\ \Rightarrow \\ \qquad\quad P_1(m, remDep(i, j, addDep(i, k, deps), ord)) \end{array}$$

In Jqwik

We show how some of these properties can be implemented in Jqwik. The following captures property P_{rem_1}:

```
@Property
@Report(Reporting.GENERATED)
void propRemDependency (
         @ForAll("dependencyListsWithoutCycles") List<List<Integer>> deps,
         @ForAll @IntRange(min=0, max=numNodes-1) Integer i,
         @ForAll @IntRange(min=0, max=numNodes-1) Integer j) {
   List<Integer> ord = topSort(numNodes, deps);
   deps = removeDependency(i, j, deps);
   Assertions.assertThat(checkOrdering(ord, deps)).isTrue();
}
```

Property P_{add_1} can be expressed by:

```
@Property
@Report(Reporting.GENERATED)
void propAddDependency1 (
         @ForAll("dependencyListsWithoutCycles") List<List<Integer>> deps,
         @ForAll @IntRange(min=0, max=numNodes-1) Integer i,
         @ForAll @IntRange(min=0, max=numNodes-1) Integer j) {
   Assume.that(i != j);
   List<List<Integer>> deps1 = addDependency(i, j, deps);
   if (aCyclic(deps1)) {
      List<Integer> ord = topSort(numNodes, deps1);
      int indexOfI = ord.indexOf(i);
      int indexOfJ = ord.indexOf(j);
      Assertions.assertThat(indexOfJ).isLessThan(indexOfI);
   } else {
      Assertions.assertThatExceptionOfType(CyclicDependenciesException.class)
                .isThrownBy(() -> {topSort(numNodes, deps1);})
                .withMessageContaining("Cyclic dependencies");
   }
}
```

What does `Assume.that(i != j)` do in the code above? If *bExp* evaluates to *false*, the statement `Assume.that(bExp)` causes the current test execution (try) to be aborted and the next input to be generated. So, in the property above, if test inputs i and j happen to generated in such a way that $i = j$, then the execution of the current test run is stopped, the inputs generated are discarded, and the inputs for the next test run are generated (unless the maximal number of tries has been reached). The example shows that `Assume.that(bExp)` is convenient to enforce constraints on inputs generated by different generators. However, it can also be used to place additional constraints on the inputs generated by a single generator. For instance, with `Assume.that` property P_{3a} could have been implemented using a generator capable of generating cyclic dependencies:

```
@Property
void propCheckProperTermination1 (
      @ForAll("dependencyListsCyclesPossible1") List<List<Integer>> deps) {
   Assume.that(aCyclic(deps));
   topSort(numNodes, deps);
}
```

However, the use of `Assume` can lead to the majority of the generated inputs to be rejected and thus waste resources and weaken the check. To alert the user of this, Jqwik keeps track of the number of tries (i.e., test executions) and actual checks (i.e., executions of the body of the property that either fail or complete). For instance, when using the property above, more than half of the generated dependencies are rejected due to a cycle:

```
                              |--------------------jqwik--------------------
tries = 1000                  | # of calls to property
checks = 466                  | # of not rejected calls
generation = RANDOMIZED       | parameters are randomly generated
```

If the ratio (called `maxdiscardratio`) between tries and actual checks exceeds a user-specified upper limit (5 by default), Jqwik aborts the check. Excessive rejection of generated inputs via `Assume.that(bExp)` should be avoided by modifying the generator of the inputs such that it respects `bExp` and avoids generating all or many of the inputs violating `bExp`.

14.2.4 Key Challenges

The following can make the use of PBT challenging:

Finding Properties

Properties should be *correct*, i.e., their failure should be the manifestation of a real bug in the code, and *effective*, i.e., they should have maximal "bug revealing power". That means that the identification of properties may require some care. The requirements the software is to satisfy, as well as the properties of individual and collections of data structures and operations on them, as illustrated, can be good sources of properties. To ensure that the properties have been captured correctly "sanity checks" are recommended. Also, it is useful to keep an eye on the relationships between properties. For instance, the addition of a property P_2 that is implied by an already existing property P_1 (i.e., $P_1 \Rightarrow P_2$) will only help reveal bugs in cases in which P_1 fails.

Expressing Properties

Another challenge can be expressing the relevant properties in the chosen PBT or assertion library. For instance, how can properties related to *non-functional requirements*, i.e., requirements related to the consumption of resources be expressed? E.g., how can a real-time constraint such as *"given a certain input, does the time or memory that an operation needs for processing this input lie below a certain threshold?"* be expressed?. Another class of challenging properties are properties related to certain computational paradigms such as concurrency or distribution. E.g., how can it be expressed that the shared state of a distributed application always eventually reaches a consistent state?

Implementing Properties

But, bugs might also creep into the implementation of a property and, as mentioned above, the need for "testing the tests" does come up.

Implementing Generators

Finally, the effectiveness of PBT also depends on the generators used. Existing libraries and tools (see below) greatly facilitate the implementation of generators for input data of useful size and format. But, given the richness of these libraries and their use of advanced programming techniques, code for a user-defined generator can easily become buggy or the semantics of constraints for built-in generators can be misunderstood. As for properties, the testing of the implementation of generators is recommended.

So, ironically, technology that is supposed to facilitate testing actually introduces the need for more, different testing. The hope is that, overall, the benefits outweigh the costs overall. Note that once properties have been identified, they

can greatly aid not just quality assurance (QA) activities, but also other software lifecycle activities such as documentation, maintenance and evolution.

14.2.5 Tools

PBT is supported by a range of libraries available for almost any programming language including Java, Python, Haskell, C/C++, C#, JavaScript, and Go. We mention three of the most well-known ones here: QuickCheck for Haskell, Hypothesis for Python, and Jqwik for Java. Apart from the host language, these libraries have several strong similarities: All are open-source, are available on GitHub, contain a large collection of built-in, customizable generators for basic and stuctured types, support user-defined generators, offer good supporting documentation, and have tags on StackOverflow.

Java: Jqwik

Jqwik is built on top of the JUnit 5 platform and can be used as an alternative test engine (other supported engines are the standard engine Jupiter and the JUnit 4 engine Vintage). Jqwik is distributed under the Eclipse Public Licence 2.0. Jqwik does not implement its own assertions. Instead, a third-party assertion library such as the one offered by JUnit needs to be used. Just like the Jqwik user guide, the examples shown in these notes use AssertJ [Ass22]. Jqwik offers a rich collection of built-in, customizable generators for basic types, but also for strings, arrays, lists, sets, streams, iterators, and arrays. Custom-made generators can be implemented using class `Arbitraries` as a kind of facade and generator factory.

Fluent Interfaces Key to using `Arbitraries` and AssertJ is the notion of a *fluent interface*, an API design pattern for object-oriented languages aimed at increasing code readability. In short, a class can be said to have or implement a fluent interface, if the methods of the class $m_1, \ldots, m_n$ all return the context object (*this* or *self*) such that the successive invocations of these methods can be expressed in a single statement as a sequence of "chained" or "cascading" method calls separated by dots:

$$c.m_1(\ldots).m_2(\ldots).\ldots.m_n(\ldots)$$

where c is an instance of the class. Jqwik and AssertJ use fluent interfaces, as do, for instance, streams in Java 8. Examples are the generators `leapYear`, `dependencyListsCyclesPossible1`, and `dependencyListsWithCycles`, and the property `PropAddDependency1` on pages 152, 155, and 155, respectively. Additional examples can be found online and in the literature on fluent interfaces which also contains a discussion of their advantages and disadvantages [Wik22c].

Python: Hypothesis

Arguably, the most popular library for PBT in Python is Hypothesis [Hyp22]. It is distributed under the Mozilla Public Licence 2.0, is compatible with different

testing frameworks (such as `py.test` and `unittest`) and can be installed via `pip`.

Haskell: QuickCheck

QuickCheck pioneered PBT [CH00, Qui22] and has influenced all available PBT tools. Properties and generators can be expressed quite elegantly in Haskell, motivating us to also use a functional style to explain the main ideas behind PBT. So, arguably, PBT integrates quite well into the functional programming paradigm.

14.2.6 More Examples and Case Studies

Classical algorithms are useful to illustrate PBT, because the problems themselves and relevant properties associated with the solutions are typically known. Apart from topological sort, we will briefly discuss the use of PBT on two more such problems: shortests paths and balanced search trees. We will also illustrate the construction of generators for rich, user-defined data models. The end of the section provides some pointers to more examples of the use of PBT, some of them introductory.

Shortests Paths

Let $G = (V, E)$ be a directed, weighted graph in which edge weights are natural numbers i.e., for all $(i, j) \in E$, we have $\mathit{weight}(i, j, G) \in \mathbb{N}$. The problem is to compute the length of the shortest path from every node in G to every other node. For nodes $i, j \in V$, let $\mathit{dist}(i, j, G)$ be the length of the shortest path from i to j in G. If there is no path from i to j, we have $\mathit{dist}(i, j, G) = \infty$. Apart from *dist*, we assume an operation $\mathit{addW}(i, j, w, G)$ which returns a graph just like G except that the weight of the edge from $i \in E$ to $j \in E$ is increased by $w \in \mathbb{N}$, an operation $\mathit{remE}(i, j, G)$ which returns the graph with the edge from i to j removed, and an operation $\mathit{addE}(i, j, w, G)$ which returns the graph with an edge (i, j) with weight w added. Parts of the relationship between *dist*, *addW*, *addE*, and *remE* can be captured by the following properties.

Property P_1 captures that increasing the weight of an edge preserves unreachability.

$$\begin{aligned} P_1 \;=\; \forall\, i, j \in V.\ \mathit{dist}(i, j, G) = \infty \;\Rightarrow\; & \\ \forall (k, l) \in E.\ \forall\, w \in \mathbb{N}.\ & \\ \mathit{dist}(i, j, \mathit{addW}(k, l, w, G)) = \infty & \end{aligned}$$

Property P_2 expresses that increasing the weight of an edge by some natural number w will not increase the length of a shortest path by more than w.

$$\begin{aligned} P_2 \;=\; \forall\, i, j \in V.\ & \\ \forall\, d \in \mathbb{N}.\ \mathit{dist}(i, j, G)) = d \;\Rightarrow\; & \\ \forall (k, l) \in E.\ \forall\, w \in \mathbb{N}.\ & \\ d \leq \mathit{dist}(i, j, \mathit{addW}(k, l, w, G)) \leq d + w & \end{aligned}$$

Property P_3 says that adding an edge will not increase the length of any shortest path.

$$P_3 \;=\; \forall\, i, j \in V.\ \forall (k,l) \in (V \times V) \backslash E.\ \forall\, w \in \mathbb{N}.\\ dist(i, j, addE(k, l, w, G))) \leq dist(i, j, G)$$

Property P_4 formalizes that removing an edge will not decrease the length of any shortest path.

$$P_4 \;=\; \forall\, i, j \in V.\ \forall (k,l) \in E.\\ dist(i, j, G)) \leq dist(i, j, remE(k, l, G)))$$

The Jqwik code implementing these tests including a generator for adjacency matrices to represent graphs can be found at [Din22]. Similar properties could be identified to capture the effect of other operations to, e.g., reduce the weight of an edge or to reverse the direction of the edges in the graph.

Balanced Binary Search Trees

Balanced binary search trees (BBSTs) also are a great example to illustate PBT, because two key properties are "baked into" their definition:

1. **Search tree property P_{search}:** To qualify as a search tree, a binary tree t typically has to be such that all keys in the left subtree of a node n in t are not greater than the key in the node, while all keys in the right subtree are greater.

2. **Balancedness property P_{bal}:** This property restricts the amount by which the height of the subtrees can differ. For AVL trees, for instance, the height of the subtrees can differ by at most 1.

We will quickly formalize these. Let V be a finite, possibly empty set of nodes, let $\epsilon \notin V$ denote the "empty node", and let $V_\epsilon = V \cup \{\epsilon\}$. Then,

$$bt = (V_\epsilon, root \in V_\epsilon, lft : V \to V_\epsilon, rgt : V \to V_\epsilon)$$

is a *binary tree* if

$$\begin{array}{l} (root = \epsilon \;\Leftrightarrow\; V = \emptyset) \wedge \\ \big(root \neq \epsilon \;\Rightarrow\; \forall\, v \in V.\ (v = root \Rightarrow\mid parents(v) \mid = 0) \wedge \\ \qquad\qquad\qquad\qquad (v \neq root \Rightarrow\mid parents(v) \mid = 1) \wedge \\ \qquad\qquad\qquad\qquad root \in parents^*(v)\big) \end{array}$$

where $parents(v \in V_\epsilon) = \{v' \in V \mid lft(v') = v \vee rgt(v') = v\}$ and $parents^*$ is the reflexive and transitive closure[2] of *parents*. Next,

$$bst = (V_\epsilon, root \in V_\epsilon, lft : V \to V_\epsilon, rgt : V \to V_\epsilon, key : V \to Key, val : V \to Val)$$

is a *binary search tree* over $< \subseteq Key \times Key$ if

[2] As discussed in Chapter 11 on Alloy

1. $(V_\epsilon, root \in V_\epsilon, lft : V \to V_\epsilon, rgt : V \to V_\epsilon)$ is a binary tree,
2. $<$ is a strict total order on Key, and
3. bst satisfies P_{search} where

$$P_{search} \;=\; \forall\, v \in V.\; (hasLft(v) \Rightarrow key(lft(v)) < key(v)) \;\wedge\; (hasRgt(v) \Rightarrow key(v) < key(rgt(v)))$$

where $hasLft(v \in V) = lft(v) \neq \epsilon$ and similarly for $hasRgt$.

Finally, a binary search tree over $<$ is is *balanced* if it satisfies P_{bal} where

$$P_{bal} \;=\; \forall\, v \in V.\; \mid height(lft(v)) - height(rgt(v)) \mid \leq 1$$

where

$$height(v \in V_\epsilon) \;= \begin{cases} 0 & \text{if } v = \epsilon \\ 1 + max(height(lft(v), height(rgt(v)) & \text{otherwise} \end{cases}$$

To check these properties, we can traverse the tree and check P_{search} and P_{bal} for each node. However, there is operation that facilitates the checking of many BBST properties, including P_{search}. That operation is $keysInOrd : V \to List\langle Key\rangle$ which traverses the tree rooted at the argument node *in-order* (as opposed to pre-order or post-order) and returns the keys in the tree in a list:

$$keysInOrd(v) \;= \begin{cases} [\,] & \text{if } v = \epsilon \\ keysInOrd(lft(v)) + [key(v)] + keysInOrd(rgt(v)) & \text{otherwise} \end{cases}$$

A binary search tree t satisfies P_{search} precisely when $keysInOrd(root(t))$ is sorted in strictly ascending order:

$$P_{search}(t) \;\Leftrightarrow\; sorted(keysInOrd(root(t))$$

Note that in-order traversal is typically also used whenever the tree needs to be serialized, i.e., put into a linear format, as for, e.g., a `toString` method.

But, BBSTs typically also come with operations to find, insert, and delete a key. The BBST properties and operations have relationships (e.g., operations that change the tree have to preserve the BBST properties) that are a rich source of properties suitable for PBT.

To implement a generator for BBSTs (see Figure 14.2), we successively insert the key-value pairs produced by a generator for lists of these pairs (`keysAndValues`) into an initially empty tree.

Generating User-defined, Complex Object Graphs

Sometimes, applications have a rich, user-defined data model and manipulate its instances during execution. Consider, for instance, software to manage organizations with members, names, memberships, fees, etc. In other words, an application that uses the data model shown in Figure 14.3.

```
@Provide
Arbitrary<BstAVL<Integer, Integer>> trees() {
   Arbitrary<Integer> keys = Arbitraries.integers().between(0,15);
   Arbitrary<Integer> values = Arbitraries.integers().between(0,0);
   final int maxNumNodes = 16;
   Arbitrary<List<Tuple2<Integer,Integer>>> keysAndValues =
                                        Combinators.combine(keys,values)
                                                   .as(Tuple::of)
                                                   .list()
                                                   .uniqueElements(Tuple1::get1)
                                                   .ofMinSize(0)
                                                   .ofMaxSize(maxNumNodes);
   return keysAndValues.map(keyValueList -> {
                               BstAVL<Integer,Integer> bst = BstAVL.empty();
                               for (Tuple2<Integer,Integer> kv : keyValueList)
                                  bst = bst.insert(kv.get1(), kv.get2());
                               return bst;
                            });
}
```

Figure 14.2: Jqwik generator for AVL trees

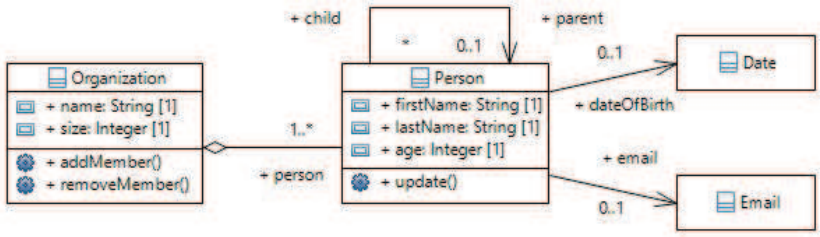


Figure 14.3: Class diagram for organizations data model

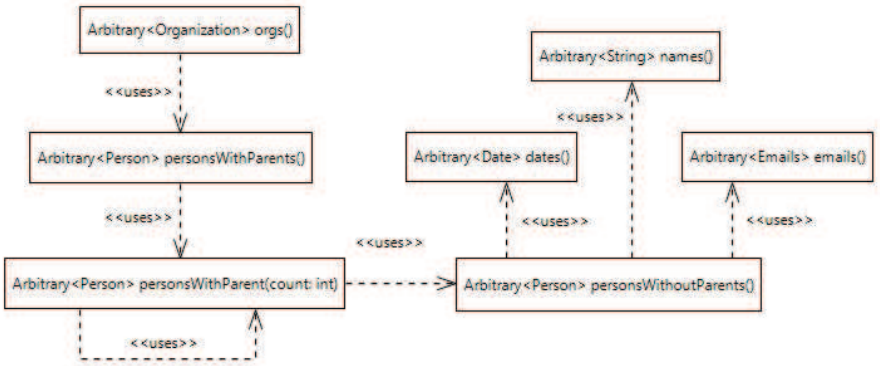


Figure 14.4: Dependency diagram of Jqwik generators for class `Organization`

PBT tools offer support for the implementation of generators that create instances of this data model. In Jqwik, for instance, `Arbitraries` can be combined, i.e., call each other, including recursively. Figure 14.4 shows the generators for the Organizations data model and their dependencies (i.e., which generator calls which other generator). With the help of generators for dates of birth, names, and email addresses, generator `personsWithoutParents()` generates instances of `Person` without parents. Generator `personsWithParents(count:int)` produces instances of `Person` that have `count` number of ancestors. If `count` is 0, it uses `personsWithoutParents()`. Otherwise, it creates an instance p of `Person` and then calls itself recursively with `count` decremented by 1 to create the ancestors of p. The example illustrates how generators for applications with complex data models can be implemented.

Other Examples

An introduction to PBT with Jqwik using JSON-based serialization and deserialization of user records as an example can be found online at [Red20]. The example also shows the use of some additional built-in features of Jqwik: generators for dates, times, and timezones, and a statistics package that collects information about the generated test cases.

The detailed post at [Rot20] also provides an introduction to PBT with Jqwik, but uses a formalization and an implementation of the well-known Rock, Paper, Scissors game.

Finally, the use of PBT for web services is discussed in [LTSF14, AS19].

Part VI

Using Formal Specifications for Behaviour Modeling and Analysis (Weeks 9-11)

Chapter 15

Introduction and overview (Week 9)

Alloy allows the description and analysis of structure as found in object-oriented systems. In this part, the use of formal specifications for describing and analyzing the behaviour of systems will be explored. There are many ways in which this can be done. In this part, we will focus on one of the most successful: *temporal logic model checking*.

Before we start, let us briefly recapitulate what we have learnt:

- Predicate logic is very expressive and widely used and understood. These features make it an appealing choice as a specification formalism. The most severe disadvantage of predicate logic is that the complexity that comes with its expressiveness prevents the implementation of a fully automatic analysis tool.
- The problem of deciding whether or not a given predicate logic formula is a theorem is undecidable, that is, it is impossible to write a program P that takes a predicate formula φ as input, always terminates, and returns "yes" if φ is a theorem and "no" otherwise. It follows that the problem of deciding whether a given predicate logic formula is satisfiable also is undecidable.
- Despite its use of predicate logic, the concept of "scope" allows Alloy to offer full automation by artificially bounding the search space.

As we will see, just like Alloy's analysis, temporal logic model checking also is fully automatic. However, whereas Alloy is based on predicate logic, model checking is (typically) based on temporal logic.

Temporal logic

In the early 1980 researchers realized the ease with which properties of programs can be expressed in a certain variant of modal logic, called *temporal logic*. Modal

logic differs from propositional logic and predicate logic in that it allows one to express that a formula is "necessarily" or "possibly" true. Consider, for instance, the statement "It is raining". From a propositional point of view this statement is either true or false. However, we know from experience that it neither is necessarily true nor necessarily false. Rather, we can imagine circumstances (eg., time or place), also called "worlds", in which it is true and circumstances in which it is false. Modal logic allows us to express that, for instance, a formula is true under all circumstances, that is, in all worlds. Temporal logic is a modal logic that uses worlds to formalize the notion of time. This notion of time enables temporal logic to describe how the execution of a program unfolds, that is, describe what kind of intermediate states the program runs through between start and termination. For instance, statements like "during every execution of the program, the variable x will never take on the value 0" or "during every execution of the program, whenever the variable *req* is true, there is a state in the future in which the variable *ack* will also have the value true" can easily be expressed in temporal logic. Thus, note that temporal logic formulas are interpreted over *sequences* of states rather than a just a single state.

Model checking

Computer science in general and software engineering in particular feature many formalisms that can be expressed in terms of *states* and *transitions* and are thus based on *state machines*. For instance, StateCharts or Message Sequence Diagrams are state machines. In the software design setting, the value of these formalisms is at least threefold:

- State machines form an intuitive, natural way of describing computation.
- Phrasing (aspects of) the design in some mathematically precise notation forces the designer to think very carefully about desired and undesired behaviours.
- Formalisms like state machines allow the precise, unambiguous, albeit sometimes tedious, communication of the behaviour of the system to be designed.

Obviously, the difficulty of finding design errors in a state machine increases with its size. Unfortunately, even systems consisting of relatively few lines of code, often give rise to large state machines. This is particularly true, for instance, in the presence of concurrency or distributed data, because they cause complex behaviours. Therefore, the analysis of realistic systems by simply inspecting the state machines that they give rise to is typically completely impractical, because too many of the subtle design flaws like, for instance, race conditions or deadlock, would be overlooked. So, what do we do? We use computers to help us solve the problem they have created. In other words, we find techniques that allow us to use computers to analyze state machines automatically.

Model checking is such a technique. More precisely, it allows us to determine whether a given state machine with a finite number of states satisfies a

certain property. *Temporal logic model checking* uses temporal logic to describe the property. In other words, the natural connection between computation and temporal logic described above is used and modified such that the question whether a given state machine satisfies a given temporal logic property can be decided automatically and implemented efficiently. The major price we have to pay is that the state machine has to be *finite*, that is, it must contain only finitely many states. Indeed, this constitutes not only the most important requirement of model checking but also its most severe drawback. Given a finite state machine M modeling some system and some initial state s, temporal logic model checking solves the following problem: Does the temporal logic formula φ hold in the model M in the initial state s? Formally, model checking determines whether $M, s \models \varphi$ holds where $\models$ is an appropriately defined satisfaction relation. Note that since we are only interested in the truth of φ in a particular model, the model checking problem is considerably less difficult than deciding whether φ is a theorem. In general, system analysis through temporal logic model checking consists of the following three steps:

1. **Modeling** The system that is to be analyzed is modeled as a finite state machine. This step is described next in Chapter 16.

2. **Specification** The specification the system is to satisfy is expressed in some temporal logic. One of the most prominent temporal logics used in model checking is *computation tree logic (CTL)* which is covered in Chapter 17.1.

3. **Verification** The model and the specification are given as input to the model checker. The model checker proves or disproves the specification in the model, typically fully automatic. More precisely, it terminates either with the output "Yes" meaning that the given finite state machine does indeed satisfy the given specification, or with the output "No" meaning that the given specification is not satisfied. If a *counter example* illustrating why the specification does not hold can be given, it will be output, too. Figure 15.1 illustrates the verification step. Chapter 17.1 will illustrate the use of model checking by means of a detailed example (Chapter 17.2), describe the model checking algorithm (Chapter 17.3), and sketch a particular model checker called NuSMV (Chapter 18).

Impact Model checking has had a very significant impact on the research and practice of software and systems development:

- Many model checkers have been developed ([Mod17] lists over 40) of which NuSMV [NuS17], Spin [Spi17], CADP [CAD17], Java Pathfinder [JPF17], CBMC [CBM17], UPPAAL [UPP17], and Prism [Pri17] probably are the most well-known.

- As of August 2022, Google Scholar lists over 310,000 papers with the term "model checking" in the title or the text.

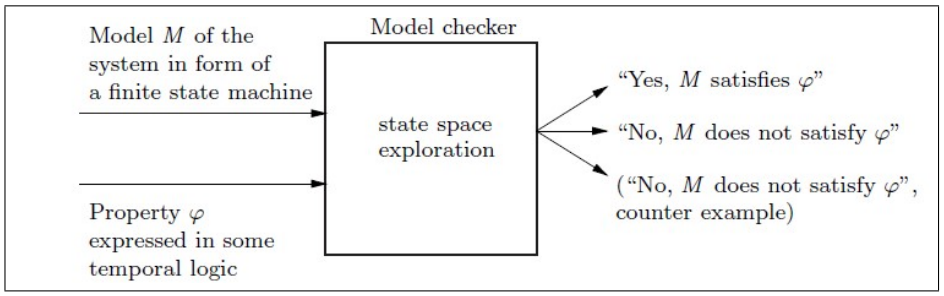


Figure 15.1: Schematic description of the verification step in model checking

- Several textbooks on model checking have been published, e.g., [CGP99, Hol03, BK08].
- In 2007, the developers received the Turing Award, generally recognized as the highest distinction in computer science and often referred to as the "Nobel Prize of computing".
- Many companies and organizations use model checking in their software and systems development, including Intel, Motorola, NASA, IBM, and Microsoft.

As usual, the presentation of this analysis technique will concentrate on the fundamental concepts. We will first (Chapter 16) present the mathematical structure that is typically used in model checking to represent the behaviour of a system. Then, the logical formalism using to capture properties and the model checking algorithm will be described (Chapter 17. Finally, we sketch the use of the NuSMV model checker (Chapter 18). For more details on using NuSMV, please consult its documentation [NuS17].

Chapter 16

Modeling systems (Weeks 9-11)

16.1 Modeling systems as Kripke structures

We have already argued in the introduction that state machines are a natural model for computation. In this section, we take a closer look at the notion of a state machine. More precisely, we will see how state machines can conveniently be expressed in terms of a simple mathematical structure called *Kripke structure*.

States

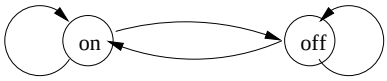
An important ingredient is the notion of state. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a finite set of system or program variables and let D be the set of values that these variables range over. D contains the value $\perp$ which allows us to express that the value of a variable v is undefined. In that case, we write $v = \perp$. A *state* s of a system is a function $s : V \to D$ that associates a value with every variable. For example, if a program contains only two variables x and y ranging over integers, then $(x = 0, y = 1)$, $(x = -1, y = 25)$, $(x = 0, y = \perp)$ are states of the program. Given a set of variables V and a set of values D, let S denote the set of all states over V and D. S is also called the *state space* of the program. An important subset of S is the set of *initial states*, that is, the set of states in which a system execution may be initiated.

Transitions

Typically, a system passes through a possibly large number of states during a single execution. The transitions describe the system's ability to move from one state to the next. Our model of the system may be *non-deterministic*, that is, a state encountered during the execution of a program may have more than one successor state. Consequently, a relation $R \subseteq S \times S$ is the appropriate mathematical model of transitions.

Modeling the system on the right level of abstraction

A critical issue in modeling concurrent systems is determining the right *granularity* of states and transitions. On the one hand, it is important to obtain transitions that are atomic in the sense that no observable state of the system can result from executing just a part of the transition. If we violate this rule and choose a transition relation that is too coarse, the Kripke structure may not include some states that are observable in the original system. There's a mismatch between the reality and our model of it. As a result, verification techniques such as model checking may fail to find important errors. Consider, for instance, the following model of a Coke machine:



The behaviours of this model resemble the behaviours of a real Coke machine only very remotely. This model is inadequate to verify that, for instance, the machine only ever outputs a Coke after the correct amount of money has been inserted. The chosen transition relation is way too coarse.

On the other hand, a problem can also arise when the granularity is too fine. In this case, transitions can interact to create new states that are not feasible or reachable in the actual system. As a result, model checking may find spurious errors that will never occur in practice. For an example, consider a system with two variables x and y and two transitions α and β that can be executed concurrently.

$$\begin{aligned} \alpha&: \quad x := x + y \\ \beta&: \quad y := y + x \end{aligned}$$

with the initial state $x = 1 \wedge y = 2$. Also consider a finer grained implementation of the same transitions. This implementation uses the assembly language instructions for loading, adding, and storing between a memory address and a register:

$$\begin{array}{llll} \alpha_0: & \mathit{load}\ R_1, x & \beta_0: & \mathit{load}\ R_2, y \\ \alpha_1: & \mathit{add}\ R_1, y & \beta_1: & \mathit{add}\ R_2, x \\ \alpha_2: & \mathit{store}\ R_1, x & \beta_2: & \mathit{store}\ R_2, y \end{array}$$

Executing α and then β results in the state $x = 3 \wedge y = 5$. When β is executed before α, we obtain $x = 4 \wedge y = 3$. If, on the other hand, the finer grained implementation is executed in the order $\alpha_0\beta_0\alpha_1\beta_1\alpha_2\beta_2$, the result is $x = 3 \wedge y = 3$. Suppose that $x = 3 \wedge y = 3$ violates some desired property of the system. Further suppose that the system is implemented using the transitions α and β. Then, it is impossible to have $x = 3 \wedge y = 3$ at the same time. However, if we model the system with the finer transitions $\alpha_0, \alpha_1, \alpha_2, \beta_0, \beta_1$, and β_2, we may erroneously conclude that the system is incorrect. Conversely, suppose that the system is implemented using $\alpha_0, \alpha_1, \alpha_2, \beta_0, \beta_1$, and β_2. In this case, it is possible to reach a state in which both $x = 3$ and $y = 3$. If we now

model the system with α and β, we will erroneously conclude that the system is correct.

To summarize, when modeling a system as a state machine, we must take into account granularity considerations like the one described above. In other words, we want the model and the implementation to be on the same level of granularity.

Kripke structures

In the 1950s and 1960s, the logician Saul Kripke used a particular kind of mathematical structure to study modal logic. It turns out that these so-called *Kripke structures*, sometimes also called *labeled transition systems* provide us with exactly what we need to model systems. They contain states and transitions. Moreover, they contain a labeling function describing the atomic propositions that hold in a state. As we will see, this labeling function will help us to express properties of states, entire executions and programs.

Definition 16.1.1. A *Kripke structure* M is a 4-tuple $M = (S, S_0, R, L)$ where

1. S is a finite set of states,
2. $S_0 \subseteq S$ is the set of initial states,
3. $R \subseteq S \times S$ is a total transition relation, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$, and
4. $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions that are true in that state, that is, for all $p \in AP$ and $s \in S$, we have $p \in L(s)$ if and only if p holds in s.

Given a structure M, a *path* π is an infinite sequence of states

$$\pi \quad = \quad s_0 s_1 s_2 \ldots s_i \ldots$$

such that $s_0 \in S_0$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

We want to use Kripke structures to represent systems. The instantiation of S and R is straight-forward. The states of the Kripke structure are just the states of the system. Similarly for the transition relation. However, how do the atomic propositions look like? Often, we will simply use boolean-values variables as atomic propositions, e.g.,

$$AP \quad = \quad \{p, q, r\}$$

More generally, given a system with variables V that range over the domain D, the set of atomic propositions AP over V and D might be

$$AP \quad = \quad \{v = d \mid v \in V \wedge d \in D\}.$$

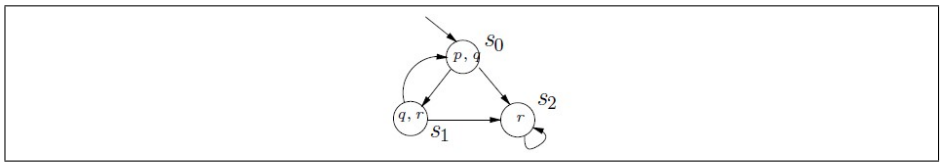


Figure 16.1: Graphical representation of a Kripke structure with atomic propositions $AP = \{p, q, r\}$. Every state contains the atomic propositions true in that state.

Figure 16.1 contains an example of a Kripke structure with $AP = \{p, q, r\}$. Note that atomic propositions not true in a state are not shown. E.g., p and q hold in state s_0, while r does not, i.e., $L(s_0) = \{p, q\}$.

To summarize, given a system with variables V that range over the domain D, the Kripke structure M corresponding to that system is $M = (S, S_0, R, L)$ where

1. S is the set of states of the system, that is, a set of functions $s : V \to D$.
2. $S_0 \subseteq S$ is the set of initial states, that is, the set of states in which the system is executed.
3. $R \subseteq S \times S$ is the transition relation describing the behaviour of the system. Because the transition relation of a Kripke structure must be total, we must extend the relation R if some state s has no successor. In this case, we modify R so that $R(s, s)$ holds.
4. $L : S \to 2^{AP}$ is a function such that for all $v = d \in AP$ and $s \in S$, we have $v = d \in L(s)$ if and only if $s(v) = d$.

Example 16.1.1. To illustrate how Kripke structures model systems, consider a simple program with variables x and y that range over $D = \{0, 1\}$. Thus, a valuation for the variables x and y is just a pair $(d_1, d_2) \in D \times D$ where d_1 is the value for x and d_2 is the value for y. The system consists of a single assignment statement

$$x := (x + y) \ \mathsf{mod}\ 2,$$

which is executed in an infinite loop starting from a state in which $x = 1$ and $y = 1$. The Kripke structure $M = (S, S_0, R, L)$ corresponding to this system is:

$$\begin{aligned}
S &= D \times D, \\
S_0 &= \{(1,1)\}, \\
R &= \{((1,1),(0,1)), ((0,1),(1,1)), ((1,0),(1,0)), ((0,0),(0,0))\}, \\
L((1,1)) &= \{x = 1, y = 1\}, \\
L((0,1)) &= \{x = 0, y = 1\}, \\
L((1,0)) &= \{x = 1, y = 0\}, \\
L((0,0)) &= \{x = 0, y = 0\}.
\end{aligned}$$

The only path in the Kripke structure that starts in an initial state is

$$(1,1)(0,1)(1,1)(0,1)\ldots$$

This path represents the only computation of the system. Note how the mod operator ensures that the state space of our model is finite and thus guarantees the most crucial requirement of model checking.

Non-determinism

In general, a computation is called non-deterministic, if it can produce more than one result when run using the exact same initial state and inputs. In the context of modeling systems, non-determinism is often used to capture the fact that a particular (sequence of) computing steps can have different outcomes, but we do not know how exactly an outcome is chosen and how exactly these computation steps look like, or that information is not relevant for the purpose of the model and would only complicate the model unnecessarily. Often, such computing steps are interactions with parts of the environment that

- result in some kind of input being provided to the system from, e.g., a user, a sensor, or another software component,
- impact execution in some other way. Examples include the execution and preemption of concurrent threads according to some scheduling policy by the runtime environment.

But, as we will see, we will also use non-determinism to, e.g., abstract from the exact point when a subcomputation terminates.

In the context of Kripke structures, the definition is very simple.

Definition 16.1.2. We say that Kripke structure $M = (S, S_0, R, L)$ is *non-deterministic* if it contains at least one state s such that M can transition to more than one state, i.e., if $(s, s') \in R$ and $(s, s'') \in R$ and $s' \neq s''$.

Example 16.1.2. According to the above definition, the Kripke structure in Figure 16.1 is non-deterministic (states s_0 and s_1 both have more than one successor) while the one in Example 16.1.1 is not.

16.2 Modeling systems as first order formulas

While Kripke structures allows us to express states and transitions, they are cumbersome to work with directly. For instance, to describe the initial states we have to explicitly enumerate all of them. More severely, to describe a transition relation we have to explicitly describe every single transition that the system may exhibit. What we need is a more concise, less verbose representation.

The main idea is to use formulas of first order logic to represent states and transitions. More precisely, we identify a formula φ with the set of states S_φ in which it evaluates to true, that is, $S_\varphi = \{s \mid \models_s \varphi\}$. For instance, $x = 5 \wedge y = 0$

represents the set of states $\{s \mid s(x) = 5 \wedge s(y) = 0\}$. The point is that this representation allows us to ignore variables we are not interested in. For instance, suppose that a program using variables x and y can only be executed when x has value 1. Furthermore, suppose that x and y range over the numbers from 0 to 10. The formula $x = 1$ is a much more concise representation of the set of permissible initial states of the program than

$$\{(x = 1, y = 0), (x = 1, y = 1), (x = 1, y = 2), \ldots, (x = 1, y = 10)\}.$$

In addition to representing sets of states, we must be able to represent sets of transitions. To do this, we reuse an idea also used by Z and Alloy. This time, we use a formula to represent a set of ordered pairs of states. We cannot do this using just a single copy of the system variables V, so we create a second set of variables V'. We think of the variables in V as *present state* variables and the variables in V' as *next state* variables. Each variable v in V has a corresponding next state variable v' in V'. A pair (s, s') of states $s : V \to D$ and $s' : V' \to D$ represents a single transition from present state s to next state s'. Consequently, a set of pairs (s, s') describes a transition relation.

Example 16.2.1. We revisit Example 16.1.1. The system contains only two variables x and y ranging over $\{0, 1\}$, that is,

$$V = \{x, y\}$$

and

$$D = \{0, 1\}.$$

The initial states and the transition relation of the system are characterized by

$$\begin{aligned} \mathcal{S}_0(V) &= x = 1 \wedge y = 1 \\ \mathcal{R}(V, V') &= x' = (x + y) \bmod 2 \wedge y' = y \end{aligned}$$

respectively.

16.2.1 How to obtain the corresponding Kripke structure

In the following subsections, we will show how the behaviour of different classes of systems can be represented in terms of first order formulas $\mathcal{S}_0(V)$ and $\mathcal{R}(V, V')$. Then, given $\mathcal{S}_0(V)$ and $\mathcal{R}(V, V')$, we obtain the corresponding Kripke structure $M = (S, S_0, R, L)$ as follows:

- The set of states S is the set of all functions $s : V \to D$. Remember that the definition of a Kripke structure requires S to be finite. Consequently, the domain D of values must be finite.
- The set of initial states S_0 is the set of all states s_0 in which $\mathcal{S}_0(V)$ evaluates to *true* when each $v \in V$ is assigned the value $s_0(v)$, that is, $S_0 = \{s_0 \mid \models_{s_0} \mathcal{S}_0(V)\}$.

- Let s and s' be two states, then $(s, s') \in R$ holds if $\mathcal{R}(V, V')$ evaluates to *true* when each $v \in V$ is assigned the value $s(v)$ and each $v' \in V'$ is assigned the value $s'(v')$.
- The labeling function $L : S \to 2^{AP}$ is defined so that $L(s)$ is the subset of all atomic propositions true in s. If v is a variable over the truth values, then $v \in L(s)$ indicates that $s(v) = \mathit{true}$, and $v \notin L(s)$ indicates that $s(v) = \mathit{false}$.

16.2.2 Example: Modeling digital circuits

We briefly show how to describe digital circuits by first order formulas. Since hardware verification is not the focus of this course, we just present the basic idea here. For simplicity, we assume that each state holding element of a circuit either has value 0 or value 1. Let V be the set of state holding elements of a circuit. For a synchronous circuit, the set V typically consists of the outputs of all the registers in the circuit together with the primary inputs. For asynchronous circuits, all wires in the circuit are usually considered to be state holding elements. If we create a boolean variable for each element in V, then a state can be described by a valuation assigning either 0 or 1 to each variable. Given a valuation, we can write a boolean expression that is true for exactly that valuation. For example, given $V = \{v_1, v_2\}$ and the valuation $(v_1 = 1, v_2 = 0)$, we derive the boolean formula $v_1 \wedge \neg\, v_2$. As before, we adopt the convention that a formula represents the set of all valuations that make it true. Thus, for describing circuits the full expressive power of predicate logic is not needed; propositional logic is sufficient. The propositional formulas $\mathcal{S}_0(V)$ and $\mathcal{R}(V, V')$ will represent the set of initial states and the transition relation of the circuit, respectively.

Synchronous circuits

The operation of synchronous circuits consists of a sequence of steps. In each step, the inputs to the circuit change and the circuit is allowed to stabilize. Then, a clock pulse occurs, and the state holding elements change. The method for deriving the transition relation of a synchronous circuit is illustrated using a small example. The circuit in Figure 16.2 is an modulo 8 counter.

Let $V = \{v_0, v_1, v_2\}$ be the set of state variables for this circuit, and let $V' = \{v_0', v_1', v_2'\}$ be another copy of the state variables. The transitions of the counter are given by

$$\begin{aligned} v_0' &= \neg\, v_0 \\ v_1' &= v_0 \oplus v_1 \\ v_2' &= (v_0 \wedge v_1) \oplus v_2 \end{aligned}$$

where $\oplus$ is the exclusive or operator. The above equations can be used to define the relations

$$\mathcal{R}_0(V, V') \quad \stackrel{def}{=} \quad (v_0' = \neg\, v_0)$$

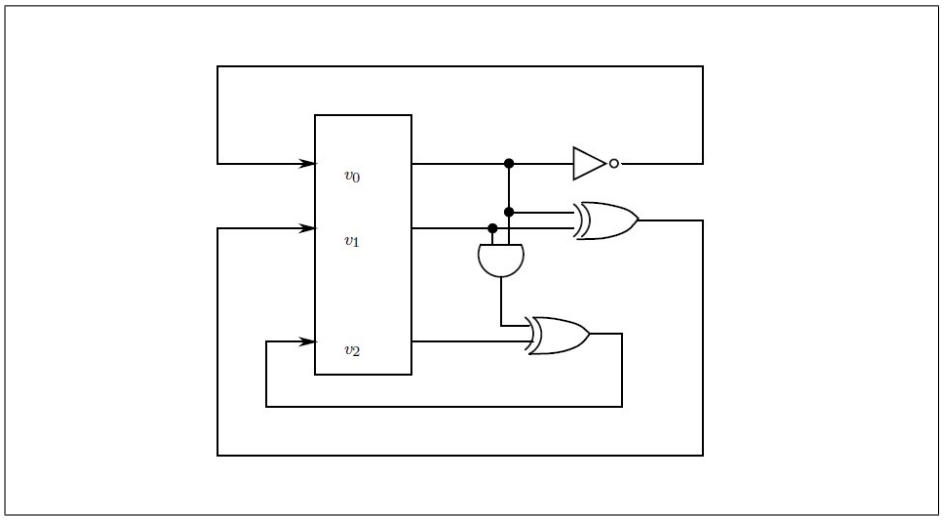


Figure 16.2: A synchronous circuit for a modulo 8 counter

$$\begin{aligned} \mathcal{R}_1(V, V') &\stackrel{def}{=} (v_1' = v_0 \oplus v_1) \\ \mathcal{R}_2(V, V') &\stackrel{def}{=} (v_2' = (v_0 \wedge v_1) \oplus v_2) \end{aligned}$$

which describe the constraints each v_i' must satisfy in a legal transition. Because all the changes occur at the same time, the constraints are combined by taking their conjunction to construct a formula for the transition relation:

$$\mathcal{R}(V, V') \stackrel{def}{=} \mathcal{R}_0(V, V') \wedge \mathcal{R}_1(V, V') \wedge \mathcal{R}_2(V, V').$$

Like in Example 16.1.1, the system has few variables. Thus, rather than representing states by functions, we represent them by tuples. Assuming that the triple (d_1, d_2, d_3) describes the state in which $v_0 = d_3$, $v_1 = d_2$ and $v_2 = d_1$,

$$(0,0,0)(0,0,1)(0,1,0)(0,1,1)(1,0,0)(1,0,1)(1,1,0)(1,1,1)(0,0,0)\ldots$$

is, as expected, a computation path of the circuit. Note that the behaviour of the circuit is deterministic, that is, for each of the 8 possible initial states the circuit has exactly one computation path that originates in that state.

In the general case of a synchronous circuit with n state holding elements, we let $V = \{v_0, \ldots, v_{n-1}\}$ and $V' = \{v_0', \ldots, v_{n-1}'\}$. Analogous to the modulo 8 counter, for each state variable v_i' there is a boolean function f_i such that $v_i' = f_i(V)$. These equations are used to define the relations

$$\mathcal{R}_i(V, V') \stackrel{def}{=} (v_i' = f_i(V)).$$

The conjunction of these formulas then forms the transition relation

$$\mathcal{R}(V, V') \stackrel{def}{=} \mathcal{R}_0(V, V') \wedge \ldots \wedge \mathcal{R}_{n-1}(V, V').$$

In summary, the transition relation of a synchronous circuit is expressed as the *conjunction* of the transition relations of the individual processes.

Asynchronous circuits

In contrast, the transition relation for an asynchronous circuit is expressed as a disjunction. For the sake of simplicity, we assume that all the components of the circuit have exactly one output and have no internal state variables. In this case, it is possible to describe each component by a function $f_i(V)$, that is, given values for the present state variables v, the component drives its output to the value $f_i(V)$. Since components in an asynchronous circuit can change their outputs independently, we will use an interleaving semantics, in which only exactly one component changes at a time. The appropriate formal model of this interleaving semantics is a disjunction of the form

$$\mathcal{R}(V, V') \quad \stackrel{def}{=} \quad \mathcal{R}_0(V, V') \vee \ldots \vee \mathcal{R}_{n-1}(V, V')$$

where each $\mathcal{R}_i(V, V')$ is

$$\mathcal{R}_i(V, V') \quad \stackrel{def}{=} \quad \big(v_i' = f_i(V)\big) \wedge \textstyle\bigwedge_{j \neq i}(v_j' = v_j).$$

The conjunct $\bigwedge_{j \neq i}(v_j' = v_j)$ expresses that the values of all other variables remain unchanged. It thus is similar to the frame conditions we encountered in the context of Z, Alloy, and OCL.

The asynchronous circuit can exhibit all computation paths the synchronous circuit could. However, additionally it can exhibit paths such as

(0,0,0)(0,0,0)(0,0,1)(0,1,0)(0,1,0),(0,1,0)(0,1,0)(0,1,1)(1,0,0)(1,0,1)(1,1,0)...

That is, not every transition will change the state. Depending on which of the three variables is recomputed in each step, the state may or may not change. In the above path, either the value of v_2 or v_1 was recomputed which left the state unchanged. Note that this model allows for so-called *unfair* executions in which one variable is recomputed repeatedly while the other remain unchanged:

(0,0,0)(0,0,0)(0,0,1)(0,1,0)(0,1,0),(0,1,0)(0,1,0)(0,1,0)(0,1,0)(0,1,0)(0,1,0)...

In practice, however, this kind of unfair behaviour is extremely unlikely or even impossible (in the presence of a fair scheduler, for instance). Sections 17.4 and 18.4 will present *fairness constraints* which, when added to the model, rule out unfair executions.

For another, smaller, example illustrating the difference between synchronous and asynchronous circuits let

- $V = \{v_0, v_1\}$,
- $v_0' = v_0 \oplus v_1$, and $v_1' = v_0 \oplus v_1$.

- s be a state with $v_0 = 1 \wedge v_1 = 1$.

The only successor state of s in the synchronous model is a state with $v_0 = 0 \wedge v_1 = 0$. In the asynchronous model, however, s has one successor state with $v_0 = 0 \wedge v_1 = 1$ and another successor state with $v_0 = 1 \wedge v_1 = 0$.

16.2.3 Example: Modeling sequential programs

We now show in a little more detail how sequential programs can be modeled in terms of first order predicate logic formulas. More specifically, we will present a compilation function T, which, when given the code of a sequential program, outputs a first order formula $\mathcal{R}(V, V')$ describing the transition relation of the program.

Without loss of generality, we assume that every program statement has a unique entry point and a unique exit point. The representation of a program C is simplified substantially, if all entry points and all exit points in C are labeled. We define a labeling function $(\_)^L$ that when given a program C returns a program C^L in which all entry points and exit points carry a unique label. In sequential programs, the exit of a statement is equal to the entry point of the following statement. It is therefore sufficient to label only entry points. The labeling function is defined by induction over the structure of the input program C.

- If C is not a composite statement, that is, C is, for instance, $x := e$ or **skip**. In this case, $C^L = C$. Note that this means that we are assuming that these constructs are executed *atomically* without interruption.
- If C is a sequential composition C_1 **;** C_2, then $C^L = C_1^L$ **;** $l{:}C_2^L$.
- If C is a conditional **if** b **then** C_1 **else** C_2 **end**, then

  $$C^L \quad = \quad \textbf{if } b \textbf{ then } l_1{:}C_1^L \textbf{ else } l_2{:}C_2^L \textbf{ end}$$

  where l_1 and l_2 are new labels.
- If C is an iteration **while** b **do** C_1 **end**, then

  $$C^L \quad = \quad \textbf{while } b \textbf{ do } l_1{:}C_1^L \textbf{ end}$$

  where l_1 is a new label.

In the remainder of this section, we will assume that every program is labeled using the procedure above.

To completely describe a "snap shot" in the execution of a program C, we not only need the current values of the variables, but also an indication of "where in C the execution is". Operational semantics, for instance, uses sequences of configurations

$$\langle C_0, s_0\rangle\langle C_1, s_1\rangle\langle C_2, s_2\rangle \ldots$$

to describe an execution of a program C_0 from initial state s_0 where C_i stands for the "rest of the program" that is still to be executed after i steps. Instead, we will use a special variable *pc*. This variable serves as a program counter and indicates which statement in a program is to be executed next. *pc* will range over the program labels introduced by the labeling function $(\_)^L$. Given a set of program variables V and a program counter *pc*, let V' and pc' stand for their primed counterparts. Moreover, given $X \subseteq V$, let $same(X)$ abbreviate $\bigwedge_{x \in X} x = x'$.

We first present a formula that describes the set of initial states. Remember that m labels the entry point of the entire program. Given some condition pre (V) on the initial values of the variables in C, let

$$\mathcal{S}_0(V, pc) \quad = \quad \text{pre}\,(V) \wedge pc = m.$$

We now describe a compilation function T that translates a standard sequential program into a first order formula representing all transitions of the program. Function T takes three parameters: the entry label l, the labeled program C^L, and the exit label l^e. $T(l, C^L, l^e)$ is defined inductively over the structure of C^L and describes the set of transitions in C^L as a disjunction of all transitions in the set.

- If C^L is an assignment $v := e$, then

$$T(l, v := e, l^e) \quad = \quad pc = l \wedge pc' = l^e \wedge v' = e \wedge same(V \backslash \{v\}).$$

- If C^L is **skip**, then

$$T(l, \mathbf{skip}, l^e) \quad = \quad pc = l \wedge pc' = l^e \wedge same(V).$$

- If C^L is a sequential composition $C_1^L \,;\, l_1{:}C_2^L$, then

$$T(l, C_1^L \,;\, l_1{:}C_2^L, l^e) \quad = \quad T(l, C_1^L, l_1) \vee T(l_1, C_2^L, l^e).$$

- If C^L is a conditional **if** b **then** $l_1{:}C_1^L$ **else** $l_2{:}C_2^L$ **end**, then

$$\begin{aligned} T(l, \mathbf{if}\ b\ \mathbf{then}\ l_1{:}C_1^L\ \mathbf{else}\ l_2{:}C_2^L\ \mathbf{end}, l^e) \\ \quad = \quad & \big(pc = l \wedge pc' = l_1 \wedge b \wedge same(V)\big) \\ & \vee \big(pc = l \wedge pc' = l_2 \wedge \neg\, b \wedge same(V)\big) \\ & \vee\ T(l_1, C_1^L, l^e) \\ & \vee\ T(l_2, C_2^L, l^e). \end{aligned}$$

- If C^L is a **while** statement **while** b **do** $l_1{:}C_1^L$ **end**, then

$$\begin{aligned} T(l, \mathbf{while}\ b\ \mathbf{do}\ l_1{:}C_1^L\ \mathbf{end}, l^e) \\ \quad = \quad & \big(pc = l \wedge pc' = l_1 \wedge b \wedge same(V)\big) \\ & \vee \big(pc = l \wedge pc' = l^e \wedge \neg\, b \wedge same(V)\big) \\ & \vee\ T(l_1, C_1^L, l). \end{aligned}$$

Non-deterministic assignments

As discussed in Section 16.1, we want to be able to equip our system descriptions with some degree of non-determinism. Non-deterministic assignments allow to capture that a variable can uncontrollably take on one of a selection of values at a certain program location. Given variable v and expressions e_1 through e_n the execution of $v := \{e_1, \ldots, e_n\}$ will assign a value to v that is the result of the evaluation of one of the expressions. All of the n expressions could be chosen and repeated executions may or may not yield different results. We formalize the effect of $v := \{e_1, \ldots, e_n\}$ using our translation T to propositional logic as follows:

- If C^L is a non-deterministic assignment $v := \{e_1, \ldots, e_n\}$, then

$$\begin{aligned} T(l, v := \{e_1, \ldots, e_n\}, l^e) \quad &= \quad pc = l \wedge pc' = l^e \wedge same(V \backslash \{v\}) \\ &\quad \wedge \big(v' = e_1 \vee \ldots \vee v' = e_n\big) \end{aligned}$$

Defining the Kripke structure

Given program C with variables V and program counter pc, we use $T(m, C, m^e)$ to define $\mathcal{R}(V, pc, V, pc')$.

$$\mathcal{R}(V, pc, V, pc') \quad = \quad T(m, C, m^e) \vee (pc = m^e \wedge pc' = pc \wedge same(V))$$

The second disjunct $pc = m^e \wedge pc' = pc \wedge same(V)$ ensures that the transition relation is total as required by the definition of Kripke structures. It does so by making the state in which C has terminated ($pc = m^e$) a successor state of itself.

Using $\mathcal{S}_0$ and $\mathcal{R}(V, pc, V, pc')$ the Kripke structure M corresponding to C is now defined as described in Section 16.2.1.

Example 16.2.2. We present a small example for the translation function and its use for the definition of a Kripke structure. Consider the labelled program

$$\begin{aligned} C \quad = \quad & x := \{0, 1\}; \\ & l_1\text{: } \textbf{if } x = 0 \textbf{ then } l_2\text{: } y := 5 \textbf{ else } l_3\text{: } y := x \textbf{ end} \end{aligned}$$

The program has two variables (x and y) and a single program counter pc ranging over the labels m, l_1, l_2, l_3, and m^e. Thus, $V = \{x, y\}$. The initial states of C are described by the formula

$$\mathcal{S}_0(V, pc) \quad = \quad pc = m$$

Note that the values of x and y in the intial state are unconstrained and thus can be arbitrary. Applying the translation function T to C, we get

$$\begin{aligned} T(m, C, m^e) \quad = \quad & T(m, x := \{0, 1\}), l_1) \\ & \vee\ T\big(l_1, \textbf{if } x = 0 \textbf{ then } l_2\text{: } y := 5 \textbf{ else } l_3\text{: } y := x \textbf{ end}, m^e\big) \end{aligned}$$

$$
\begin{aligned}
= \quad & pc = m \wedge pc' = l_1 \wedge (x = 0 \vee x = 1) \wedge same(\{y\}) && (16.1)\\
& \vee \big[pc = l_1 \wedge x = 0 \wedge pc' = l_2 \wedge same(\{x, y\}) && (16.2)\\
& \quad \vee pc = l_1 \wedge \neg\, x = 0 \wedge pc' = l_3 \wedge same(\{x, y\}) && (16.3)\\
& \quad \vee pc = l_2 \wedge pc' = m^e \wedge y' = 5 \wedge same(\{x\}) && (16.4)\\
& \quad \vee pc = l_3 \wedge pc' = m^e \wedge y' = x \wedge same(\{x\})\big] && (16.5)
\end{aligned}
$$

Note how the first disjunct (line 12.1) captures the non-deterministic assignment and the second (spanning lines 12.2 to 12.5) formalizes the effect of the **if** statement, where the formulas in lines 12.2 and 12.3 advance the program counter into appropriate branch depending on whether $x = 0$ holds or not.

We can now define the transition relation of the Kripke structure representing C:

$$\mathcal{R}(V, pc, V, pc') \quad = \quad T(m, C, m^e) \vee (pc = m^e \wedge pc' = pc \wedge same(V))$$

An example of a path in this Kripke structure is

$$(m, d_x, d_y)(l_1, 1, d_y)(l_3, 1, d_y)(m^e, 1, 1)(m^e, 1, 1)\ldots$$

where the elements in each triple denote the values of pc, x, and y respectively, and d_x and d_y represent the intial values of x and y respectively.

Note how this path (and any path of the Kripke structure) satisfies the formula $\mathcal{R}(V, pc, V, pc')$ defined above.

Exercise 16.2.1. Consider the two following programming language constructs

$$\textbf{repeat } C \textbf{ until } b$$

and

$$\textbf{case } b_1 : C_1 \;\mid\; \ldots \;\mid\; b_{n-1} : C_{n-1} \;\mid\; true : C_n \textbf{ end}.$$

Assume that **repeat** has the standard, expected semantics. The **case** statement evaluates the boolean expressions b_i in order of ascending indices and executes the first C_j for which b_j evaluates to true.

1. Extend the labeling function $(\_)^L$ to the two constructs above.
2. Extend the translation function T to the labeled versions of the two constructs above.

16.2.4 Example: Modeling concurrent programs

We now extend the translation described above from sequential to concurrent programs. Before we do this, we note that concurrent systems can be distinguished

- based on how the concurrently executing program units interact and exchange information. There are typically three choices: *shared variables (memory)*, *message passing*, or both.

- based on whether or not the units execute *synchronously* or *asynchronoulsy*. Synchonous means that executions are not completely independent and coupled in some way, while asynchrous means that no such coupling exists.

Our focus will be on shared variable concurrent systems. In this context, synchrony typically means that concurrent program units execute in lock step, that is, they all make execution steps together and at the same time. An example of that are synchronous circuits (see Section 16.2.2) or some synchronous languages such as Lustre.

Units in asynchronously executing shared variable concurrent programs execute independently unless specific synchronization statements force them to block and wait until, e.g., some condition holds. Examples of languages supporting asynchronous shared variable concurreny include, C/C++ and Java. Since this paradigm is most important to us, we will start with showing how it can be captured using Kripke structures. The treatment of synchronous shared variable concurrency is similar to that of synchronous circuits in Section 16.2.2 and will not be discussed further. In the following, the term *concurrent* program refers to an asynchronously executing shared variable concurrent program.

Program C is said to be *concurrent* if it is of the form

$$\textbf{cobegin } C_1 \| \ldots \| C_n \textbf{ end}$$

where each C_i is a *process*, that is, a sequential program as described in the previous section. During the execution of a concurrent program, the processes C_i are executed in an asynchronous, interleaved fashion. Processes communicate by means of message passing or shared variables. Let V_i be the set of program variables used by process C_i. Note that we do not require that the sets V_i be disjoint. Let pc_i be the program counter of process C_i and let PC be the set of all program counters, that is, $PC = \{pc, pc_1, \ldots, pc_n\}$. The labeling function $(\_)^L$ is extended such that a concurrent program can occur as a statement in a sequential program. The function attaches a label to the entry point and exit point of each process. Note that the exit point of a process C_i will be different from the entry points of all other processes. Therefore, the exit points of processes must be labeled explicitly using the notation $l_i{:}C_i{:}l_i^e$.

- If C is a concurrent program **cobegin** $C_1 \| \ldots \| C_n$ **end**, then

$$C^L \quad = \quad \textbf{cobegin } l_1{:}C_1^L{:}l_1^e \| \ldots \| l_n{:}C_n^L{:}l_n^e \textbf{ end}.$$

Similarly, the translation function T is extended by the following clause.

- If C^L is a concurrent program **cobegin** $l_1{:}C_1^L{:}l_1^e \| \ldots \| l_n{:}C_n^L{:}l_n^e$ **end**, then

$$\begin{aligned} & T(l, \textbf{cobegin } l_1{:}C_1^L{:}l_1^e \| \ldots \| l_n{:}C_n^L{:}l_n^e \textbf{ end}, l^e) \\ = \quad & \Big(pc = l \wedge (\textstyle\bigwedge_{i=1}^n pc_i' = l_i) \wedge pc' = \bot \wedge same(V)\Big) && (16.6) \\ & \vee \Big(pc = \bot \wedge (\textstyle\bigwedge_{i=1}^n pc_i = l_i^e \wedge pc_i' = \bot) \wedge pc' = l^e \wedge same(V)\Big) && (16.7) \\ & \vee \Big(\textstyle\bigvee_{i=1}^n T(l_i, C_i^L, l_i^e) \wedge same(V \setminus V_i) \wedge same(PC \setminus \{pc_i\})\Big). && (16.8) \end{aligned}$$

The first disjunct (12.6) describes the initialization of the concurrent processes: A transition is made from the entry point of the **cobegin** statement to the entry points of each of the processes. The second disjunct (12.7) describes the termination of the concurrent program: A transition is made from the exit points of the processes to the exit point of the **cobegin** statement; this transition will only be executed if all the processes terminate. The third disjunct (12.8) describes the interleaved execution of the n concurrent processes: The formula for the transition relation of process C_i is added as a conjunct to $same(V \setminus V_i) \wedge same(PC \setminus \{pc_i\})$; these last two conjuncts ensure that a transition in process C_i^L can change only the variables in V_i and that only one process can make a transition at any time.

Non-determinism and fairness Before we move on, we return to the notion of non-determinism already discussed in Section 16.1. So far, non-deterministic assignments were the only source of non-determinism in our Kripke structures. However, the concurrent execution of processes as defined above adds concurrency as a second source of non-determinism: In general, in a given state s, more than one process will be able to make an execution step, and correspondingly the third disjunct (2.8) will ensure that s has a different successor state for each process that can execute.

We also note that the definition above allows *unfair* executions, i.e., executions along which, from some state on, only one process ever gets to execute. In other words, the corresponding Kripke structure contains paths along which, from some state on, only one process executes.

Assuming that the Kripke structure models a concurrent program in a language such as C/C++ or Java, these unfair paths can be considered "unrealistic" because any reasonable runtime scheduler will ensure that all processes eventually get a turn and that no process is ignored forever. In the next two chapters, we will see how fairness constraints can be used to remove unfair paths from the Kripke structure.

Exercise 16.2.2. Consider the following programming language construct

$$\textbf{coroutinebegin } C_1 \parallel C_2 \textbf{ end}.$$

The statement executes C_1 and C_2 in an alternating fashion, starting with C_1. More precisely, first, C_1 is allowed to make a transition, then C_2, then C_1 again, then C_2 again etc. If C_1 terminates before C_2 does, C_2 is executed alone. Similarly, for C_2. The statement terminates as soon as C_1 and C_2 have terminated.

1. Extend the labeling function $(\_)^L$ to the coroutine statement.
2. Extend the translation function T to the labeled version of the coroutine statement.

Shared variables

Remember that the sets of variables used by each process in a concurrent program may overlap, that is, a variable v may be used by more than one process. In that case, v is called *shared* and the program is called a *shared variable concurrent program*. The synchronization statements **await**, **lock**, and **unlock** control the access to shared variables. Statement **await** b blocks until the current state satisfies b. Given a boolean variable v, **lock**(v) blocks until v is 0 and then sets it to 1. Statement **unlock**(v) simply sets v to 0. We extend the translation T appropriately.

- If C_i^L is the **await** statement **await** b, then

$$\begin{aligned} T(l, \textbf{await } b, l^e) \quad &= \quad \big(pc_i = l \wedge pc_i' = l \wedge \neg b \wedge same(V_i)\big) \\ &\quad \vee \big(pc_i = l \wedge pc_i' = l^e \wedge b \wedge same(V_i)\big). \end{aligned}$$

- If C_i^L is the **lock** statement **lock**(v), then

$$\begin{aligned} &T(l, \textbf{lock}(v), l^e) \\ &\quad = \quad \big(pc_i = l \wedge pc_i' = l \wedge v = 1 \wedge same(V_i)\big) \\ &\qquad \vee \big(pc_i = l \wedge pc_i' = l^e \wedge v = 0 \wedge v' = 1 \wedge same(V_i \backslash \{v\})\big). \end{aligned}$$

- If C_i^L is the **unlock** statement **unlock**(v), then

$$T(l, \textbf{unlock}(v), l^e) \quad = \quad \big(pc_i = l \wedge pc_i' = l^e \wedge v' = 0 \wedge same(V_i \backslash \{v\})\big).$$

Example 16.2.3. We present a small example to illustrate the treatment of concurrent programs. Consider the following concurrent program:

$$C \quad = \quad m\text{:} \quad \textbf{cobegin } C_0 \| C_1 \textbf{ end } :m^e$$

where

$$\begin{aligned} C_0 \quad = \quad & l_0\text{:} \quad \textbf{while } tt \textbf{ do} \\ & \qquad nc_0\text{: } \textbf{await } turn = 0\textbf{;} \\ & \qquad cr_0\text{: } turn \textbf{:=} 1 \\ & \quad \textbf{end} \\ & :l_0^e \\ C_1 \quad = \quad & l_1\text{:} \quad \textbf{while } tt \textbf{ do} \\ & \qquad nc_1\text{: } \textbf{await } turn = 1\textbf{;} \\ & \qquad cr_1\text{: } turn \textbf{:=} 0 \\ & \quad \textbf{end} \\ & :l_1^e \end{aligned}$$

The program counter pc of the program C takes on only three values: m, the entry point of C; m^e, the exit point of C; and $\perp$, the value of pc when C_0 and C_1 are active. Each process C_i has a program counter pc_i that ranges over

the labels l_i, l_i^e, nc_i, cr_i, and $\perp$. The two processes share a single variable *turn*. Thus, $V = V_0 = V_1 = \{turn\}$ and $PC = \{pc, pc_0, pc_1\}$. When the value of the program counter of a process C_i is cr_i, the process is in its critical region. Both processes are not allowed to be in their critical regions at the same time. When the value of the program counter is nc_i, the process is in its non-critical region. In this case it waits until $turn = i$ in order to gain exclusive entry into the critical region. The initial states of C are described by the formula

$$\mathcal{S}_0(V, PC) \quad = \quad pc = m \wedge pc_0 = \perp \wedge pc_1 = \perp$$

Note that no restriction is imposed on the value of *turn*. Thus, it may initially be either 0 or 1. Applying the translation procedure T we obtain the formula for the transition relation of C, $\mathcal{R}(V, PC, V', PC')$, which is the disjunction of the following four formulas:

- $pc = m \wedge pc_0' = l_0 \wedge pc_1' = l_1 \wedge pc' = \perp \wedge same(V)$
- $pc_0 = l_0^e \wedge pc_1 = l_1^e \wedge pc' = m^e \wedge pc_0' = \perp \wedge pc_1' = \perp \wedge same(V)$
- $T(l_0, C_0, l_0^e) \wedge same(V \backslash V_0) \wedge same(PC \backslash \{pc_0\})$, which is equivalent to $T(l_0, C_0, l_0^e) \wedge same(\{pc, pc_1\})$.
- $T(l_1, C_1, l_1^e) \wedge same(V \backslash V_1) \wedge same(PC \backslash \{pc_1\})$, which is equivalent to $T(l_1, C_1, l_1^e) \wedge same(\{pc, pc_0\})$.

For each process C_i, $T(l_i, C_i, l_i^e)$ is the disjunction of:

- $pc_i = l_i \wedge pc_i' = nc_i \wedge tt \wedge same(turn)$
- $pc_i = nc_i \wedge pc_i' = cr_i \wedge turn = i \wedge same(turn)$
- $pc_i = cr_i \wedge pc_i' = l_i \wedge turn' = (i + 1) \mod 2$
- $pc_i = nc_i \wedge pc_i' = nc_i \wedge turn \neq i \wedge same(turn)$
- $pc_i = l_i \wedge pc_i' = l_i^e \wedge ff \wedge same(turn)$

We note that the last disjunct will never hold, because it contains *ff* and is, thus, contradictory. Since this disjunct captures the termination of program C, this means that C never terminates (as expected).

The Kripke structure in Figure 16.3 is derived from the formulas $\mathcal{S}_0$ and $\mathcal{R}$ as described in the previous section. Examining this structure, we can make the following observations:

- *Mutual exclusion:* We can see that the processes will never be in their critical region at the same time. Thus, the program guarantees the mutual exclusion property.
- *Non-determinism:* Due to the concurrency, the Kripke structure is non-deterministic; once the two processes have started, every reachable state has two successors due to the fact that each of the two processes can execute from that state.

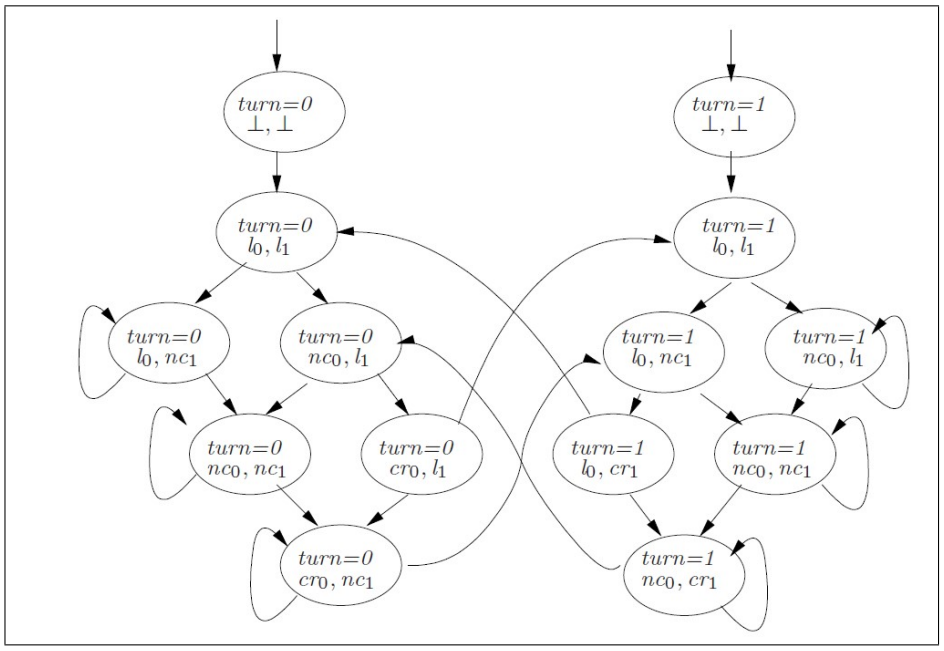


Figure 16.3: Reachable states of the Kripke structure for the mutual exclusion example.

- *Fairness:* Unfair executions are possible. An example of such an unfair path is

$$(0, \perp, \perp)(0, l_0, l_1)(0, l_o, nc_1)(0, l_o, nc_1)\ldots$$

  representing an execution in which process C_0 is never executed and process C_1 gets stuck at its **await** statement forever because $turn = 0$. As mentioned before, these unfair paths are, in some sense, unrealistic, and, as we will see, model checking allows for them to be removed from consideration via fairness constraints.

To conclude this chapter, we illustrate the relationship between the various kinds of systems and representations presented in Figure 16.4.

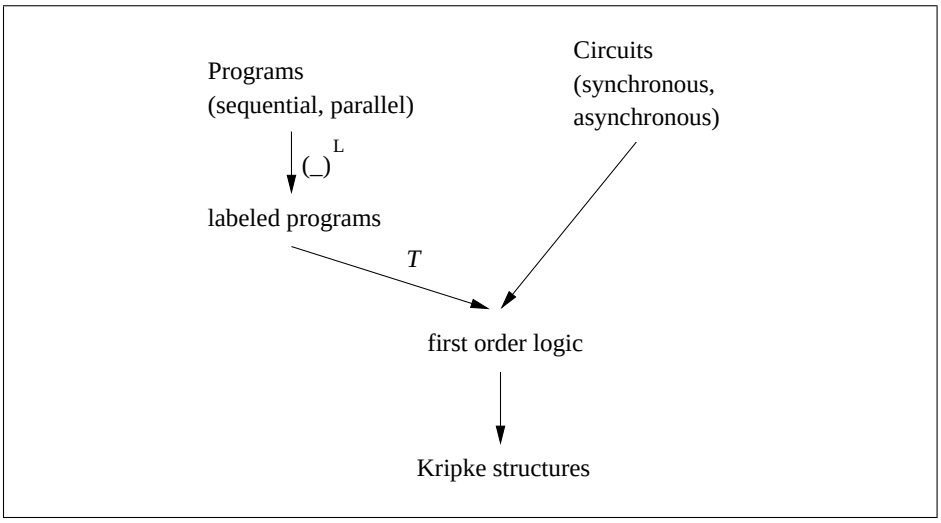


Figure 16.4: Relationships between the various representations

Chapter 17

CTL model checking

CTL model checking uses a temporal logic called computation tree logic (CTL) as specification logic.

17.1 CTL (Weeks 9-10)

17.1.1 Syntax

CTL formulas are defined by the following BNF

$$\begin{array}{rcl} \varphi & ::= & \mathit{ff} \mid \mathit{tt} \mid p \mid (\neg\, \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \\ & & \mathbf{AX}\, \varphi \mid \mathbf{EX}\, \varphi \mid \mathbf{AG}\, \varphi \mid \mathbf{EG}\, \varphi \mid \mathbf{AF}\, \varphi \mid \mathbf{EF}\, \varphi \mid \\ & & \mathbf{A}\big[\varphi_1\ \mathbf{U}\ \varphi_2\big] \mid \mathbf{E}\big[\varphi_1\ \mathbf{U}\ \varphi_2\big] \end{array}$$

where p is an atomic proposition, that is, $p \in AP$. Note that every temporal connective is a pair of letters. The first one ('**A**' or '**E**') can be thought of as quantification over the set of paths from the current state. The second one ('**X**','**G**','**F**', or '**U**') can be thought of as quantification over the states in a selected path. Each pair of letters thus represents a nested quantification, the first one over paths, the second over states. The following table indicates the intuitive meaning of each pair.

$\mathbf{AX}\ \varphi$	"Along all paths, in the next state, φ holds"
$\mathbf{EX}\ \varphi$	"Along at least one path, in the next state, φ holds"
$\mathbf{AG}\ \varphi$	"Along all paths, in all future states, φ holds"
	"Along all paths, φ holds globally"
$\mathbf{EG}\ \varphi$	"Along at least one path, in all future states, φ holds"
	"Along at least one path, φ holds globally"
$\mathbf{AF}\ \varphi$	"Along all paths, in some future state, φ holds", or
	"Along all paths, φ holds eventually"
$\mathbf{EF}\ \varphi$	"Along at least one path, in some future state, φ holds", or
	"Along at least one path, φ holds eventually"
$\mathbf{A}[\varphi_1\ \mathbf{U}\ \varphi_2]$	"Along all paths, φ_1 holds at least until φ_2 does"
$\mathbf{E}[\varphi_1\ \mathbf{U}\ \varphi_2]$	"Along at least one path, φ_1 holds at least until φ_2 does"

The binding priorities of the new connectives generalize the ones for propositional logic.

$\neg, \mathbf{AX}, \mathbf{EX}, \mathbf{AG}, \mathbf{EG}, \mathbf{AF}, \mathbf{EF}$	bind most tightly
$\wedge, \vee$	
$\rightarrow$	
$\leftrightarrow, \mathbf{AU}, \mathbf{EU}$	bind least tightly

17.1.2 Semantics

Formulas are interpreted over Kripke structures. Given a Kripke structure M, a state s, and a CTL formula φ, the satisfaction relation $(M, s) \models \varphi$ is defined as follows:

$(M, s) \models tt$

$(M, s) \models p$ if $p \in L(s)$

$(M, s) \models \neg\, \varphi_1$ if not $(M, s) \models \varphi_1$

$(M, s) \models \varphi_1 \wedge \varphi_2$ if $(M, s) \models \varphi_1$ and $(M, s) \models \varphi_2$

$(M, s) \models \varphi_1 \vee \varphi_2$ if $(M, s) \models \varphi_1$ or $(M, s) \models \varphi_2$

$(M, s) \models \varphi_1 \rightarrow \varphi_2$ if not $(M, s) \models \varphi_1$ or $(M, s) \models \varphi_2$

$(M, s) \models \mathbf{AX}\ \varphi$ if for all s' such that $R(s, s')$ we have $(M, s') \models \varphi$

$(M, s) \models \mathbf{EX}\ \varphi$ if for some s' such that $R(s, s')$ we have $(M, s') \models \varphi$

$(M, s) \models \mathbf{AG}\ \varphi$ if for all paths $s_1 s_2 s_3 \ldots$ in M such that $s = s_1$ we have $(M, s_i) \models \varphi$ for all $i \geq 1$

$(M, s) \models \mathbf{EG}\ \varphi$ if for some path $s_1 s_2 s_3 \ldots$ in M such that $s = s_1$ we have $(M, s_i) \models \varphi$ for all $i \geq 1$

$(M, s) \models \mathbf{AF}\ \varphi$ if for all paths $s_1 s_2 s_3 \ldots$ in M such that $s = s_1$ there exists $i \geq 1$ such that $(M, s_i) \models \varphi$

$(M, s) \models \mathbf{EF}\ \varphi$ if for some path $s_1 s_2 s_3 \ldots$ in M such that $s = s_1$

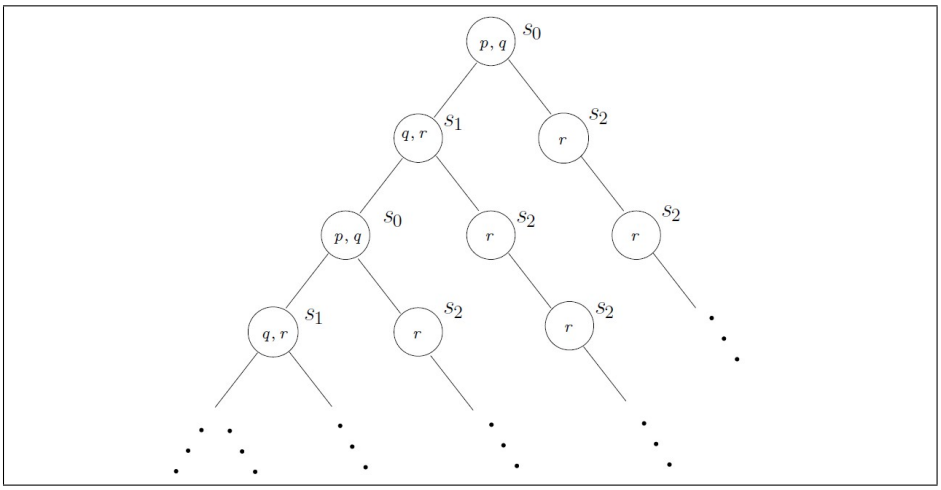


Figure 17.1: Unwinding of the system in Figure 16.1 into a computation tree from state s_0.

$$
\begin{array}{lcl}
 & & \text{there exists } i \geq 1 \text{ such that } (M, s_i) \models \varphi \\
(M, s) & \models & \mathbf{A}\big[\varphi_1 \ \mathbf{U} \ \varphi_2\big] \text{ if for all paths } s_1 s_2 s_3 \ldots \text{ in } M \text{ such that } s = s_1 \\
 & & \text{there exists some } i \geq 1 \text{ such that } (M, s_i) \models \varphi_2, \text{ and} \\
 & & \text{for all } 1 \leq j < i, \text{ we have } (M, s_j) \models \varphi_1 \\
(M, s) & \models & \mathbf{E}\big[\varphi_1 \ \mathbf{U} \ \varphi_2\big] \text{ if for some path } s_1 s_2 s_3 \ldots \text{ in } M \text{ such that } s = s_1 \\
 & & \text{there exists some } i \geq 1 \text{ such that } (M, s_i) \models \varphi_2, \text{ and} \\
 & & \text{for all } 1 \leq j < i, \text{ we have } (M, s_j) \models \varphi_1
\end{array}
$$

17.1.3 Computation trees

The clauses involving propositional connectives only offer no surprises. To illustrate the temporal connectives, it is useful to unwind a Kripke structure into a so-called *computation tree*.

Given Kripke structure M and an initial state s, the computation tree of (M, s) contains all paths in (M, s) in a tree-like format, meaning that there are no cycles and every node except the initial node has exactly one predecessor. The advantage of this representation is that the computation paths of a system can be directly read off. If M is deterministic (i.e., not non-deterministic), the tree of (M, s) will consist of a single infinite path.

For instance, Figure 17.1 contains the beginning of the computation tree of the system in Figure 16.1.

The computation trees in Figures 17.2, 17.3, 17.4, and 17.5, illustrate the four unary temporal connectives. More precisely, for each connective we give one or two examples of a system in form of a computation tree whose initial state satisfies the formula built using that connective.

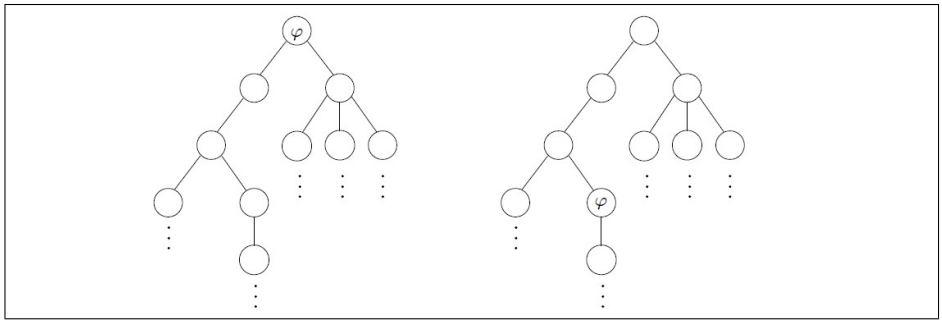


Figure 17.2: Beginnings of two systems whose initial states satisfy **EF** φ.

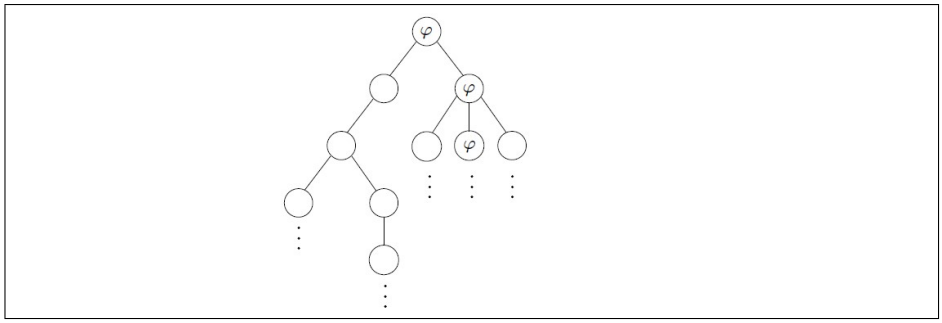
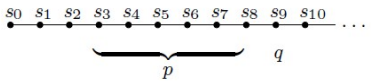


Figure 17.3: Beginning of a system whose initial state satisfies **EG** φ.

To illustrate the until operator, consider the following computation path.



Each of the states s_3 to s_9 satisfies $\big[p \ \mathbf{U} \ q\big]$, while the states s_0 to s_2 do not.

Safety and liveness properties: a fundamental distinction

Software properties in general and CTL formulas in particular can be partitioned into two categories:

- *Safety properties:* Informally, a safety property is a property that states that "nothing bad ever happens" [Lam77]. For instance, deadlock freedom is a safety property. In CTL, safety properties are thus related to the temporal connectives **AG** and **EG**. Examples of safety properties in CTL include: **AG** $x \neq 0$ and **EG** $(\mathit{doorOpen} \rightarrow \mathit{fanOff})$.

  A safety property is violated iff the system has a (finite or infinite) execution $s_0, s_1, s_2 \ldots$ with state s_i such that "something bad has happened

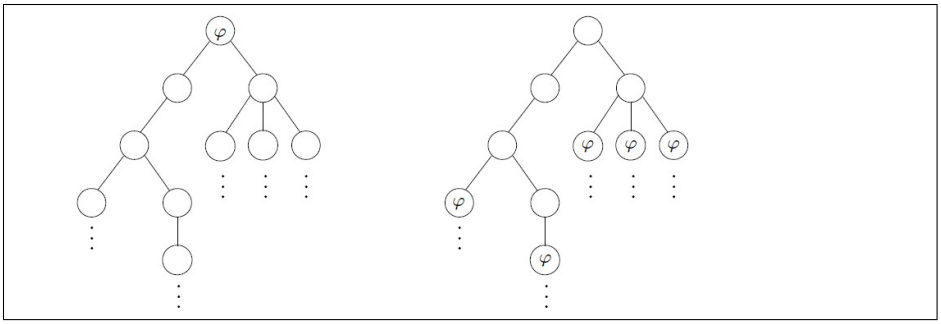


Figure 17.4: Beginnings of two systems whose initial states satisfy **AF** φ.

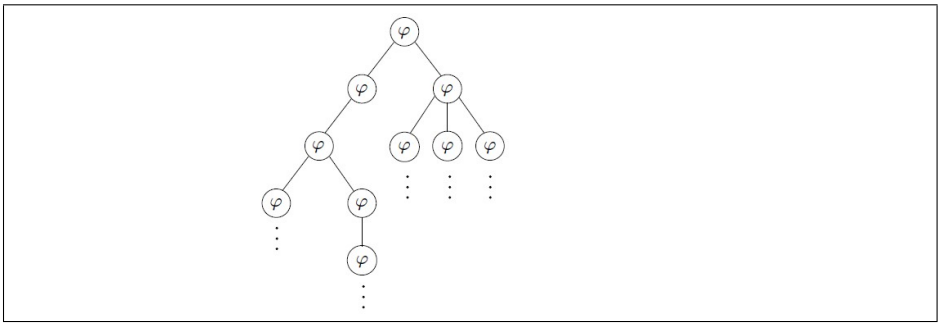


Figure 17.5: Beginning of a system whose initial state satisfies **AG** φ.

in s_i". Then, the sequence of states $s_0, s_1, \ldots, s_i$ is a counter example. Safety properties, therefore, always have finite counter examples.

- *Liveness properties:* In contrast, a liveness property expresses that "something good will eventually happen". As before, "eventually" here means after an unknown, arbitrary but finite number of steps. For instance, termination is a liveness property. In CTL, liveness properties are related to the temporal connectives **AF** and **EF**. Examples of liveness properties in CTL include: **AF** $x \neq 0$ and **AG** (*request* $\rightarrow$ **AF** *granted*).

  A liveness property is violated iff the system has an infinite execution $s_0, s_1, s_2 \ldots$ along which "the good thing never happens". In this case, the entire execution constitutes the counter example. In other words, liveness properties always have infinite counter examples.

It turns out that this classification is "exhaustive" in the sense that every property can be expressed through the combination of a safety and a liveness property [AS85]. In other words, when reasoning about the correctness of a system, safety and liveness properties are the only kinds of properties you need to worry about.

Schemas of useful CTL formulas

We list a few examples for the kinds of formulas that are often used.

- "It is possible to become super user":

$$\mathbf{EF}\ \mathit{superUser}$$

- "It is always possible to become super user":

$$\mathbf{AG}\ \mathbf{EF}\ \mathit{superUser}$$

- "A request for some resource will always eventually be acknowledged":

$$\mathbf{AG}\ (\mathit{requested}\ \rightarrow \mathbf{AF}\ \mathit{acknowledged})$$

- "Along every computation path, enabled always holds eventually":

$$\mathbf{AG}\ \mathbf{AF}\ \mathit{enabled}$$

  Note that this means that *enabled* will hold infinitely often along every computation path.

- "A system will never deadlock":

$$\mathbf{AG}\ \neg\ \mathit{deadlocked}$$

- "From every hypertext page it is always possible to get to a hypertext page named 'Home' ":

$$\mathbf{AG}\ \mathbf{EF}\ \mathit{name{=}'Home'}$$

- "When picking up the phone it is possible to never receive a dial tone":

$$\mathbf{AG}\ \big(\mathit{rcvPickedUp} \rightarrow \mathbf{EG}\ \neg\ \mathit{dialTone}\big)$$

- "An upwards traveling elevator at the second floor will not change its direction when it has passengers wishing to go the fifth floor":

$$\mathbf{AG}\ \Big(\mathit{floor{=}2} \wedge \mathit{direction{=}up} \wedge \mathit{Button5Pressed} \rightarrow \mathbf{A}\big[\mathit{direction{=}up}\ \mathbf{U}\ \mathit{floor{=}5}\big]\Big)$$

Equivalences

We already know that conjunction and disjunction are dual to each other, that is, $(\varphi \vee \psi) \leftrightarrow \neg\,(\neg\,\varphi \wedge \neg\,\psi)$. Similarly, universal quantification and existential quantification in predicate logic are dual, that is, $(\exists x.\varphi) \leftrightarrow \neg\,(\forall x.\neg\,\varphi)$. It turns out that the temporal operators **AG** and **EF** are also dual to each other.

$$\begin{array}{rcl} \neg\,\mathbf{AG}\;\varphi & \leftrightarrow & \mathbf{EF}\;\neg\,\varphi \\ \neg\,\mathbf{EF}\;\varphi & \leftrightarrow & \mathbf{AG}\;\neg\,\varphi \end{array}$$

The next state operator X is its own dual.

$$\neg\,\mathbf{AX}\;\varphi \quad \leftrightarrow \quad \mathbf{EX}\;\neg\,\varphi$$

A path π contains at least one state in which φ holds if and only if tt holds along π until φ does. Consequently, the eventuality operator **F** can be expressed in terms of the until operator **U**.

$$\begin{array}{rcl} \mathbf{AF}\;\varphi & \leftrightarrow & \mathbf{A}\big[tt\;\mathbf{U}\;\varphi\big] \\ \mathbf{EF}\;\varphi & \leftrightarrow & \mathbf{E}\big[tt\;\mathbf{U}\;\varphi\big] \end{array}$$

Finally, the operator **AU** can be expressed in terms of negation, disjunction, **EU**, and **EG**.

$$\mathbf{A}\big[\varphi\;\mathbf{U}\;\psi\big] \quad \leftrightarrow \quad \neg\,\Big(\mathbf{E}\big[\neg\,\psi\;\mathbf{U}\;(\neg\,\varphi \wedge \neg\,\psi)\big] \vee \mathbf{EG}\neg\,\psi\Big)$$

These equations indicate that there is redundancy between the temporal connectives. We can restrict our attention to an adequate set of connectives[1] to remove this kind of redundancy and to identify minimal sets of connectives from which all other connectives can be obtained.

Definition 17.1.1. Given a logic L, we say that a set C of connectives of L is *adequate* for L, if all connectives in L can be expressed in terms of connectives in C.

Example 17.1.1. 1. The set of connectives $\{\neg\,, \wedge\}$ is adequate for propositional logic. To see this, consider the following equivalences:

- $(P \vee Q) \leftrightarrow (\neg\,(\neg\,P \wedge \neg\,Q))$
- $(P \rightarrow Q) \leftrightarrow (\neg\,P \vee Q)$
- $(P \leftrightarrow Q) \leftrightarrow ((P \rightarrow Q) \wedge (Q \rightarrow P))$

2. The set of connectives $\{\neg\,, \vee, \forall\}$ is adequate for predicate logic.

Exercise 17.1.1.

1. Show that the set $\{\mathbf{AU}, \mathbf{EU}, \mathbf{EX}\}$ is adequate for all temporal connectives in CTL, that is, express the remaining temporal connectives in terms of negation and **AU**, **EU**, and **EX**.

[1] As discussed during the logic review in Section 3.2.3

2. Express $\mathbf{A}\big[p\ \mathbf{U}\ q\big]$ in terms of $\neg\, p$, $\neg\, q$, $\wedge$, $\vee$, $\neg$, **EU**, and **EG**.

There are lots of adequate sets of connectives for CTL. The following theorem singles out an adequate set that we will use in Section 17.3 when constructing the model checking algorithm.

Theorem 12. The set of operators *false*, $\neg$, $\wedge$, **EX**, **AF**, and **EU** are adequate for all connectives in CTL.

We conclude this section with a final list of equations.

$$\begin{aligned}
\mathbf{AG}\ \varphi &\leftrightarrow \varphi\ \wedge \mathbf{AX}\ \mathbf{AG}\ \varphi \\
\mathbf{EG}\ \varphi &\leftrightarrow \varphi\ \wedge \mathbf{EX}\ \mathbf{EG}\ \varphi && (17.1)\\
\mathbf{AF}\ \varphi &\leftrightarrow \varphi\ \vee \mathbf{AX}\ \mathbf{AF}\ \varphi && (17.2)\\
\mathbf{EF}\ \varphi &\leftrightarrow \varphi\ \vee \mathbf{EX}\ \mathbf{EF}\ \varphi \\
\mathbf{A}\big[\varphi\ \mathbf{U}\ \psi\big] &\leftrightarrow \psi \vee \big(\varphi \wedge \mathbf{AX}\ \mathbf{A}\big[\varphi\ \mathbf{U}\ \psi\big]\big) \\
\mathbf{E}\big[\varphi\ \mathbf{U}\ \psi\big] &\leftrightarrow \psi \vee \big(\varphi \wedge \mathbf{EX}\ \mathbf{E}\big[\varphi\ \mathbf{U}\ \psi\big]\big) && (17.3)
\end{aligned}$$

The equation for $\mathbf{AG}\ \varphi$ captures the fact that $\mathbf{AG}\ \varphi$ holds in the current state s if and only if

- φ holds in s, and
- $\mathbf{AX}\ \mathbf{AG}\ \varphi$ holds in s, that is, for all possible next states s' $\mathbf{AG}\ \varphi$ holds in s'.

The equation for $\mathbf{AF}\ \varphi$, on the other hand, expresses that $\mathbf{AF}\ \varphi$ holds in the current state s if and only if

- either φ holds in s, or
- $\mathbf{AX}\ \mathbf{AF}\ \varphi$ holds in s, that is, for all possible next states s', $\mathbf{AF}\ \varphi$ holds in s'.

The computation trees in Figure 17.4 illustrate both cases. Finally, the equation for $\mathbf{A}\big[\varphi\ \mathbf{U}\ \psi\big]$ expresses that that $\mathbf{A}\big[\varphi\ \mathbf{U}\ \psi\big]$ holds in the current state s if and only if

- ψ holds in s, or
- φ holds in s and $\mathbf{AX}\ \mathbf{A}\big[\varphi\ \mathbf{U}\ \psi\big]$ holds in s, that is, for all possible next states s', $\mathbf{A}\big[\varphi\ \mathbf{U}\ \psi\big]$ holds in s'.

Similar arguments apply to the equations for $\mathbf{EF}\ \varphi\ \mathbf{EG}\ \varphi$, and $\mathbf{E}\big[\varphi\ \mathbf{U}\ \psi\big]$ respectively.

Notice how each equation describes each connective in terms of itself. Intuitively, for each equation, the formula on the right is obtained by "unwinding" the formula on the left once — a process not unlike, for instance, unwinding a **while** loop into the sequential composition of a conditional and the same loop:

$$\textbf{while}\ b\ \textbf{do}\ C \quad = \quad \textbf{if}\ b\ \textbf{then}\ C\,;\textbf{while}\ b\ \textbf{do}\ C\ \textbf{end}$$

Indeed, just like in denotational semantics, where the behaviour of a **while** loop is described in terms of all its unwindings, the meaning of each of the temporal connectives can be described in terms of all its unwindings. This description is called *fixed point semantics*.

17.2 Example: Mutual exclusion (Week 10)

At this point, we have discussed state machines and CTL and thus have both inputs to a CTL model checker in place. Before we give more detail on how precisely the model checker works, let us look at an example showing how all the notions introduced so far fit together.

Suppose the processes C_i in the concurrent program

$$C \quad = \quad \textbf{cobegin } C_0 \| \ldots \| C_{n-1} \textbf{ end}$$

all share a resource, such as a printer, a database or a file on a disk. To ensure consistency of the resource, it may be necessary to prevent multiple processes from updating the resource simultaneously. To solve this problem, we identify so-called *critical sections* and *non-critical sections* in the code of each process and restrict access to the resource using shared variables and synchronization statements such that at most one process is executing its critical region at any given time. This property is called *mutual exclusion*. We will assume that each process is labeled in the sense of Section 16.2.4 and that each process C_i has the following shape

$$\begin{array}{rcl} C_i & = & l_i\textbf{:while } \mathit{true} \textbf{ do} \\ & & \quad nc_i\textbf{:}\ \ C_{i,nc} \\ & & \quad cr_i\textbf{:}\ \ C_{i,cr} \\ & & \textbf{end}\textbf{:}l_i^e \end{array}$$

where the labels nc_i and cr_i indicate the non-critical and the critical sections of C_i respectively. Whereas the processes used in Example 16.2.3 had empty non-critical and critical sections, we now want our model to be more realistic and drop that assumption. However, we have to assume that all critical sections $C_{i,cr}$ always terminate. The non-critical sections $C_{i,nc}$, however, may or may not terminate. Moreover, since the state machine corresponding to C must be finite for model checking to be applicable, we assume that all critical and non-critical sections have a finite state space.

We want to modify each process C_i such that access to the critical sections is mutually exclusive. The main idea is to place the critical section of each process between an *entry* and an *exit protocol*. The entry protocol in process C_i will protect the critical section $C_{i,cr}$ by checking if other processes are currently executing their critical section. The exit protocol will notify the other processes of the termination of the critical section and possibly allow other processes to enter their critical section.

17.2.1 First attempt

Our first attempt starts where Example 16.2.3 left off. A new variable *turn* that ranges over the numbers from 0 to $n-1$ is used to indicate which process will be allowed to enter its critical region. Let $\oplus$ denote addition modulo n.

$$
\begin{array}{rcl}
C_{i,1} & = & \textbf{while } \mathit{true} \textbf{ do} \\
 & & \quad nc_i\colon\ C_{i,nc}\textbf{;} \\
 & & \quad en_i\colon\ \textbf{await } \mathit{turn} = i\textbf{;} \\
 & & \quad cr_i\colon\ C_{i,cr}\textbf{;} \\
 & & \quad ex_i\colon\ \mathit{turn} := \mathit{turn} \oplus 1 \\
 & & \textbf{end}
\end{array}
$$

The above algorithm is also called *round robin* algorithm. Before we use a CTL model checker to verify that the modified system satisfies mutual exclusion, we need to express the mutual exclusion in CTL.

- If we're dealing with only two processes, the formula

  $$\mathit{mutex} \quad = \quad \mathbf{AG}\ \neg\ (pc_0 = cr_0 \wedge pc_1 = cr_1)$$

  expresses that they cannot be in their critical section at the same time. This formula generalizes to more than two processes as follows by requiring that every reachable state is such that there is no pair of processes C_i and C_j such that $j \neq i$ and C_i and C_j are both in their critical region.

  $$\mathit{mutex} \quad = \quad \mathbf{AG}\ \neg\ \bigvee\nolimits_{i,j\in\{0,\ldots,n-1\}} (i \neq j \wedge pc_i = cr_i \wedge pc_j = cr_j)$$

  which is, by the way, equivalent to

  $$\mathit{mutex} \quad = \quad \neg\ \mathbf{EF}\ \bigvee\nolimits_{i,j\in\{0,\ldots,n-1\}} (i \neq j \wedge pc_i = cr_i \wedge pc_j = cr_j)$$

  and

  $$\mathit{mutex} \quad = \quad \mathbf{AG}\ \bigwedge\nolimits_{i,j\in\{0,\ldots,n-1\}} (i = j \vee pc_i \neq cr_i \vee pc_j \neq cr_j)$$

After this modification the execution of the critical sections is indeed mutually exclusive. More precisely, if M_C is the Kripke structure corresponding to C for some fixed n, and s is an arbitrary initial state of M_C, then

$$(M_C, s) \models \mathit{mutex}$$

holds.

Unfortunately, this solution suffers a drawback. If the execution of the non-critical section $C_{i,nc}$ of process i never terminates, process $i \oplus 1$ will never get permission to enter its critical section. In other words, a protocol must satisfy more properties than mutual exclusion to be considered a good solution. Thus, before we proceed, let us collect the other properties that we want the protocol to satisfy. It turns out that there are two more properties besides mutual exclusion.

- **Eventual entry** Whenever any process wants to enter its critical section, it will eventually be permitted to do so.

$$evtlEntry \quad = \quad evtlEntry_0 \wedge \ldots \wedge evtlEntry_{n-1}$$

where $evtlEntry_i = \mathbf{AG}\ (pc_i = en_i \rightarrow \mathbf{AF}\ pc_i = cr_i)$.

- **Deadlock freedom** The system never deadlocks, more precisely, it is never the case that all processes get stuck forever in their entry protocols.

$$noDeadlock \quad = \quad \mathbf{AG}\ \neg\ (blocked_0 \wedge \ldots \wedge blocked_{n-1})$$

where $blocked_i = \mathbf{AG}(pc_i = en_i)$. Note that this definition of deadlock is slightly different from "all processes are stuck because they are waiting for each other".

Exercise 17.2.1. What is the logical relationship (if any) between eventual entry and deadlock freedom?

17.2.2 Second attempt

The problem with our first attempt above is that a process can be given the right to enter its critical section without being interested in entering it. To remedy this, we introduce a boolean variable req_i which, when set, indicates that process i is interested in entering its critical section.

$$\begin{array}{rcl} C_{i,2} & = & \textbf{while } true \textbf{ do} \\ & & \quad nc_i\colon\ C_{i,nc}\textbf{;} \\ & & \quad en_{i,1}\colon\ req_i := tt\textbf{;} \\ & & \quad en_{i,2}\colon\ \textbf{await } turn = i\textbf{;} \\ & & \quad cr_i\colon\ C_{i,cr}\textbf{;} \\ & & \quad ex_{i,1}\colon\ req_i := ff\textbf{;} \\ & & \quad ex_{i,2}\colon\ turn := f(i) \\ & & \textbf{end} \end{array}$$

where

$$f(i) \quad = \quad \begin{cases} j, & j \text{ is smallest interested process, ie,} \\ & j \text{ is smallest } k \text{ for which } req_k = tt. \\ i, & \text{if there is no interested process.} \end{cases}$$

Unfortunately, this approach doesn't completely correct the problem encountered in the previous attempt. Consider for instance the 2-process system

$$C \quad = \quad turn := 0\ \textbf{; cobegin } req_0 := ff\ \textbf{;}\ C_{0,2}\ \|\ req_1 := ff\ \textbf{;}\ C_{1,2}\ \textbf{end}$$

where $C_{0,2}$ has a non-terminating non-cricital section and $C_{1,2}$ has an empty non-critical section, that is, $C_{(0,2),nc} = C_{(1,2),nc} = \textbf{skip}$. This system has an

execution that ends in the following trace

pc_0	req_0	pc_1	req_1	$turn$
nc_0	ff	nc_1	ff	0
nc_0	ff	$en_{1,1}$	ff	0
nc_0	ff	$en_{1,2}$	tt	0
nc_0	ff	$en_{1,2}$	tt	0
		$\vdots$		

Variable never gets set to point the interested process, because process $C_{0,2}$ never leaves its non-critical section. In other words, this system still does not satisfy eventual entry, that is,

$$(M_C, s) \not\models \mathit{evtlEntry}$$

where M_C models C above.

17.2.3 Third attempt

We abandon the idea of the single shared variable *turn* granting access. Instead, we start out very naively and allow process i to enter its critical region if there is no other interested process. For simplicity, we assume for the moment that we are dealing with 2 processes only.

$C_{i,3}$ = **while** *true* **do**
nc_i: $C_{i,nc}$**;**
$en_{i,1}$: req_i **:=** tt**;**
$en_{i,2}$: **await** $\neg\, req_{i\oplus 1}$**;**
cr_i: $C_{i,cr}$**;**
ex_i: req_i **:=** ff
end

This ensures mutual exclusion, but now the system can deadlock, that is,

$$(M_C, s) \not\models \mathit{noDeadlock}.$$

To see this, consider the execution below

pc_0	req_0	pc_1	req_1
nc_0	ff	nc_1	ff
nc_0	ff	$en_{1,1}$	ff
nc_0	ff	$en_{1,2}$	tt
$en_{0,1}$	ff	$en_{1,2}$	tt
$en_{0,2}$	tt	$en_{1,2}$	tt
		$\vdots$	

in which both processes move out of their non-critical sections immediately and express interest in entering their critical section at roughly the same time.

17.2.4 Fourth attempt

The problem with the above solution is that both process can execute their entry protocol at the same time and then get blocked at the **await** statement. To remedy this, we introduce a new variable *last*, that, intuitively, whenever both processes are blocked, "breaks the tie" between them and allows one of them to proceed. The resulting solution is called the *tie-breaker algorithm*, or also *Peterson's algorithm*. Let $x_1, x_2 := e_1, e_2$ denote a multiple assignment statement expressing that x_1 and x_2 are updated with the values of e_1 and e_2 respectively at exactly the same time in one atomic step.

$$
\begin{array}{lcl}
C_{i,4} & = & \textbf{while } \mathit{true} \textbf{ do} \\
 & & \quad nc_i\colon\ C_{i,nc}\textbf{;} \\
 & & \quad en_{i,1}\colon\ req_i, last := tt, i\textbf{;} \\
 & & \quad en_{i,2}\colon\ \textbf{await } \big(\neg\, req_{i\oplus 1} \vee last \neq i\big)\textbf{;} \\
 & & \quad cr_i\colon\ C_{i,cr}\textbf{;} \\
 & & \quad ex_i\colon\ req_i := ff \\
 & & \textbf{end}
\end{array}
$$

Note that variable *last* is shared. This attempt finally works. Mutual exclusion, eventual entry and deadlock freedom are all satisfied. Moreover, it also scales to arbitrary numbers of processes.

However, being the perfectionists that we are, we are still dissatisfied. The multiple assignment statement used in $C_{i,4}$ is hard, if not impossible, to implement on realistic machines. We will try to replace it with two simple assignments executed sequentially. This leaves us with two versions depending on which of the two assignments is executed first. Perhaps surprisingly, it turns out that these two versions are not identical.

$$
\begin{array}{lcl}
C'_{i,4} & = & \textbf{while } \mathit{true} \textbf{ do} \\
 & & \quad nc_i\colon\ C_{i,nc}\textbf{;} \\
 & & \quad en_{i,1}\colon\ req_i := tt\textbf{;} \\
 & & \quad en_{i,2}\colon\ last := i\textbf{;} \\
 & & \quad en_{i,3}\colon\ \textbf{await } \big(\neg\, req_{i\oplus 1} \vee last \neq i\big)\textbf{;} \\
 & & \quad cr_i\colon\ C_{i,cr}\textbf{;} \\
 & & \quad ex_i\colon\ req_i := ff \\
 & & \textbf{end}
\end{array}
$$

If the variable *last* is updated *after* req_i is set, we obtain a correct solution. Just like with $C_{i,4}$, all three properties are satisfied.

If, however, *last* is updated *before* req_i, the resulting protocol is incorrect.

$$
\begin{array}{lll}
C''_{i,4} & = & \textbf{while } \mathit{true} \textbf{ do} \\
 & & \quad nc_i\colon\ C_{i,nc}\textbf{;} \\
 & & \quad en_{i,1}\colon\ \mathit{last} := i\textbf{;} \\
 & & \quad en_{i,2}\colon\ \mathit{req}_i := \mathit{tt}\textbf{;} \\
 & & \quad en_{i,3}\colon\ \textbf{await } \big(\neg\, \mathit{req}_{i\oplus 1} \vee \mathit{last} \neq i\big)\textbf{;} \\
 & & \quad cr_i\colon\ C_{i,cr}\textbf{;} \\
 & & \quad ex_i\colon\ \mathit{req}_i := \mathit{ff} \\
 & & \textbf{end}
\end{array}
$$

Exercise 17.2.2. Which of the three properties would the system using $C''_{i,4}$ violate? Explain your answer by giving a counter example, that is, describe a violating computation path.

Exercise 17.2.3. The correctness of the above protocol is subject to the assumption that the critical sections of all processes always terminate. Why?

17.3 The CTL model checking algorithm (Week 11)

When defining the algorithm below we make use of the fact that the connectives ff, $\neg$, $\wedge$, **AF**, **EU**, and **EX** form an adequate set (Theorem 12). So, here's the algorithm:

Input: A Kripke structure $M = (S, S_0, R, L)$ and a CTL formula φ

Output: "Yes", if $(M, s_0) \models \varphi$ for all initial states $s_0 \in S_0$. "No", otherwise.

1. **Preprocessing** Translate φ into an equivalent formula φ' that contains only the adequate connectives mentioned in Theorem 12.

2. **Labeling** We compute the set of states that satisfy φ'. Label the states of M with the subformulas of φ' that are satisfied there, starting with the smallest subformulas. Suppose that ψ is a subformula of φ' and that the states satisfying all the immediate subformulas of φ' have already been labeled. The states to label with ψ are determined with the following case analysis. If ψ is

   - ff: No states are labeled with ff,
   - p: Label state s with p if $p \in L(s)$,
   - $\psi_1 \wedge \psi_2$: Label state s with $\psi_1 \wedge \psi_2$ if s is already labeled with ψ_1 and ψ_2,
   - $\neg\, \psi_1$: Label state s with $\neg\, \psi_1$ if s is not already labeled with ψ_1,

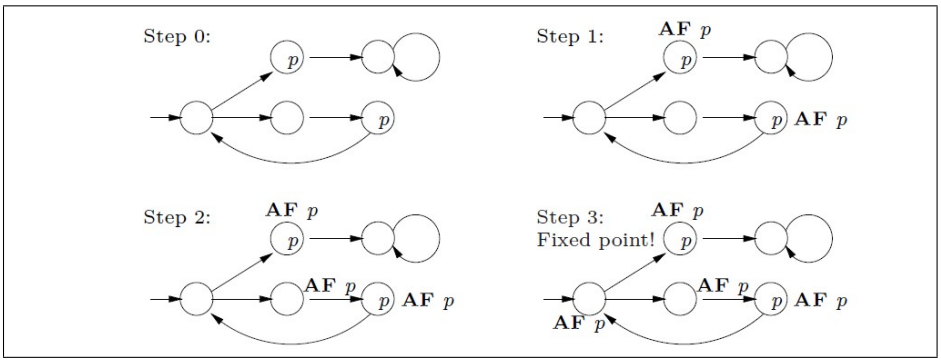


Figure 17.6: Sample execution of the CTL model checking algorithm

- **EX** ψ_1: Label state s with **EX** ψ_1 if at least one of its successor states is labeled with ψ_1,
- **AF** ψ_1:
  (a) If any state s is already labeled with ψ_1, then label it with **AF** ψ_1,
  (b) label any state with **AF** ψ_1 if all successor states are labeled with **AF** ψ_1,
  (c) if step 2 changed the labeling, then go back to 2. Otherwise, stop.
- $\mathbf{E}\big[\psi_1\ \mathbf{U}\ \psi_2\big]$:
  (a) If any state s is already labeled with ψ_2, then label it with $\mathbf{E}\big[\psi_1\ \mathbf{U}\ \psi_2\big]$,
  (b) label any state with $\mathbf{E}\big[\psi_1\ \mathbf{U}\ \psi_2\big]$ if it is labeled with ψ_1 and at least one of its successor states is already labeled with $\mathbf{E}\big[\psi_1\ \mathbf{U}\ \psi_2\big]$,
  (c) if step 2 changed the labeling, then go back to 2. Otherwise, stop.

3. **Postprocessing** If all initial states S_0 are labeled with φ', output "Yes". Otherwise, output "No".

Figure 17.6 contains a sample execution of the algorithm. Let M be the top left Kripke structure and let φ be the CTL formula **AF** p. The algorithm reaches a fixed point after three steps. Since all initial states are labeled with φ upon termination, the Kripke structure satisfies φ, that is, $M \models \varphi$.

Exercise 17.3.1. 1. Handsimulate the above algorithm on the Kripke structure in Figure 16.1 and the formulas

$$\mathbf{AF}\ r$$

and

$$\mathbf{E}\big[p \vee q\ \mathbf{U}\ \neg\, p \wedge \neg\, q \wedge r\big].$$

function SAT(φ) =
(* returns the set of states satisfying φ *)
begin case φ **of**
tt : **return** S
ff : **return** $\emptyset$
p : **return** $\{s \mid p \in L(s)\}$
$\neg\, \varphi_1$: **return** $S \backslash \mathrm{SAT}(\varphi_1)$
$\varphi_1 \wedge \varphi_2$: **return** $\mathrm{SAT}(\varphi_1) \cap \mathrm{SAT}(\varphi_2)$
$\varphi_1 \vee \varphi_2$: **return** $\mathrm{SAT}(\varphi_1) \cup \mathrm{SAT}(\varphi_2)$
$\varphi_1 \rightarrow \varphi_2$: **return** $\mathrm{SAT}(\neg\, \varphi_1 \vee \varphi_2)$
AX φ_1 : **return** $\mathrm{SAT}(\neg\, \mathbf{EX}\, \neg\, \varphi_1)$
EX φ_1 : **return** $\mathrm{SAT}_{\mathbf{EX}}(\varphi_1)$
AF φ_1 : **return** $\mathrm{SAT}_{\mathbf{AF}}(\varphi_1)$
EF φ_1 : **return** $\mathrm{SAT}(\mathbf{E}[tt\ \mathbf{U}\ \varphi_1])$
AG φ_1 : **return** $\mathrm{SAT}(\neg\, \mathbf{EF} \neg\, \varphi_1)$
EG φ_1 : **return** $\mathrm{SAT}(\neg\, \mathbf{AF} \neg\, \varphi_1)$
$\mathbf{A}[\varphi_1\ \mathbf{U}\ \varphi_2]$: **return** $\mathrm{SAT}(\neg\, (\mathbf{E}[\neg\, \varphi_2\ \mathbf{U}\ (\neg\, \varphi_1 \wedge \neg\, \varphi_2)] \vee \mathbf{EG} \neg\, \varphi_2))$
$\mathbf{E}[\varphi_1\ \mathbf{U}\ \varphi_2]$: **return** $\mathrm{SAT}_{\mathbf{EU}}(\varphi_1, \varphi_2)$
end
end

Figure 17.7: The function SAT. Given a CTL formula φ it returns the set of states satisfying φ. Uses helper functions in Figure 17.8.

2. Handsimulate the above algorithm on the Kripke structure in Figure 16.3 and the formulas

$$\mathbf{AG}\ \neg\, (pc_0 = cr_0 \wedge pc_1 = cr_1)$$

and

$$\mathbf{AG}\ (pc_0 = nc_0 \rightarrow (\mathbf{AF}\ pc_0 = cr_0)).$$

We now present the above algorithm in more concrete terms. We restrict our attention to the labeling step of the algorithm. Figure 17.7 contains the main function whereas Figure 17.8 contains helper functions to handle the cases **EX**, **AF**, and **EU**.

17.3.1 Counter examples

Counter examples are execution traces that demonstrate why a CTL specification fails. For instance, the execution trace

$$s_0 s_2 s_2 \ldots$$

```
function SAT_EX(φ) =
(* returns the set of states satisfying EX φ *)
new X, Y;
begin
  X :=SAT(φ);
  Y :={s_0 ∈ S | R(s_0, s_1) for some s_1 ∈ X}
  return Y
end

function SAT_AF(φ) =
(* returns the set of states satisfying AF φ *)
new X, Y;
begin
  X :=∅ ; Y :=SAT(φ);
  while X ≠ Y do
    X := Y;
    Y := Y ∪ {s | for all s' with R(s, s') we have s' ∈ Y}
  end;
  return Y
end

function SAT_EU(φ, ψ) =
(* returns the set of states satisfying E[φ U ψ] *)
new W, X, Y;
begin
  W :=SAT(φ) ; X := S ; Y :=SAT(ψ);
  while X ≠ Y do
    X := Y;
    Y := Y ∪ (W ∩ {s | exists s' with R(s, s') and s' ∈ Y})
  end;
  return Y
end
```

Figure 17.8: Helper functions for function SAT in Figure 17.7.

of the structure in Figure 16.1 is a counter example for **AG** p, because p does not hold in s_2. Also, the executions given in in Sections 17.2.2 and 17.2.3 to show why properties *evtlEntry* and *noDeadlock* fail on our second and third versions respectively are counter examples. The above algorithm can be extended to output these counter examples.

However, some CTL formulas do not have a single trace (i.e., a linear sequence of states) as counter examples, but a whole, possibly infinite collection of traces. Consider, for example the CTL formula **EF** $(p \wedge q \wedge r)$ asserting the reachability of state in which propositions p, q, and r hold. The structure in Figure 16.1 does not satisfy this formula, but there is no single trace that could serve as counter example illustrating this violation. Rather, the *entire* collection of all possible traces, i.e., the computation tree of the structure is the counter example, because only it shows that no state in which p, q, and r hold is reachable. In general, it is the CTL formulas starting with an existential path quantifier for which a single trace cannot serve as counter example. In these cases, most model checking tools (such as NuSMV) do not output any counter example, because the output of the entire computation tree is usally not practical.

17.3.2 Complexity

Let us analyze the complexity of this algorithm. The clauses for **AF** and **EU** are the most expensive. In particular, they both contain a loop which, in every iteration, applies a labeling step to every vertex in the graph and which terminates only when these labeling steps stop incurring any changes. Using a standard breadth-first traversal algorithm, every node in a graph can be visited in $O(\mid S \mid + \mid R \mid)$ giving the worst-case complexity of a single execution of step 2 where $\mid S \mid$ and $\mid R \mid$ denote the size of the state space and the transition relation respectively. In the worst case, step 2 is executed $\mid S \mid$ times. Thus, the complexity of each individual clause of the algorithm is $O(\mid S \mid \cdot(\mid S \mid + \mid R \mid))$. The number of subformulas of a formula is linear in the number n of connectives in that formula. Thus, the complexity of the entire algorithm is $O(n \cdot \mid S \mid \cdot(\mid S \mid + \mid R \mid))$. Since we want to be able to verify large systems with lots of states, the above result is not encouraging.

17.3.3 Optimizations

Explicit treatment of AX, EF, AG, EG and AU

The labeling algorithm treats the connectives **AX**, **EF**, **AG**, **EG** and **AU** in terms of the adequate connectives **EX**, **AF**, and **EU**. This means that the labeling algorithm only has to handle three different cases and allows a very concise presentation. However, it also slows the algorithm down by a constant factor. To handle **AX**φ, for instance, we have to compute all states satisfying φ, $\neg\ \varphi$, **EX**$\neg\ \varphi$, and then finally, $\neg$ **EX**$\neg\ \varphi$. An explicit treatment of **EX** computes the states satisfying **AX**φ directly from those satisfying φ:

- **AX** ψ_1: Label state s with **AX** ψ_1 if all of its successor states are labeled with ψ_1,

Similarly for **EF** and **AU**:

- **EF** ψ_1:
  1. If any state s is already labeled with ψ_1, then label it with **EF** ψ_1,
  2. label any state with **EF** ψ_1 if some successor state is labeled with **EF** ψ_1,
  3. if step 2 changed the labeling, then go back to 2. Otherwise, stop.
- **A**$\big[\psi_1$ **U** $\psi_2\big]$:
  1. If any state s is already labeled with ψ_2, then label it with **A**$\big[\psi_1$ **U** $\psi_2\big]$,
  2. label any state with **A**$\big[\psi_1$ **U** $\psi_2\big]$ if it is labeled with ψ_1 and all of its successor states are already labeled with **A**$\big[\psi_1$ **U** $\psi_2\big]$,
  3. if step 2 changed the labeling, then go back to 2. Otherwise, stop.

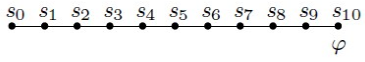
The explicit treatment of **AG** is slightly different:

- **AG** ψ_1:
  1. Label all states with **AG** ψ_1,
  2. remove label **AG** ψ_1 from any state s, if s is not labeled ψ_1,
  3. remove label **AG** ψ_1 from any state s, if s at least one successor not labeled with **AG** ψ_1,
  4. if step 3 changed the labeling, then go back to 3. Otherwise, stop.

The explicit treatment of **EF** is similar. Adding these cases to the algorithm speeds it up by a constant factor.

Using backwards search for AF and EU

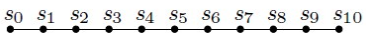
It turns out that performing the iterations in clauses **AF** and **EU** in a *particular order* can improve the complexity of the above algorithm considerably. More precisely, visiting the vertices in *backwards* breadth-first order avoids any vertex having to be visited more than once, and improves the complexity of the algorithm by a factor of $\mid S \mid$ to $O(n \cdot (\mid S \mid + \mid R \mid))$. To illustrate this improvement, consider the Kripke structure below.



Suppose we want to determine the set of states in which **AF** φ holds. When using forward breadth-first search (starting with s_0), ten iterations are needed before the system stabilizes. However, when using backward breadth-first search (starting with s_{10}), only one iteration is needed. The main idea is that the iterations in clause **AF** φ start in states satisfying φ. Similarly, iterations in clause $\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]$ start in states satisfying ψ_2. Using backward search also speeds up the explicit treatments of **EF**, **AG**, and **AU** suggested above.

Using strongly connected components for EG

Our treatment of **EG** formulas is another, albeit much more benign, source of inefficiency. Consider, for instance, the following Kripke structure



and the formula **EG** p. None of the states in the structure satisfy p. Therefore, no state satisfies **EG** p. However, even if **EG** is treated explicitly, we are forced to perform several iterations over the state space. In most cases, the following treatment of **EG** using strongly connected components (SCCs) is more efficient.

- **EG** ψ_1:
    1. Restrict the graph to states satisfying ψ_1, that is, all states not satisfying ψ_1 and their transitions are deleted.
    2. Find the maximal SCCs in the restricted graph. A SCC is a region of the state space in which any two states are connected through a finite path.
    3. Use backwards breadth-first searching on the restricted graph to any state that can reach an SCC.

Since SCCs can be computed in $O(\mid S \mid + \mid R \mid)$, the worst-case complexity remains unchanged.

17.4 Model checking with fairness constraints

The verification of $(M, s) \models \varphi$ may fail because the model M may contain behaviour which is unrealistic, or guaranteed not to occur in the actual system being modeled. One way of dealing with this problem is to refine the model such that it does not contain these unrealistic paths. However, in most cases, it is far easier to stick to the original model and to impose a condition or a filter on the model checking process. Thus, instead of checking $(M, s) \models \varphi$, we verify $(M, s) \models \psi \rightarrow \varphi$, where formula ψ rules out the undesired or unrealistic behaviour. Unfortunately, this technique does not always work. For instance, often we are only interested in computation paths in which a resource is used

fairly. For instance, if we are verifying the asynchronous model of the concurrent system

$$\textbf{cobegin } C_0 \| C_1 \textbf{ end}$$

we may only want to consider only those computations in which each of the processes is always eventually scheduled to execute, that is, the underlying scheduler never ignores a process ready to execute forever. Similarly, if we are verifying an asynchronous circuit with an arbiter, we may wish to consider only those executions in which the arbiter does not ignore one of its inputs forever. Alternatively, we may want to consider communication protocols that operate over reliable channels which have the property that no message is ever continuously transmitted but never received. It turns out that a filter of the kind described above cannot be used to rule out unfair paths. Instead, we introduce *fairness constraints*, that is, CTL formulas that must hold infinitely often along every computation path. We call such paths *fair computation paths*. In the presence of the fairness constraint φ the connectives **A** and **E** range over fair paths only. We will return to the issue of fairness when discussing the model checker NuSMV in the next section. Section 18.4 will then discuss how the the model checking algorithm can be adapted to handle fairness constraints.

Chapter 18

The NuSMV system (Weeks 9-11))

NuSMV is the best known CTL model checker. It is a reimplementation of the original version called SMV (Symbolic Model Verifier). Its input language was especially designed to facilitate the description of finite state machines. In other words, NuSMV does not explicitly target circuits or programs. While NuSMV takes care of the translation of NuSMV input code into Kripke structures, the user has to translate programs and circuits into NuSMV (see Figure 18.1).

NuSMV automatically checks the validity of CTL formulas on the input machine and outputs counter examples if they exist. We will now sketch how the modeling, specification, and verification steps are supported in NuSMV.

We will first describe the language that NuSMV uses to describe Kripke structures through a few of examples. For full details of the language, please see [NuS17].

18.1 NuSMV language: Specifying synchronous systems

Every NuSMV program contains a module called `main`. Inside every module local variables can be declared. To enforce finite models, variables are allowed to range over booleans (denoted by 0 and 1), enumerations, and integer subranges only. Moreover, arrays of the above types are also supported. Variable declarations are preceded by the keyword `VAR`. The declaration section is typically followed by an assignment section introduced by keyword `ASSIGN` in which the initial states and the transition relation are defined. The initial value of a variable is described by means of the `init` construct. Transition relations are expressed using the `next` construct, which describes the next values a variable can take on given its current value. Consider, for instance, the following NuSMV program P1.1 describing the Kripke structure in Figure 16.1 on page 176.

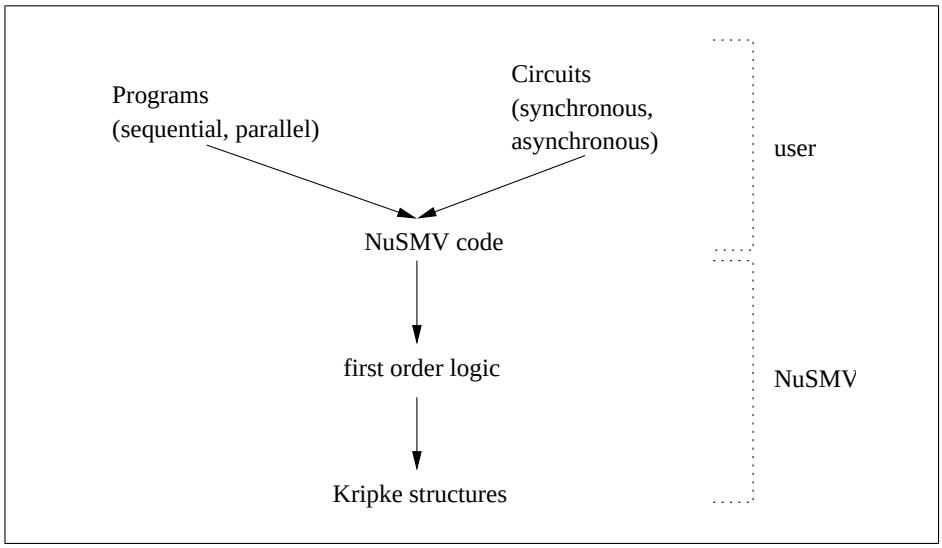


Figure 18.1: NuSMV code as intermediate language

```
-- NuSMV program P1.1: no p, q, r
MODULE main
VAR
        state : {s0, s1, s2};
ASSIGN
        init(state) := s0;

        next(state) := case
                         state=s0 : {s1, s2};
                         state=s1 : {s0, s2};
                         state=s2 : state;
                       esac;
```

The double dash `--` starts a comment. Every NuSMV program contains a module called `main`. The `VAR` section declares a variable `state`. The `ASSIGN` section defines how these variables are initialized (using `init`) and how their values can change (using `next`). In the example, `state` is initialized to `s0`. The way `state` can change is defined using a `case` expression, which is evaluated from top to bottom as follows:

1. Find the first line l whose guard evaluates to `TRUE`.
2. In l, if the expression to the right of the colon evaluates to a single value that value is returned. If the expression to the right of the colon evaluates to a set, an element from that set is returned non-deterministically.
3. If none of the guards evaluate to `TRUE`, NuSMV will complain that the case conditions are not exhaustive. Note that in the above example, this case never occurs.
4. To easily make the case conditions exhaustive, `TRUE` can be used as guard,

typically in the last line. So, e.g., instead of `state=s2 : state` we could have written `TRUE : state`.

Program P1.1 does reflect the atomic propositions p, q, and r that occur in the Kripke structure in Figure 16.1. Program P1.2 below shows how they can be incorporated.

```
-- NuSMV program P1.2: with p, q, r
MODULE main
VAR
        state : {s0, s1, s2};
        p : boolean;
        q : boolean;
        r : boolean;
ASSIGN
        init(state) := s0;
        init(p) := TRUE;
        init(q) := TRUE;
        init(r) := FALSE;

        next(state) := case
                         state=s0 : {s1, s2};
                         state=s1 : {s0, s2};
                         state=s2 : state;
                       esac;
        next(p) := case
                     next(state)=s0 : TRUE;
                     TRUE : FALSE;
                   esac;
        next(q) := case
                     next(state)=s0 : TRUE;
                     next(state)=s1 : TRUE;
                     TRUE : FALSE;
                   esac;
        next(r) := case
                     next(state)=s0 : FALSE;
                     next(state)=s1 : TRUE;
                     next(state)=s2 : TRUE;
                   esac;
```

Given a top-level `next` statement to define how some variable v_1 changes, we see that another `next` statement can be used in a guard of the top-level `next` to make the change of v_1 dependent on the concurrent change to some other variable v_2. For instance, whether or not p should hold in the next state, depends on whether or not we are transitioning into state s_0 or not.

For another example, let's see how the modulo 8 counter from Figure 16.2 is represented in NuSMV.

```
-- NuSMV program P2.1
MODULE main
VAR
        v0 : boolean;
        v1 : boolean;
        v2 : boolean;
ASSIGN
        init(v0) := FALSE;
        init(v1) := FALSE;
        init(v2) := FALSE;

        next(v0) := !v0;
        next(v1) := (v0 | v1) & !(v0 & v1);
        next(v2) := ((v0 & v1) | v2) & !((v0 & v1) & v2);
```

In NuSMV, state changes *inside* a module are carried out in lock step, that is, synchronously. Thus, the three variables `v0`, `v1`, and `v2` are always changed at the same time. Running NuSMV on this example reveals that the counter has, surprise, surprise, eight states. Note that in constrast to the previous example, the transition relation of the counter is completely deterministic. The next value of a variable is always unique. Every state along every path has exactly one successor. Also, we see that the `case` expression is not always necessary to define the value of a variable in the next state.

The following is another version of an modulo 8 counter.

```
-- NuSMV program P2.2
MODULE main
VAR
        v0 : CounterCell(1);
        v1 : CounterCell(v0.carryOut);
        v2 : CounterCell(v1.carryOut);
SPEC
        AG AF v2.carryOut=1

MODULE CounterCell(carryIn)
VAR
        value : 0..1;
ASSIGN
        init(value) := 0;
        next(value) := case
 value=0 | value=1 : (value + carryIn) mod 2;
 TRUE : value; -- to satisfy exhaustiveness checker
     esac;
DEFINE
        carryOut := case               -- simulates logical 'AND'
    value=1 & carryIn=1 : 1;
    TRUE : 0;
  esac;
```

This example introduces two new features.

- The main module contains a second module called `CounterCell` taking one parameter `carryIn`. The module is instantiated three times to create three variables `v0`, `v1` and `v2` representing the three bits. In contrast to the previous example, this representation brings out the fact that each bit carries out the same computation. Note that, by default, modules are composed synchronously. All bits receive a new value at the same time.
- To make a description more concise or readable, a symbol can be associated with an expression after the `DEFINE` keyword. Defined symbols are usually preferable to variables, since they don't increase the state space. However, they cannot be assigned non-deterministically.
- Finally, we have included a specification. Specifications are preceded by the keyword `SPEC`. In NuSMV, the connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$ are represented by `&`, `|`, `->`, and `!` respectively.

18.2 Using NuSMV for analysis

We do not have any specifications yet that we could use to determine the correctness of the examples above using model checking. However, we can use

NuSMV to run some initial analyses.

1. Starting NuSMV We first start the interactive mode using `NuSMV -int` from a DOS, Linux, or Mac terminal prompt:

```
> NuSMV -int
*** This is NuSMV 2.5.4 (compiled on Fri Oct 28 14:13:29 UTC 2011)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
...
NuSMV >
```

We will assume that all NuSMV commands described below are given to the NuSMV command prompt `NuSMV >`.

2. Getting help `help` lists all available commands. Given a command `cmd`, `cmd -help` lists all options and parameters for this command.

3. Building a model We first need to read in an NuSMV program: `read_model -i "program1_1.smv"` assuming that Program P1.1 is stored in file `program1_1.smv`. Then, the Kripke structure must be build. The command `go` is the most convenient way to achieve this.

4. Basic analyses Now, analysis commands can be issued. All of these analyses will be performed on the Kripke structure that resulted from the most recent invocation of `go`. Basic analyses include `print_reachable_states` and `check_fsm`:

```
NuSMV > print_reachable_states
############################################################################
system diameter: 2
reachable states: 3 (2^1.58496) out of 3 (2^1.58496)
############################################################################
NuSMV > check_fsm
--------------------------------------------------------
The transition relation is total
--------------------------------------------------------
```

5. Simulation We can also simulate the model, i.e., execute it and collect traces. To this end, we first need to pick an intial state. We can do this in different ways, including randomly using `pick_state -r`, or interactively using `pick_state -i`, which will display all possible initial states and prompt the user to pick one.

Then, simulation can start

```
NuSMV > simulate -k 5 -r -v
********  Simulation Starting From State 1.1   ********
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 1.1 <-
  state = s0
-> State: 1.2 <-
  state = s1
-> State: 1.3 <-
```

```
  state = s2
-> State: 1.4 <-
  state = s2
-> State: 1.5 <-
  state = s2
-> State: 1.6 <-
  state = s2
```

where

- option `-k 5` indicates that the structure should be simulated for 5 steps,
- option `-r` indicates that whenever execution has reached a state with more than one successor that that successor is chosen randomly. An alternative is option `-i`, which allows the user to pick the successor.
- option `-v` prints the values of all variables. An alternative is `-p`, which prints only the values of variables that have changed from one state to another.

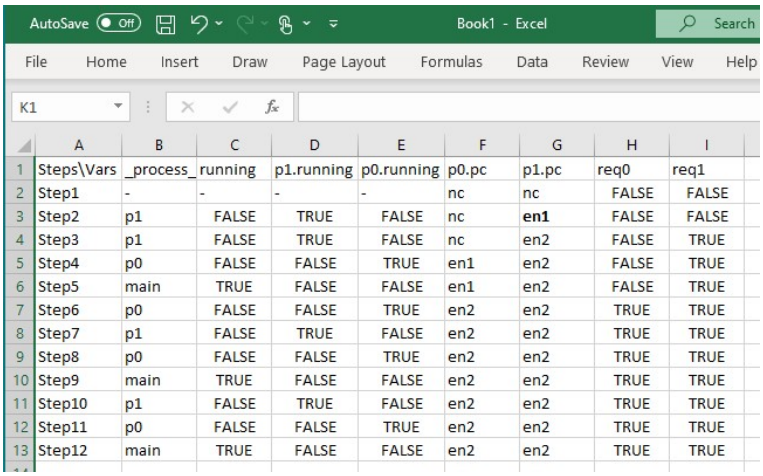
Other options are available (use `simulate -help`).

6. Display of traces The most recently generated trace can be displayed and stored in some file using `show_traces`.

The format in which a trace is shown can be changed using different 'trace plugins'.

```
NuSMV > show_plugins
[D]  0   BASIC TRACE EXPLAINER - shows changes only
     1   BASIC TRACE EXPLAINER - shows all variables
     2   TRACE TABLE PLUGIN - symbols on column
     3   TRACE TABLE PLUGIN - symbols on row
     4   TRACE XML DUMP PLUGIN
     5   TRACE COMPACT PLUGIN - Shows the trace in tabular fashion
NuSMV > show_traces -p 5
<!-- ################## Trace number: 1 ################## -->
Steps/Vars      state
Step1           s0
Step2           s1
Step3           s2
Step4           s2
Step5           s2
Step6           s2
```

The 'trace compact plugin' produces output that can be cut and pasted into a spreadsheet tool such as Excel which then allows for easier search and manipulation. E.g., the screen shot below shows a trace of a faulty version of our mutual exclusion protocol ('Third attempt') demonstrating why the version does not satisfy eventual entry for process p1.



	A	B	C	D	E	F	G	H	I
1	Steps\Vars	_process_	running	p1.running	p0.running	p0.pc	p1.pc	req0	req1
2	Step1	-	-	-	-	nc	nc	FALSE	FALSE
3	Step2	p1	FALSE	TRUE	FALSE	nc	**en1**	FALSE	FALSE
4	Step3	p1	FALSE	TRUE	FALSE	nc	en2	FALSE	TRUE
5	Step4	p0	FALSE	FALSE	TRUE	en1	en2	FALSE	TRUE
6	Step5	main	TRUE	FALSE	FALSE	en1	en2	FALSE	TRUE
7	Step6	p0	FALSE	FALSE	TRUE	en2	en2	TRUE	TRUE
8	Step7	p1	FALSE	TRUE	FALSE	en2	en2	TRUE	TRUE
9	Step8	p0	FALSE	FALSE	TRUE	en2	en2	TRUE	TRUE
10	Step9	main	TRUE	FALSE	FALSE	en2	en2	TRUE	TRUE
11	Step10	p1	FALSE	TRUE	FALSE	en2	en2	TRUE	TRUE
12	Step11	p0	FALSE	FALSE	TRUE	en2	en2	TRUE	TRUE
13	Step12	main	TRUE	FALSE	FALSE	en2	en2	TRUE	TRUE

7. Changing the program Changes to the NuSMV program are not automatically reflected in the model used for analysis. After each change, the `reset` command must be issued, the modified program must be read in using `read_model` and the model must be rebuilt using `go`.

```
NuSMV > reset
NuSMV > read_model -i "program1_1.smv"
NuSMV > go
```

8. Checking CTL specifications CTL formulas can be added to the NuSMV program. Suppose the file "program1_3.smv" contains program P1.3 that is just like program P1.2 except that the lines `SPEC AG (q -> EX q)`, `SPEC EF AG r`, and `SPEC AG (p | q)` have been added at the end

```
-- Program 1.3: with CTL formulas
MODULE main
VAR
        state : {s0, s1, s2};
        p : boolean;
        q : boolean;
        r : boolean;
ASSIGN
        ...
SPEC AG (q -> EX q)
SPEC EF AG r
SPEC AG (p | q)
```

After reading in the program and building the model, the `check_ctlspec` command is used to check the included formulas.

```
NuSMV > reset
NuSMV > read_model -i "program1_3.smv"
NuSMV > go
NuSMV > check_ctlspec
-- specification AG (q -> EX q)  is true
-- specification EF (AG r)  is true
-- specification AG (p | q)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
```

```
Trace Type: Counterexample
-> State: 1.1 <-
  state = s0
  p = TRUE
  q = TRUE
  r = FALSE
-> State: 1.2 <-
  state = s2
  p = FALSE
  q = FALSE
  r = TRUE
```

However, CTL formulas can also be supplied as arguments `check_ctlspec -p "AF r"`. Counterexamples are stored as traces that can be displayed and stored using `show_traces` as described above. As mentioned in Section 16.2.4, some CTL formulas do not have a single trace as counter example. NuSMV does not output a counter example in this case, and, unfortunately, its output is somewhat misleading in this case:

```
NuSMV > check_ctlspec -p "EF p&q&r"
-- specification EF ((p & q) & r)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = s0
  p = TRUE
  q = TRUE
  r = FALSE
```

18.3 NuSMV language: Specifying asynchronous systems

Based on these examples, we can see that describing Kripke structures or a synchronous circuits in NuSMV is very natural and straight forward. To express concurrent programs as discussed in Sections 16.2.4 and 17.2 in NuSMV, we have to do a little more work. NuSMV program in Figure 18.2 models the program in the example on page 188. The main module contains a second module called `MutexProcess` taking two parameters `id` and `t`. This module is instantiated twice to create two variables `C0` and `C1` representing the two processes in the example. Each process is instantiated with its own id and the variable `turn` which allows it to be shared. Each process also has its own program counter. The reason is that the NuSMV language does not have the control flow constructs that are standard in imperative programming languages such as **if**, **while**, etc. Instead, we need to encode the control flow explicitly using a program counter in very much the same way as we have done in Sections 16.2.3 and 16.2.4. Turns out, the NuSMV language is more like *data flow* language than an imperative programming language, with an emphasis on how data is being manipulated than on control flow.

The keyword `process` in front of each instantiation causes both processes to be executed asynchronously, rather than synchronously as in the previous example. That is, it is not the case that every transition by process `C0` is accompa-

```
MODULE main
VAR
        turn : {0,1};
        C0 : process MutexProcess(0, turn);
        C1 : process MutexProcess(1, turn);
SPEC
        AG !(C0.pc=cr & C1.pc=cr)            -- mutual exclusion
SPEC
        AG (C0.pc=nc -> AF (C0.pc=cr))       -- eventual entry

MODULE MutexProcess(id, t)
VAR
        pc : {l, nc, cr};
DEFINE
        otherId := 1-id;
ASSIGN
        init(pc) := nc;

        next(pc) := case
                      pc=l : nc;
                      pc=nc & (t=id) : cr;
                      pc=nc & !(t=id) : pc;
                      pc=cr : l;
                    esac;

        next(t) := case
                     pc=cr : otherId;
                     TRUE : t;
                   esac;
```

Figure 18.2: NuSMV program modeling the concurrent program on page 188.

nied by a transition by process `C1`. Rather, the transitions of the two processes occur in an arbitrarily interleaved, asynchronous fashion, which, of course, is exactly what we want. Asynchronous composition is useful for describing systems without a global clock like communication protocols or asynchronous circuits.

Running NuSMV on this example yields the output shown in Figure 18.3.

We see that while mutual exclusion is satisfied, eventual entry is not. A counter example is output, that is, a sequence of states satisfying the negation of the eventual entry property,

$$\mathbf{EF}\ (C_0.pc = nc\ \wedge\ \mathbf{EG}\ \neg\ C_0.pc = cr).$$

This counter example starts in State 1.1 with all variables set to an initial value. Then, in State 1.2, process `C0` is executed. By default, NuSMV only shows the variables that change in a transition. Thus, in State 1.3 all variables are unchanged except that the program counter process `C0` now is `nc`. Intuitively, `C0` has moved into its non-critical section. State 1.3 also constitutes the beginning of a loop (remember that every path in a Kripke structure must be infinite). To get to State 1.4, `C0` is executed again with no state change. Since `turn` is 1, `C0` must wait. This marks the end of the loop, that is, `C0` is executed again with no change (`pc` already is `nc`) and so on. This sequence does indeed violate eventual entry. `C0` is blocked forever along the given path.

```
> NuSMV -r /home/jd/classes/cisc835/modelCheck/mutex.smv
-- specification AG (!(C0.pc = cr & C1.pc = cr)) is true
-- specification AG (C0.pc = nc -> AF C0.pc = cr) is false
-- as demonstrated by the following execution sequence
State 1.1:
turn = 1
C0.otherId = 1
C0.pc = 1
C1.otherId = 0
C1.pc = 1

State 1.2:
[executing process C0]

-- loop starts here --
State 1.3:
C0.pc = nc

State 1.4:

############################################################
Runtime Statistics
------------------
reachable states: 12 (2^3.58496) out of 18 (2^4.16993)
SMV finished at Sat Sep 23 16:22:19
```

Figure 18.3: Command line ouput produced by NuSMV when run on the program in Figure 18.2

18.3.1 Fairness constraints

What is surprising about this counter example is that process `C1` is never executed, and that the counter example is based on an *unfair* execution as already discussed in Section 16.2.4. As every runtime scheduler reasonably can be assumed to not ignore processes forever from some point on, unfair executions can be considered unrealistic and thus uninteresting. To remove them from the analysis (and, possibly, showing up as spurious counter examples), we need to prevent NuSMV explicitly from ignoring a process forever. This is accomplished with a fairness constraint. Every NuSMV process has a special boolean variable `running` associated with it which is true in a state if and only if the process has just been executed. Thus, to rule out paths in which a process is executed finitely many times (in the example above it was zero times) we require that for each process `running` is true infinitely often. In NuSMV this is accomplished by adding the line

```
FAIRNESS running
```

at the end. After this modification, both specifications are satisfied.

As another example of an NuSMV program, Figure 18.4 contains the NuSMV program modelling the second version of the tie-breaker algorithm presented in Section 17.2.4.

Exercise 18.3.1. 1. Using the NuSMV code above, check if the Kripke structure in Figure 16.1 satisfies the formulas **AF AG** r, **EF AG** r and $\mathbf{A}\big[(p \wedge q) \vee (q \wedge r)\ \mathbf{U}\ r\big]$. If a formula is not satisfied, find a fairness

```
1  -- Asynchronous, fair model allowing non-termininating non-critical section
2  -- Tie-breaker algorithm, version 2: req set first and then last
3  -- Satisfies mutex and eventual entry
4  MODULE P(id, myReq, otherReq, last)
5  VAR
6    st : {nc, en1, en2, en3, cr, ex};
7  ASSIGN
8    init(st) := nc;
9    next(st) :=
10     case
11       st=nc : {nc, en1};
12       st=en1 : en2;
13       st=en2 : en3;
14       st=en3 & myReq & (!otherReq | !last=id) : cr;
15       st=cr : ex;
16       st=ex : nc;
17       1 : st;
18     esac;
19   next(myReq) :=
20     case
21       st=en1 : 1;
22       st=ex : 0;
23       1 : myReq;
24     esac;
25   next(last) :=
26     case
27       st=en2 : id;
28       1 : last;
29     esac;
30 FAIRNESS
31   running
32
33 MODULE main
34 VAR
35   p0 : process P(0, req0, req1, last);
36   p1 : process P(1, req1, req0, last);
37   req0 : boolean;
38   req1 : boolean;
39   last : boolean;
40 ASSIGN
41   init(req0) := 0;
42   init(req1) := 0;
43 SPEC
44   AG !(p0.st=cr & p1.st=cr)
45 -- true
46 SPEC
47   AG (p0.st=en1 -> AF p0.st=cr)
48 -- true
49 SPEC
50   AG (p1.st=en1 -> AF p1.st=cr)
51 -- true
```

Figure 18.4: NuSMV program modelling the second version of the tie-breaker algorithm presented in Section 17.2.4.

constraint such that the model satisfies the formula after addition of that constraint.

2. Consider the NuSMV code in the mutual exclusion example used above.

   (a) Change the code so that each process can stay in its non-critical region forever. Are the specifications still satisfied? If not, which one(s) fail(s) and why?

   (b) Change the code so that each process can stay in its critical region arbitrarily long (but not forever) without violating the specifications. Hint: You may want to introduce new fairness constraints.

18.4 Model checking with fairness

We need to adapt the model checking algorithm such that the path quantifiers **A** and **E** range over fair paths only. Let FC be a set of fairness constraints $\{\psi_1, \ldots, \psi_n\}$. A path is fair with respect to FC if each φ_i holds infinitely often along that path. We will write $\mathbf{A}_{FC}$ and $\mathbf{E}_{FC}$ for the operators restricted to fair paths. Using the same argument as before, we can show that $\mathbf{E}_{FC}\mathbf{X}$, $\mathbf{E}_{FC}\mathbf{G}$, and $\mathbf{E}_{FC}\mathbf{U}$ form an adequate set. Moreover, we have

$$\begin{aligned} \mathbf{E}_{FC}[\varphi\ \mathbf{U}\ \psi] &= \mathbf{E}[\varphi\ \mathbf{U}\ (\psi \wedge \mathbf{E}_{FC}\mathbf{G}\ tt)] \\ \mathbf{E}_{FC}\mathbf{X}\ \varphi &= \mathbf{E}\mathbf{X}\ (\varphi \wedge \mathbf{E}_{FC}\mathbf{G}\ tt) \end{aligned}$$

Due to these equations we only need to provide an algorithm for $\mathbf{E}_{FC}\mathbf{G}\ \varphi$. The first two steps are as in Section 17.3.

- $\mathbf{E}_{FC}\mathbf{G}\ \varphi$:

  1. Restrict the graph to states satisfying φ.
  2. Find the maximal SCCs of the restricted graph.
  3. Remove an SCC if, for some φ_i, it does not contain a state satisfying φ_i. The resulting SCCs are the fair SCCs. Any state of the restricted graph that can reach a fair SCC has a fair path from it.
  4. Use backwards breadth-first searching to find the states in the restricted graph that can reach a fair SCC.

The complexity of the algorithm still is $O(n\cdot \mid S \mid \cdot(S+R))$, or $O(n \cdot (\mid S \mid + \mid R \mid))$ if the optimizations presented in Section 17.3.3 are adopted.

Chapter 19

The state explosion problem (Week 11)

Model checking requires us to express the system to be analyzed as a mathematical structure with finitely many states, but with possibly infinitely many paths. The analysis itself explores this state machine through an exhaustive search using smart algorithms and data structures. The analysis is always guaranteed to terminate, because even if the structure has infinitely many paths, these paths can use only finitely many states and thus every infinite path must contain a loop, i.e., a state that occurs more than once. The algorithms leverage the fact that (the states in) this only need to be explored once.

As we seen in Section 17.3.2, the aymptotic worst-case complexity is $O(n \cdot (\mid S \mid + \mid R \mid))$, i.e., it is linear in the length of the formula n and the size of the state space (i.e., the number of states and transitions).

In contrast to predicate logic theorem proving model checking is fully automatic (Section 5), and in contrast to bounded satisfiability checking (as in Alloy) there are no 'false positives' (Section 11.10). Model checking appears to be superior, or is there a catch?

Yes, there is. The problem is that the size of the state space is guaranteed to be finite, but often becomes so large that it overwhelms all available computing resources and an exhaustive search becomes impossible. In these cases, the analysis needs to be terminated and the construction of a model that is small enough to allow exhaustive exploration, but also detailed enough to make the analysis and its results meaningful (see Section 16.1). The construction of such a model for realistic systems is challenging and a topic of ongoing research.

Why is the state space of realistic systems so large? Given an NuSMV program, there are several factors that determine the size of its Kripke structure:

- the number of variables,
- the number of different values that these variables can take on, and
- the number of concurrent processes.

Of these, the last is, by far, the most serious.

To see this, consider the program template P4.n below which creates n processes by instantiating module `P` n times. Each process `pi` successively assigns all integers between 1 and 1,000 to a local variable `x`.

```
-- Program P4.n: n processes counting up
MODULE main
VAR
        p1 : process P(1000);
        ...
        pn : process P(1000);

MODULE P(TO)
VAR
        x : 1..TO;
ASSIGN
        init(x):=1;
        next(x):= case
              x<TO: x+1;
              TRUE: x;
            esac;
```

As expected, the Kripke structure of P4.1 (i.e, n=1) has 1,000 states which takes NuSMV only a few seconds to compute (on a standard laptop with an Intel i5 CPU and 16GB of RAM).

```
NuSMV > reset
NuSMV > read_model -i "program4_1.smv"
NuSMV > go
NuSMV > time
...
NuSMV > print_reachable_states
############################################################################
system diameter: 1000
reachable states: 1000 (2^9.96578) out of 1000 (2^9.96578)
############################################################################
NuSMV > time
elapse: 5.40 seconds, ...
```

For different values of n, the table below lists the number of states of P4.n and the approximate time taken by NuSMV to compute this. We can see that the number of states and the time required to compute it grows exponentially with the number of processes.

Program	P4.1	P4.2	P4.3	P4.4	P4.5	P4.6
Number of reachable states	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
Time taken to compute (in secs)	5	8	22	63	281	13,115

This phenomenon is called the *state explosion problem*. While model checking has proven useful for the design and analysis of concurrent and distributed systems, its use can be challenging and require an understanding of the system deep enough to allow the construction of abstract, yet still useful models.

Chapter 20

Conclusion and discussion

The success of temporal logic model checking is due to the following advantages:

- Finite state machines are a natural and expressive modelling formalism.
- Temporal logics like CTL and LTL are expressive enough to express interesting properties.
- Temporal logics like CTL and LTL are not too expressive, that is, the problem of deciding whether a given finite state machine satisfies a given formula is decidable. Given clever data structures like BDDs and algorithms, the problem often is efficiently solvable.
- Once we have a finite state machine model of the system under investigation, the analysis is fully automatic. Full automation is essential for successfully hiding the complexities of the analysis from the user.
- A negative analysis result is, if possible, accompanied with a counter example that demonstrates the failure of the specification and can be used to improve the design.

However, a user of temporal logic model checking may also be faced with the following problems and challenges:

- The system under investigation has to be expressed as a finite state machine on an appropriate level of abstraction. This modelling step usually has to be performed by hand which makes it very error-prone. If the model does not correspond to the original system, the analysis results may be spurious and thus worthless.
- Even if the system can be expressed as a finite state machine, the size of the state space may be too large to be handled efficiently. In general, the size of state space grows exponentially with the number of processes (state explostion problem).

- Properties have to be expressed in temporal logic. Not all properties can be expressed as naturally and concisely as, for instance, deadlock freedom. In other words, expressing a property in temporal logic may require a lot of expertise. The use of "specification patterns" and a good user interface may mitigate this problem.

Chapter 21

Bibliographic notes

Most of the material in this document is taken from or inspired by [CGP99, HR00]. The translation from programs into formulas is given in [MP95]. CTL was invented by E.M. Clarke and A. Emerson [CE81]. Algorithms for model checking were developed independently by Clarke and Emerson [CE81] and Quielle and Sifakis [QS81].

Appendix A

List of included readings

Three papers are included:

1. A short reflection by the vice president and Chief Internet Evangelist at Google on how it was beneficial for the evolution and success of the internet that the standards and protocols the internet is based on are specifications and thus declarative, rather than operational.

   > Vinton G. Cerf. In Praise of Under-Specification? Communications of the ACM 60(8):7-7. August 2017.

2. Article by the author of Alloy briefly describing the language and tool, together with some applications ranging from critical systems, to network protocols, web security, and code verification.

   > Daniel Jackson. Alloy: A Language and Tool for Exploring Software Designs. Communications of the ACM 62(9):66-76. September 2019.

3. Engineers at Amazon Web Services (AWS) have used formal specification and model checking to help solve difficult design problems in critical systems since 2011. This article describes the motivation and experience, and what has worked well, and what has not.

   > Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. Communications of the ACM 62(9):66-76. September 2019.

Bibliography

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[ACM15] Software engineering 2014, curriculum guidelines for undergraduate degree programs in software engineering, 2015.

[And11] M. Andreesen. Why software is eating the world. *The Wall Street Journal*, August 2011.

[Arb16] S. Arbesman. *Overcomplicated: Technology at the Limits of Comprehension*. Penguin Random House, 2016.

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[AS19] B.K. Aichernig and R. Schumi. Property-based testing of web services by deriving properties from business-rule models. *Software and Systems Modeling*, 2019.

[Ass22] AssertJ: fluent assertions Java library, 2022. Available at `https://assertj.github.io/doc`.

[Bah99] A. Bahrami. *Object-Oriented Systems Development*. McGraw-Hill, 1999.

[Bal14] Caroline Baldwin. Airbus interview: research, innovation and the future of aviation. *ComputerWeekly*, Nov 2014.

[Bar14] M. Barr. Bookout v. Toyota. Available at `http://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf`, last accessed August 30, 2016, 2014.

[BBFM99] P. Behm, P. Benoit, A. Faivre, and J.M. Meynadier. Météor: A successful application of B in a large project. In *Proceedings of the World Conference on Formal Methods in the Development of Computing Systems*, pages 369–387. Springer Verlag, 1999. LNCS 1708.

[BEKV94] K. Broda, S. Eisenbach, H. Khoshnevisan, and S. Vickers. *Reasoned Programming*. Prentise Hall International, 1994.

[BI 16] BI Intelligence. Heres how the Internet of Things will explode by 2020, Aug 2016.

[BK08] C. Baier and J. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BKK03] D.M. Berry, E. Kamstries, and M.M. Krieger. From contract drafting to software specification: Linguistic sources of ambiguity. Available at `se.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf`, November 2003.

[BL16] S. Borowiec and T. Lien. AlphaGo beats human Go champ in milestone for artificial intelligence. *Los Angeles Times, March 12*, 2016.

[Blo18] Tricentis Blog. Real life examples of software development failures, December 2018. Available at `https://www.tricentis.com/blog/real-life-examples-of-software-development-failures`; last accessed July 15, 2020.

[Boo54] G. Boole. *An Investigation of the Laws of Thought*. Dover, New York, 1854.

[Boo91] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, Menlo Park, CA, 1991.

[Bos14] N. Bostrom. *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press, 2014.

[Bow] J. Bowen. World wide web virtual library: The Z method. `http://www.afm.sbu.ac.uk/z`.

[BS03] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[CAD17] Cadp web site, 2017. `http://cadp.inria.fr`.

[Cam] Cambridge University and Technical University of Munich. *Isabelle: A generic theorem proving environment*. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.

[CBM17] Cbmc web site, 2017. `http://www.cprover.org/cbmc`.

[CE81] E.M. Clarke and A. Emerson. Synthesis of synchronous skeletons for branching time temporal logic. In D. Kozen, editor, *Logic of Programs Workshop*. Springer Verlag, 1981. LNCS 131.

[Cer17] V.G. Cerf. In praise of under-specification? *Communications of the ACM*, 60(8):7–7, 2017.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CH00] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[Cha05] R.N. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, 2005.

[Chu] E.H. Chudler. Neuroscience for kids. `https://faculty.washington.edu/chudler/what.html`, last aaccessed April 15, 2016.

[CM90] R.H. Cobb and H.D. Mills. Engineering software under statistical quality control. *IEEE Software*, (6):45–54, 1990.

[Com99] President's Information Technology Advisory Committee. Information technology research: Investing in our future. report to the u.s. president. Technical report, National Coordination Office for Information Technology Research and Development, February 24 1999. Available at `http://www.nitrd.gov/pitac/report`, last accessed August 30, 2016.

[CY89] P. Coad and E. Yourdon. *OOA-Object-Oriented Analysis*. Englewood Cliffs New Jersey: Prentice Hall, 1989.

[DHH01] M. Dwyer, J. Hatcliff, and R. Howell. Slides for course on Software Specifications (CIS771), Kansas State University. Personal communication, November 2001.

[Din16] J. Dingel. Complexity is the only constant: Trends in computing and their relevance for model driven engineering. In *International Conference on Graph Transformation (ICGT'16)*, Vienna, Austria, July 2016.

[Din22] J. Dingel. CISC/CMPE 422/835 GitHub repository, 2022. Available at `https://github.com/CISC422`.

[dM93] J.P. Potocki de Montalk. Computer software in civil aircraft. *Cockpit/Avionics Engineering*, 17(1):17–23, 1993.

[Fre03] G. Frege. Grundgesetze der Arithmetik, begriffsschriftlich abgeleitet, 1903. Volumes I and II (Jena).

[FS97] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modelling Language*. Addison Wesley, 1997.

[Ge93] John V. Guttag and James J. Horning (editors). *Larch: Languages and Tools for Formal Specification*. Springer Verlag, 1993.

[Gen69] G. Gentzen. Investigations into logical deduction. In M.E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–129. North Holland, 1969.

[Gla02] R.L. Glass. Sorting out software complexity. *Communications of the ACM*, 45(11):19–21, 2002.

[GM93] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gri03] K. Grimm. Software technology in an automotive company — major challenges. In *International Conference on Software Engineering (ICSE'03)*, 2003.

[Gro95] The Standish Group. Chaos. Technical Report T23E-T10E, The Standish Group, 1995.

[Gro14] Object Modelling Group. Object Constraint Language Specification. Technical report, OMG, 2014. `https://www.omg.org/spec/OCL`.

[GW89] D. Gause and G. Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House Publishing, 1989.

[HD01] T. Homer-Dixon. *The Ingenuity Gap*. Vintage Canada, 2001.

[Hol03] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[HR00] M. Huth and M. Ryan. *Logic in Computer Science: Modeling and reasoning about systems*. Cambridge University Press, 2000.

[HWZ17] Q. Huang, W. Wang, and Q. Zhang. Your glasses know your diet: Dietary monitoring using electromyography sensors. *IEEE Internet of Things Journal 4(3).*, 4(3), June 2017.

[Hyp22] Hypothesis, 2022. Available at `https://hypothesis.works`.

[(IS11] International Standards Organization (ISO). Road vehicles – functional safety (iso 26262), 2011.

[Jac98] M. Jackson. *Software Requirements & Specifications*. Addison Wesley, 1998.

[Jac99] D. Jackson. A Comparison of Object Modelling Notations: Alloy, UML and Z. MIT, Lab for Computer Science, August 1999.

[Jac00] D. Jackson. Lecture notes for 6.170 (Laboratory in Software Engineering). MIT Laboratory for Computer Science, October 2000.

[Jac01] D. Jackson. Micromodels of software: Modelling & analysis with alloy. Draft, available at `http://sdg.lcs.mit.edu/alloy/book.pdf`, November 2001.

[Jac06a] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[Jac06b] M. Jackson. *Software Abstractions: Logic, Languge, and Analysis*. MIT Press, 2006.

[JCJG92] I. Jacobson, M. Christerson, P. Jonsson, and O. Gunnar. *Object-oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, 1992.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM*. Prentise Hall, 1986.

[JPF17] Java pathfinder web site, 2017. `http://babelfish.arc.nasa.gov/trac/jpf`.

[Kel11] K. Kelly. *What Technology Wants*. Penguin Books, 2011.

[Kir11] D. Kirkpatrick. Now every company is a software company. *Forbes*, November 30 2011.

[KMM00a] M. Kaufmann, P. Manolios, and J. Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[KMM00b] M. Kaufmann, P. Manolios, and J. Strother Moore, editors. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[Koo14] P. Koopman. A case study of Toyota unintended acceleration and software safety. Available at `https://betterembsw.blogspot.ca/2014/09/a-case-study-of-toyota-unintended.html`, last accessed August 30, 2016, September 2014.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Lev11] N. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2011.

[LG00] B. Liskov and J. Guttag. *Program Development in Java*. Addison Wesley, 2000.

[LLF+96] J.L. Lions, L. Lbeck, J.L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D.M. Thomson, and C. O'Halloran. Ariane 5: Flight 501 failure, report by the inquiry board, July 1996.

[LN17] U. Lindqvist and P.G. Neumann. The future of the Internet of Things. *Communications of the ACM*, 60(2):26–30, 2017.

[Loh16] S. Lohr. G.E., the 124-year-old software start-up. *The New York Times*, August 27 2016.

[LT93] N. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[LTSF14] H. Li, S. Thompson, P.L. Seijas, and M. Francisco. Automating property-based testing of evolving web services. In *Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, 2014.

[Mar11] J. Markoff. The iPad in your hand: As fast as a supercomputer of yore. New York Times article based on interview with Dr. Jack Dongarra, May 9, 2011 2011. `http://bits.blogs.nytimes.com/2011/05/09/the-ipad-in-your-hand-as-fast-as-a-supercomputer-of-yore`, last accessed April 15, 2016.

[McCa] J.C. McCallum. Memory prices (1957-2015). `http://www.jcmit.com/memoryprice.htm`, last accessed March 2016.

[McCb] D. McCandless. Information is beautiful: Million lines of code. `http://www.informationisbeautiful.net/visualizations/million-lines-of-code`, last accessed April 15, 2016.

[McC93] S. McConnell. *Code Complete*. Microsoft Press, 1993.

[McC16] Caitie McCaffrey. The verification of a distributed system. *Communications of the ACM*, 59(2):52–55, Feb 2016.

[MDL90] H.D. Mills, M. Dyer, and R.C. Linger. Cleanroom software engineering. *IEEE Software*, pages 19–25, September 1990.

[Mey85] B. Meyer. On formalism in specifications. *IEEE Software*, 3(1):6–25, 1985.

[Mic] Microsoft Research. Research in software engineering. Available at `https://www.microsoft.com/en-us/research/group/research-in-software-engineering-rise`.

[Min08] J.R. Minkel. The 2003 Northeast Blackout — five years later. *Scientific American*, August 2008.

[MMZ01] *Chaff: engineering an efficient SAT solver*, 2001. Las Vegas, Nevada, United States.

[Mod17] List of model checking tools, 2017. `https://en.wikipedia.org/wiki/List_of_model_checking_tools`.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer, 1995.

[NRZ+15] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, April 2015.

[NuS17] Nusmv web site, 2017. `http://nusmv.fbk.eu`.

[oCP] ACM Committee on Computers and Public Policy. The Risks Digest: A forum on risks to the public in computers and related systems. Moderated by Peter Neumann. Available at `http://catless.ncl.ac.uk/Risks`, last accessed August 30, 2016.

[O'H15] P. O'Hearn. From categorical logic to facebook engineering. In *30th IEEE/ACM Symposium on Logic in Computer Science*, pages 17–21, 2015.

[oHHSA02] U.S. Department of Health, Food Human Services, and Drug Administration. Principles of software validation; final guidance for industry and fda staff, 2002. Available at `https://www.fda.gov/regulatory-information/search-fda-guidance-documents/general-principles-software-validation`; last accessed July 15, 2020.

[ORA] ORA Canada. *Never: An automated deduction system*. `http://www.ora.on.ca/eves.html`.

[oTFHAU] U.S. Department of Transportation Federal Highway Administration (USDOT). Frequently asked questions – part 4 – highway traffic signals. `http://mutcd.fhwa.dot.gov/knowledge/faqs/faq_part4.htm`, last accessed August 30, 2016.

[Oxf] Oxford University. *Just Another Proof Editor (JAPE)*. `http://users.comlab.ox.ac.uk/bernard.sufrin/jape.html`.

[Pat16] J. Patel. Software is still eating the world. *Techcrunch*, June 7 2016.

[Pet96] I. Peterson. *Fatal Defect: Chasing Killer Computer Bugs*. Vintage Books, 1996.

[Pra65] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Almquist & Wiksell, 1965.

[Pre05] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005. Sixth Edition.

[Pri17] Prism web site, 2017. `http://www.prismmodelchecker.org`.

[PS99] R. Pooley and P. Stevens. *Using UML. Software Engineering with Objects and Components*. Addison Wesley, 1999.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium on Programming*, pages 337–350, 1981.

[Qui22] Quickcheck: Automatic testing of haskell programs, 2022. Available at `https://hackage.haskell.org/package/QuickCheck`.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modelling and Design*. Prentise Hall, 1991.

[Red20] A. Redko. In praise of the thoughful design: how property-based testing helps me to be a better developer, 2020. Available at `https://www.javacodegeeks.com/2020/02/in-praise-of-the-thoughful-design-how-property-based-testing-helps`

[Ric07] D. Rice. *Geekonomics: The Real Cost of Insecure Software*. Addison-Wesley, 2007.

[Rig96] F. Riguzzi. A survey of software metrics. Technical Report DEIS-LIA-96-010, Università degli Studi di Bologna, 1996.

[Rot20] A. Rothuis. Testing general rules through property-based tests, 2020. Available at `https://www.arothuis.nl/posts/property-based-testing-rock-paper-scissors`.

[RTC12] RTCA. Do-178c, software considerations in airborne systems and equipment certification. Available at `https://en.wikipedia.org/wiki/DO-178C`, January 2012.

[RTI02] RTI. The economic impacts of inadequate infrastructure for software testing. Technical Report Planning Report 02-3, National Institute of Standards & Technology (NIST), May 2002.

[RWCP10] D. Redman, D. Ward, J. Chilenski, and G. Pollari. Virtual integration for improved system design. In *Proceedings of the First Analytic Virtual Integration of Cyber-Physical Systems Workshop*, pages 57–64, November 2010.

[SAT20] The international SAT competition web page, 2020. Available at `http://www.satcompetition.org`; last accessed July 9, 2020.

[Sau14] L. Saunders. Don't make these tax mistakes: Fifteen common tax-filing errors that can cost you dearly. *The Wall Street Journal*, January 31, 2014.

[Sch01] S. Schneider. *The B-Method: An Introduction*. Palgrave, 2001.

[SMF03] A.N. Clark S.J. Mellor and T. Futagami. Model-driven development. *IEEE Software*, (13), September/October 2003.

[SMK+01] I. Stoica, R. Morris, D. Karger, F.M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, pages 149–160, 2001.

[Som04] I. Sommerville. *Software Engineering*. Addison-Wesley, 2004. Seventh edition.

[Som17] James Somers. The coming software apocalypse. *The Atlantic*, Sept 2017.

[Spi89] J.M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 1989.

[Spi17] Spin web site, 2017. `http://spinroot.com`.

[SRI] SRI International. Computer Science Laboratory. *Prototype Verification System (PVS)*. `http://pvs.csl.sri.com`.

[Str12] B. Stroustrup. Software development for infrastructure. *IEEE Computer*, 45(1):47–58, January 2012.

[SW18] Chris Stokel-Walker. The commas that cost companies millions. BBC Capital, July 2018. Available at `http://www.bbc.com/capital/story/20180723-the-commas-that-cost-companies-millions`.

[Tai96] J.A. Tainter. Complexity, problem solving, and sustainable societies. In R. Costanza, O. Segura, and J. Martinez-Alier, editors, *Getting Down to Earth: Practical Applications of Ecological Economics*. Island Press, 1996.

[UPP17] Uppaal web site, 2017. `http://www.uppaal.org`.

[vEVD89] P.H.J. van Eijk, C. A. Vissers, and M. Diaz. *The formal description technique LOTOS*. Elsevier Science Publishers B.V., 1989.

[War13] D. Ward. Avsis system architecture virtual integration program: Proof of concept demonstrations. INCOSE MBSE Workshop, January 27 2013.

[Wea15] O. Weatherall. Why the flash crash really matters. *Nautilus*, April 2015.

[Wik22a] Wikipedia. Agile software development, 2022. Available at `https://en.wikipedia.org/wiki/Agile_software_development`.

[Wik22b] Wikipedia. Devops, 2022. Available at `https://en.wikipedia.org/wiki/DevOps`.

[Wik22c] Wikipedia. Fluent interface, 2022. Available at `https://en.wikipedia.org/wiki/Fluent_interface`.

[Win95] J. Wing. Hints to specifiers. Technical Report CMU-CS-95-118R, CMU, 1995. Available at www.cs.cmu.edu/ wing.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language. Precise Modeling with UML*. Object Technology Series. Addison Wesley, 1999.

[WLBF09] J. Woodcook, P.G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), October 2009.

[Zav12] P. Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):50–57, April 2012.

[Zha97] *SATO: An efficient propositional prover*, July 1997.

Paper 1

Vinton G. Cerf. In Praise of Under-Specification? Communications of the ACM 60(8):7-7. August 2017.

A short reflection by the vice president and Chief Internet Evangelist at Google on how it was beneficial for the evolution and success of the internet that the standards and protocols the internet is based on are specifications and thus declarative, rather than operational.

Paper 2

Daniel Jackson. Alloy: A Language and Tool for Exploring Software Designs. Communications of the ACM 62(9):66-76. September 2019.

Article by the author of Alloy briefly describing the language and tool, together with some applications ranging from critical systems to network protocols, web security, and code verification.

Paper 3

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. Communications of the ACM 58(4):66-73. April 2015.

Engineers at Amazon Web Services (AWS) have used formal specification and model checking to help solve difficult design problems in critical systems since 2011. This article describes the motivation and experience, and what has worked well, and what has not.