Software Engineering II
Theory

Christian Rossi

Academic Year 2023-2024

Abstract

The objective of the course is to teach the principals methods and processes of software engineering needed to develop complex and qualitative software.

The course covers the following arguments:

- Software process and its organization.
- Modelling languages.
- Requirements analysis and definition.
- Software development methods and tools.
- Approaches for verify and validate the software.

Contents

1 Introduction **2**
1.1 Definition . . . 2
1.2 History . . . 2
1.3 The process and product . . . 3
1.4 Development process . . . 4

2 Requirements engineering **7**
2.1 Definition . . . 7
2.2 Importance and difficulties . . . 7
2.3 Requirement engineering process . . . 8
2.4 World-machine relationship . . . 9
2.5 Elicitation of requirements . . . 10
2.6 Modeling requirements . . . 11
2.7 Use cases and requirements . . . 12
2.8 Requirements-level class diagrams . . . 14
2.9 Dynamic modeling . . . 14

3 Alloy **16**
3.1 Definition . . . 16
3.2 Introduction . . . 16
3.3 Syntax . . . 17
3.4 Address book example . . . 19
3.5 Family relations example . . . 21

Chapter 1

Introduction

1.1 Definition

The field of software engineering aims to find answers to the many problems that software development projects are likely to meet when constructing large software systems. Such systems are complex because of their sheer size, because they are developed by a team involving people from different disciplines, and because they will be modified regularly to meet changing requirements, both during development and after installation.

Definition

> *Software engineering* is a methodological and managerial discipline concerning the systematic production and maintenance of software products that are developed and maintained within anticipated and controlled time and cost limits.

The programmer develops a complete software and works on known specifications individually. Instead, the software engineer identifies requirements and develop specifications, designs components that will be combined with others and works in a team. The main skills of a software engineer are: technical, managerial, cognitive, organizational.

1.2 History

Initially, the software was considered as an art. The computers were used for computing to solve mathematical problems and the designers were also the users. The first programs were created with low-level languages and had high resources constraints.

When the request for new custom software exploded the art became a craft: the developer started to create programs also for the people with new high-level languages. At the end of this period there were a "software crisis" due to increasing software complexity and lack of effective software development techniques.

To solve this problem in 1968 was defined the term *software engineering* in a NATO conference. The main focuses of this conference was on:

- Development of software and standards.
- Planning and management.
- Automation.
- Modularization.
- Quality verification.

1.3 The process and product

The developing of a software program needs a process. Both software and processes have a quality and the software engineer needs to reach the optimal quality because the process modifies the final output.

The software is different from traditional types of products because it is:

1. Intangible (difficult to describe and evaluate).
2. Malleable.
3. Human intensive (does not involve any trivial manufacturing process).

The quality of the software is influenced by the following variables: development technology, process quality, people quality, cost, time and schedule. The software quality attribute are:

- Correctness: software is correct if it satisfies the specifications.
- Reliability: probability of absence of failures for a certain time period.
- Robustness: software behaves reasonably even in unforeseen circumstances.
- Performance: efficient use of resources.
- Usability: expected users find the system easy to use.

- Maintainability.
- Reusability: similar to maintainability but applies to components.
- Portability: adaption to different target environments.
- Interoperability: coexist and cooperate with other applications.

The process quality attribute are:

- Productivity.
- Unity of effort (person month).
- Delivered item (lines of code and function points).
- Timeliness: ability to respond to change requests in a timely fashion.

1.4 Development process

Initially there were no reference model, so it was simple code&fix. As a reaction to the software crisis mentioned before it became necessary to have a model. The first complete model was the "waterfall". The key requirements of this model are:

1. Identify phases and activities.
2. Force linear progression from a phase to the next (without returns).
3. Standardize outputs from each phase.
4. Software is considered like manufacturing.

After this model many other flexible processes were proposed: iterative models, agile movement and DevOps.

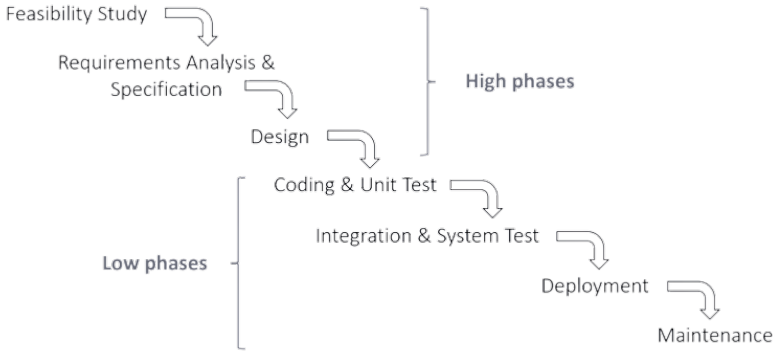


Figure 1.1: Waterfall process model

The main phases shown in the image are:

1. Feasibility study and project estimation: determines wheather the project should be started, the possible alternatives and needed resources. This phase produces a *Feasibility Study Document* which contains: preliminary problem description, scenarios describing possible solutions, cost and schedule for the different alternatives.

2. Requirement analysis and specification: analyze the domain in which the application takes place, identify requirements and derive specification for the software. This phase produces *Requirement Analysis and Specification Document*.

3. Design: defines the software architecture (components, relation and interactions among components). The goal is to support concurrent development and separate responsibilities. It produces the *Design Document*.

4. Coding and unit test: each module is implemented and tested. Inspection can be used as an additional quality assurance approach. Programs include their documentation.

5. Integration and system test: the modules are integrated into systems and integrated systems are tested. This phase and the previous may be integrated in an incremental implementation scheme.

6. Deployment.

7. Maintenance: the maintenance can be:

    - Corrective: deals with the repair of faults or defects found.
    - Adaptive: consist of adapting software to changes in the environment.
    - Perfective: deals with accommodating to new or changed user requirements.
    - Preventive: concerns activities aimed at increasing the system's maintainability.

The principal problems with software evolution are:

- It is almost never anticipated and planned.

- Software is very easy to change (changes applied directly to the code that causes inconsistent state of project documents).

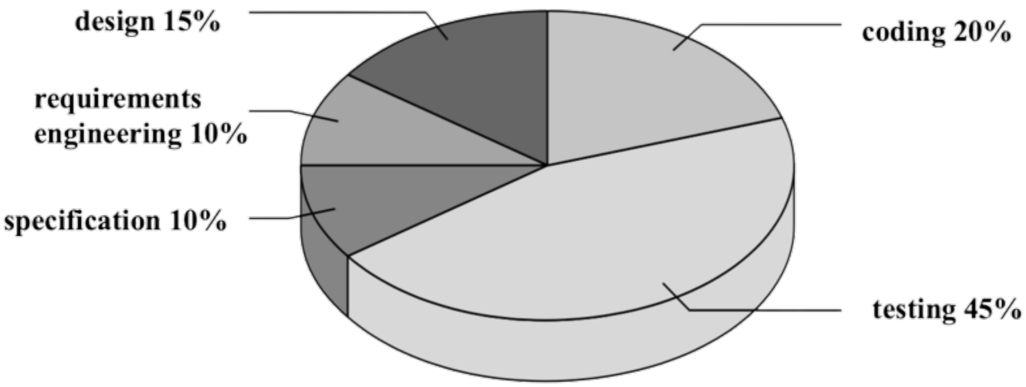


Figure 1.2: Effort in each phase

To face properly the evolution we need a good engineering practice that consist in two main steps: modify the design and then change the implementation and apply changes consistently in all documents. In fact, one of the main goal of software engineering is to create software that must be designed to accommodate future changes reliably and cheaply.

Waterfall model is a black-box system because the company that requests the software makes requirements and doesn't interact during the development phase. If we need more transparency with the customer we need to use a different development model (that allows the customer to give feedback regularly). With every interaction with the customer is possible to check two main things:

- Validation: check if the product follows the customer's requests.
- Verification: check if the product works in the right way.

The idea of flexible process is to adapt to changes, in particular the requirements and specification. The idea is to have incremental processes and be able to get feedback on increments. They exists in many forms, for example: SCRUM, extreme programming, incremental releases and rapid prototyping, DevOps, . . .

Chapter 2

Requirements engineering

2.1 Definition

The primary measure of success of a software system is the degree to which it meets the purpose for which it was intended.

Definition

> Software systems *requirements engineering* is the process of discovering that purpose, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation.

The important issues of this phase are: identify stakeholders, identify their needs, produce documentation and analyse, communicate and implement requirements. Another possible definition is the following.

Definition

> *Requirements engineering* is the branch of software engineering concerned with the real-world goals for, function of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software engineering behaviour, and to their evolution over time and across software families.

2.2 Importance and difficulties

The requirements given from the customer can be classified in three main types:

- Functional: describes the interaction between the system and its environment independent from implementation. They are the main goals that the software has to fulfill.

- Nonfunctional: user visible aspects of the system not directly related to functional behaviour.
- Constraints: imposed by the client or the environment in which the system operates.

The nonfunctional requirements are constraints on how functionality has to be provided to the end user. They are independent of application domain but the application domain determines: their relevance and their prioritization. They are also called Quality of Service attribute.

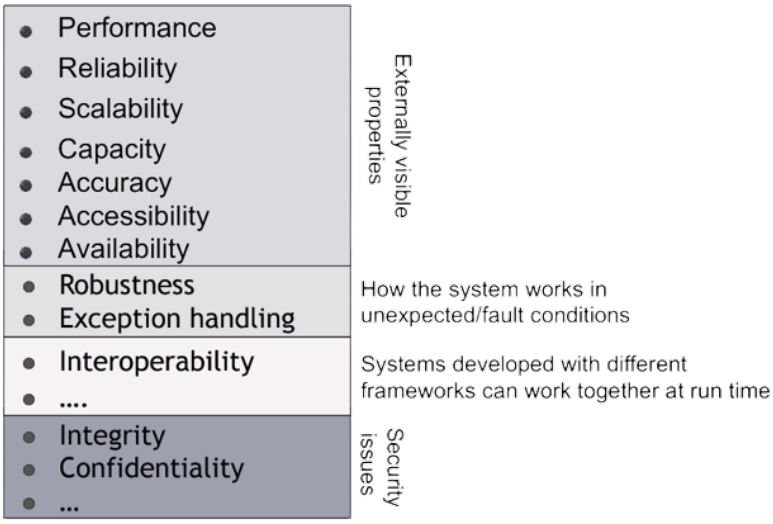
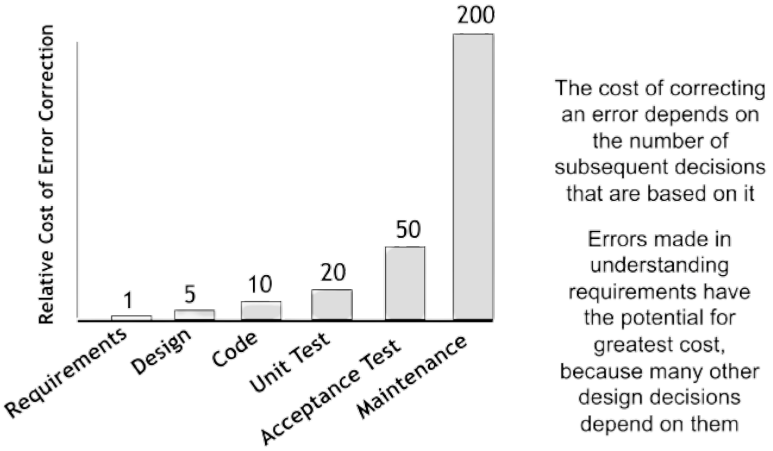


Figure 2.1: Some relevant QoS characteristics

2.3 Requirement engineering process

Poor requirements are ubiquitous. Requirement engineering is also hard and critical because a problem with the initial phases can be up to two hundred times costly in the final phase.



The cost of correcting an error depends on the number of subsequent decisions that are based on it

Errors made in understanding requirements have the potential for greatest cost, because many other design decisions depend on them

Figure 2.2: Cost of late correction [Boehm, 1981]

Requirement engineering is so complex because of: composite systems. more than one system, multiple abstraction levels, multiple concerns and multiple stakeholders with different background.

The requirement engineers needs to:

- Eliciting information (project objectives, context and scope; domain scope and requirements).
- Modelling and analysis (goals, objects, use cases and scenarios).
- Communicating requirements (analysis feedback, RASD document, system prototypes).
- Negotiating and agreeing requirements (handling conflicts and risks; helping in requirement selection and prioritization).
- Managing and evolving requirements (managing requirements during development: backward and forward traceability; managing requirements changes and their impacts).

2.4 World-machine relationship

The machine indicates the portion of system to be developed, while world indicates the portion of the real-world affected by the machine. The purpose of the machine is always in the world.

Requirements engineering is concerned with phenomena occurring in the world as opposed to phenomena occurring inside the machine. We can say that requirements models are models of the world.

Some world phenomena are shared with the machine. This type of phenomena can be:

- Controlled by the world and observed by the machine.
- Controlled by the machine and observed by the world.

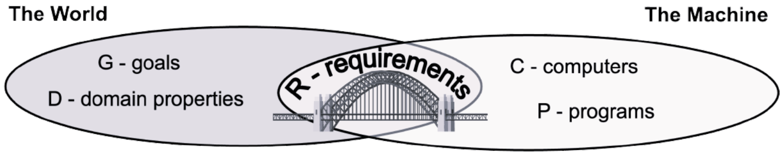


Figure 2.3: Goals, domain assumptions, and requirements

Goals are prescriptive assertions formulated in terms of world phenomena (not necessarily shared). Domains properties/assumptions are descriptive assertions assumed to hold in the world. Requirements are prescriptive assertions formulated in terms of shared phenomena.

The requirements R are complete if:

- R ensures satisfaction of the goals G in the context of the domain properties D, this means that $R \wedge D \models G$.
- G adequately capture all the stakeholders' needs.
- D represent valid properties/assumptions about the world.

2.5 Elicitation of requirements

The complexity in requirement engineering can be coped with:

- Adopting different approaches and strategies and combining the results reached with all of them.
- Being as close as possible to stakeholders.
- Letting stakeholders describing their viewpoints.

The scenario can be generalized in term of:

- Participation actors.
- Describe the entry condition.

- Describe the flow of events.

- Describe the exit condition.

- Describe exceptions.

- Describe special requirements.

2.6 Modeling requirements

Definition

> A *model* is a representation in a certain medium of something in the same or another medium. The model captures the important aspects of the thing being modeled and simplifies or omits the rest.

The reality R is composed by: real things, people, processes and relationship. The model M is an abstraction of things, people, processes and relationship between these abstraction.

The reality needs to be interpreted (I) with a mapping function. To have a good model the relationships that are valid in the reality R need to be valid also in the model M.

$$\begin{tikzcd}
\mathbf{M} \arrow[r, "f_M"] & \mathbf{M} \\
\mathbf{R} \arrow[u, "\mathbf{I}"] \arrow[r, "f_R"'] & \mathbf{R} \arrow[u, "\mathbf{I}"']
\end{tikzcd}$$

Figure 2.4: Relationship between model M and reality R

The software models are used for:

- Capture and precisely state requirements and domain knowledge.

- Think about the design of a software system Generate usable work products.

- Give a simplified view of complex systems Evaluate and simulate a complex system.

- Generate potential configurations of systems.

The principal modeling issues are coherence (different views of the system must be coherent) and variation in interpretation and ambiguity (define where different interpretation of the model are acceptable).

In requirements engineering we should only model:

- The objects and people that are of interest for the given problem.
- The relevant phenomena.
- The goals, requirements, and domain assumptions.

The tool that we can use for modeling are:

- Natural language (English, Italian, . . .):
  - Pros: simplicity of use.
  - Cons: high level of ambiguity, it is easy to forget to include relevant information.
- Formal language (FOL, Alloy, Z, . . .):
  - Pros: possibility to use tool to support analysis and validation, the approach forces the user in specifying all relevant details.
  - Cons: you need to be expert in the use of the language.
- Semi-formal language (UML):
  - Pros: simpler than a formal language, impose some kind of structure in the models.
  - Cons: not amenable for automated analysis, some level of ambiguity.
- Mixed approach: use a semi-formal language for the basics. Comment and complement the semi-formal models with explanatory informal text. use a formal language for the most critical parts.

2.7 Use cases and requirements

The main steps when formulating use cases are:

1. Name the use case
2. Find the actors: generalize the concrete names to participating actors.

3. Concentrate on the flows of events, entry and exit condition using natural language.

4. Focus on exceptional cases and special requirements.

Each use case may lead to one or more requirements.

A use case is a flow of events in the system, including interaction with actors. The use cases are initialized by an actor and has a termination condition.

Definition

The *use case model* is the set of all use cases specifying the complete functionality of the system.

Definition

A *use case association* is a relationship between use cases. The principal types of use case association are:

- Include(a use case uses another use case).
- Extends (a use case extends another use case).
- Generalization (an abstract use case has several different specializations).

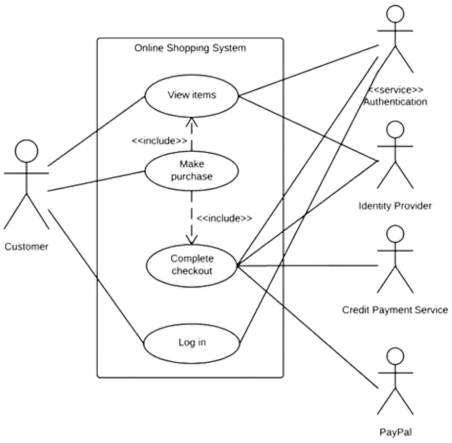


Figure 2.5: Use case model example

2.8 Requirements-level class diagrams

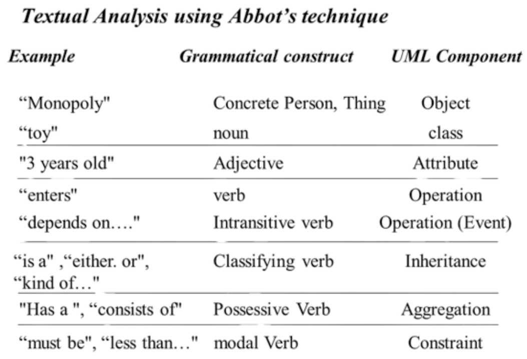
The requirements-level class diagrams are conceptual models for the application domain. They may model objects that will not be represented in the software-to-be. Usually, the do not attach operations to objects: it's best to postpone this kind of decisions until software design.

To find objects and classes we need to:

- Analyze any description of the problem and application domain you may have.

- Analyze your scenarios and use cases descriptions.

Finding objects is the central piece in object modeling. A possible tool to use in the analysis is the Abbott Textual Analysis also called noun-verb analysis: nouns are good candidates for classes and verbs are good candidates for associations and operations.



Textual Analysis using Abbot's technique

Example	*Grammatical construct*	*UML Component*
“Monopoly"	Concrete Person, Thing	Object
“toy"	noun	class
"3 years old"	Adjective	Attribute
“enters"	verb	Operation
“depends on…."	Intransitive verb	Operation (Event)
“is a" ,“either. or", “kind of…"	Classifying verb	Inheritance
"Has a ", “consists of"	Possessive Verb	Aggregation
“must be", “less than…"	modal Verb	Constraint

Figure 2.6: Abbott Textual Analysis example

2.9 Dynamic modeling

The purpose of the dynamic modeling is to supply methods to model interactions, behaviours of participants and workflow. This can be done with: sequence diagram, state machine diagram and activity diagram. Some objects can be found whilst completing those diagrams.

The sequence diagram is created following the flow of events in the use case diagram. A sequence diagram is a graphical description of objects participating in a use case scenario using a Directed Acyclic Graph notation. The principal rules to create a sequence diagrams are:

- An event always has a sender and a receiver.

- The representation of the event is sometimes called a message.
- Find sender and receiver for each event.

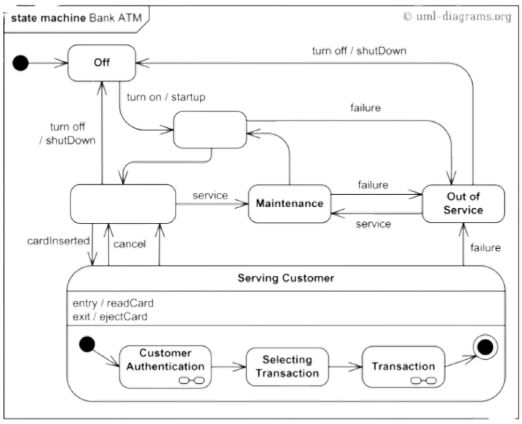


Figure 2.7: Example of a state diagram

For a good dynamic modeling we ha to construct a model only for classes with significant dynamic behaviour and consider only relevant attributes. We have also to look ate the granularity of the application when deciding on actions and activities and reduce notational clutter.

Chapter 3

Alloy

3.1 Definition

Alloy is a formal notation for specifying models of systems and software. It looks like a declarative object oriented language, but also has a mathematical foundation.

Alloy comes with a supporting tool to simulate specifications and performs property verification.

Alloy has been created to offer an expressive power similar to the Z language as well as the strong automated analysis of the SMW model checker.

Alloy can be used in requirement engineering to formally describe the domain and its properties, or operations that the machine has to provide. In software design Alloy can be used to formally model components and their interactions.

3.2 Introduction

Alloy is a mixture of first order logic and relational calculus. The main elements of this formal language are:

- Carefully chosen subset of relational algebra (uniform model for individuals, sets and relations; no high-order relations).
- Almost no arithmetic.
- Modules and hierarchies.
- Suitable for small, explanatory specification.
- Powerful and fast analysis tool.

3.3 Syntax

Alloy shows bounded snapshots of the world that satisfy the specification. Allow does bounded exhaustive search for counterexample to a claimed property using SAT.

Definition

Atoms are Alloy's primitive entities (indivisible, immutable and uninterpreted).

Definition

Relations associate atoms with one another (set of tuples; tuples are sequence of atoms).

The relations in Alloy are typed. The relation type is determined by the declaration of the relation. The basic Alloy relations type are: *none* (empty set), *univ* (universal set) and *iden* (identity relation). The logic operators in Alloy are:

- Union $\cup$.
- Intersection &.
- Difference $-$.
- Subset *in*.
- Equality $=$.
- Cross product $\rightarrow$ (similar to a natural join).
- Dot join . or [ ] (the last element of the first relation joins on the corresponding first elements of the second relation, and then removes the element from the relation).

The possible binary closures on the relations are:

- Transpose $\sim$ (inverts the order of the elements in the relation).
- Transitive $\wedge$ ($^\wedge r = r + r.r + r.r.r + \dots$).
- Reflexive transitive $*$ ($^* r = iden +^\wedge r$)

The possible restrictions are:

- Domain restriction <:, that restricts the elements on the left side to the set on the right side.
- Range restriction :>, that is same as before but the relations are inverted.
- Override ++, that removes the tuples on the left that are in the right relations and adds all the remaining relations of the right relation.

The Alloy Boolean operators are the following: negation (! or *not*), conjunction (& or *and*), disjunction (|| or *or*), implication ($\implies$ or *implies*), alternative (, or *else*) and bi-implication ($\iff$ or iff). The alloy logic quantifiers are:

- *all*: holds for every element.
- *some*: holds for at least one element.
- *no*: holds for no elements.
- *lone*: holds for at most one element.
- *one*: holds for exactly one element.

To define a relation with a singleton we can use the following declaration:

$$x \ : \ m\, e$$

Where x is the name of the relation, m is the multiplicity of the element (that can be: *set*, *one*, *lone* or *some*) and e is the name of the element in the relation. If the relation is composed by couple the declaration became like this:

$$r \ : \ A\, m \ \rightarrow \ n\, B$$

Where r is the name of the relation, A and B are the name of the elements with multiplicity m and n respectively.

let is used to define a formula or expression that can be reused. Other useful operators are: $\#r$ (that define the number of tuples in r), $0, 1, \ldots$ (integer used to define the value of some variables or constants), + (plus), $-$ (minus), all the comparison operators (<, <=, =, =>, >). There is also the operator *sum* that adds all the elements in the selected tuple.

It is possible to define a mutable relation with the keyword *var*, $after$, *always*, and *eventually*. You can define enumeration, using the keyword *enum*. Other keywords: *historically*, $before$, *once*.

3.4 Address book example

Imagine that we are asked to model a very simple address book. The books that contain a bunch of addresses linked to the corresponding names.

In this example we have three entities, which are: *Name*, *Addr* and *Book*. *addr* is linking *Name* to *Addr* within the context of *Book*. To indicate this relation we use the keyword *lone* that indicates that each *Name* can correspond at most one *Addr*. We can resume the previous statements as:

```
sig Name {}
sig Addr {}
sig Book {
    addr: Name -> lone Addr
}
```

This specification creates the following relations:

- Sets are unary relations.
- Scalars are singleton sets.
- The ternary relation involving the three predicates.

We can declare a new predicate with the keyword *pred*.

```
pred show {}
run show for 3 but exactly 1 Book
```

Where the second line indicates that we need to find at most three elements for every *Book*. The predicate *show* defined previously is empty and return always *true*. Now we can define a predicate with some argument, for example:

```
pred show [b:Book]{
    # b.addr > 1
}
run show for 3 but exactly 1 Book
```

The predicate (consistent) in the previous example adds a constraint on the number of *Address* relations in a given *Book*. The predicate (consistent) in the following example adds a constraint on the number of different *Address* that appears in the *Book*.

```
pred show [b:Book]{
    # b.addr > 1
    # Name.(b.addr) > 1
}
run show for 3 but exactly 1 Book
```

The predicate (inconsistent) in the following example contains the keyword *some* that indicates the existence of an element. In this case we have only one *Book* so the tool will say that no instances can be found.

```
pred add [b:Book]{
    # b.addr > 1
    some n:Name | # n.(b.addr) > 1
}
run show for 3 but exactly 1 Book
```

All the previous predicates are static because they doesn't change the signature. In Alloy there are also dynamic predicates for dynamic analysis. For example we can define a predicate that adds an *Address* and *Name* to a *Book* in the following way:

```
pred add [b,b':Book, n:Name,a:Addr]{
     b'.addr=b.addr + n -> a
}
pred showAdd [b,b':Book, n:Name,a:Addr]{
     add[b,b',n,a]
     #Name.(b'.addr) > 1
}
run showAdd
```

We can now define a predicate for the *Book* deletions.

```
pred del [b,b':Book, n:Name]{
     b'.addr=b.addr - n -> Addr
}
```

We can check if running a delete after an add returns us in the initial situation or not by using an *assertion*:

```
assert delRevertsAdd{
    all b1,b2,b3:Book,n:Name,a:Addr
    add[b1,b2,n,a] and del[b2,b3,n]
    implies b1.addr=b3.addr
}
```

While checking an assertion, Alloy searches for counterexamples. In this case we will find a counterexample so the assert will result *false*. To correct the assert we need to modify it in the following way:

```
assert delUndoesAdd{
    all b1,b2,b3:Book,n:Name,a:Addr |
    no n.(b1.addr) and add[b1,b2,n,a] and del[b1,b2,n]
    implies b1.addr=b3.addr
}
```

We can also need to get some signature. To do that we can use the Alloy functions. For example we can declare a function that search a certain *Book* and return a set of *Address*:

```
fun lookup[b:Book,n:Name]: set Addr{
    n.(n.addr)
}
```

3.5 Family relations example

We now consider a family relationship tree. First of all we have to define a generic person, that can be a men or a woman.

```
abstract sig Person {
    father: lone Man
    mother: lone Woman
}
sig Man extends Person {
    wife: lone Woman
}
sig Woman extends Person {
    husband: lone Man
}
```

We have set that each *Person* has at most one father and one mother (keyword *lone*) because we need a root for the tree. The person at the root needs to have no parents (for example they are unknown). The signature *Person* is *abstract* because it needs to be specialized in one of the subsequent signatures, that are *Man* or *Woman*. Signatures by using keyword *sig* represents a set of atoms. Before this keyword we can define the number of entities that we need (*lone*, *one* or *some*).

Definition

The *fields* of a signature are relations whose domain is a subset of the signature. The keyword *extends* is used to declare a subset of signature.

To get the set of grandpas of a given person we can define a function like this:

```
fun grandpas[p:Person]:set Person {
    p.(mother+father).father
}
pred ownGrandpa[p:Person] {
    p in p.grandpas[p]
}
```

We have also defined a predicate that checks if the person is in the set of grandpas returned by the function *grandpas*. The problem now is that we have not set constraints on relations. To do that we need to define two new operators for binary relations:

- Transitive closure: $\hat{}r = r + r.r + r.r.r + \ldots$
- Reflex transitive closure: $*r = iden + \hat{}r$

We can now define that no one can be the father/mother of himself:

```
fact {
    no p:Person | p in p.^(mother+father)
}
```

We have also to set a constraint that if X is husband of Y, then Y is the wife of X:

```
fact {
    all m:Man,w:Woman | m.wife=w iff w.husband=m
}
fact {
    wife = ~husband
}
```

The two facts are equivalent, but the second has been written using the transpose operator. The fact can contains multiple constraints. So the previous constraints can be written in one fact. The difference between $fact$ and *pred* is that the first are global, while the second needs to be invoked.