Software Engineering II
Exercises

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

CHAPTER 1

Exercises session I

1.1 AdmissionManager

Your company has been tasked to develop a system that handles the admission applications that parents send, on behalf of their children, to the high schools of a metropolitan area. Parents can send admission applications to multiple schools. Before sending an application, they must register their child in the system; the registration includes login credentials (username and password), the personal data of the child (first name, last name, birthdate, etc.), the name of at least one parent, contact information (which must include an email address and a phone number), the name of the last school they attended, and the list of grades (which includes the obtained score, from 1 to 10, for each subject). Each application is assigned an identifier by the system, to allow parents to check its status after sending it (which can be "accepted", "rejected", or "not evaluated"). Parents can withdraw applications previously sent. They can also ask the system to be notified by email when the outcome of the evaluation of an application is available. School administrators use the system to check the applications sent to their schools and to approve/reject them. In particular, they can retrieve the list of applications sent to their schools that have yet to be evaluated; they can also leave comments on the applications, and they can decide to accept or reject the applications. Administrators can also set a preference to receive a notification, in the form of an email, when a new application is sent to their school.

1. Define the goals for the `AdmissionManager` system.
2. Select one of the goals defined in the previous point and define in natural language suitable domain assumptions and requirements to guarantee that the `AdmissionManager` system fulfills the selected goal.
3. Draw a UML Use Case Diagram describing the main use cases of the `AdmissionManager` system.
4. Pick one of the use cases, and define it.

Solution

1. The goals are world phenomena shared between the machine and the real world. They are problem of the real world that the `AdmissionManager` needs to address. We have four examples, which are:

- User sends an application.
- User withdraws an application.
- School administrator evaluates an application.
- User is notified about an application evaluation.

The problem with those goals is that they are only on world side, so they are not well formulated. The term user is ambiguous, it needs to be specified (parents and school administrator). The formulation can be changed to make them correct:

- Parents can manage (send and then monitor) applications to schools on behalf of their children.
- School administrators can manage (check and approve/reject) applications sent to their schools.

2. A domain assumption like ”as soon as an application arrives to the system, a status needs to be assigned to it” is not correct because the status depend on a method in the program and not on something that is granted by the real world. Examples of correct domain assumption, that are not well formulated are:

   - Parents must be registered into the system to issue an application.
   - The system must allow parents to register by providing their email address and personal information.

   In the end, for the first goal we have the following domain assumptions:

   - `AdmissionManager` allows system administrators to open application windows for their schools.
   - `AdmissionManager` allows parents to register into the system and provide contact information and information about their child.
   - `AdmissionManager` allows parents to log into the system using the credentials input at registration time.
   - `AdmissionManager` allows parents to indicate in their profile that they want to be notified when the outcome of an application is available.
   - `AdmissionManager` allows parents to send an application to a school.
   - `AdmissionManager` assigns a unique identifier to each application received.
   - `AdmissionManager` allows parents to see the list of applications sent.
   - `AdmissionManager` allows parents to withdraw an application previously sent.

   The assumption is the following:

   - Parents provide correct information (in particular, contact information) when registering.

   And for the second goal we have the following domain assumptions:

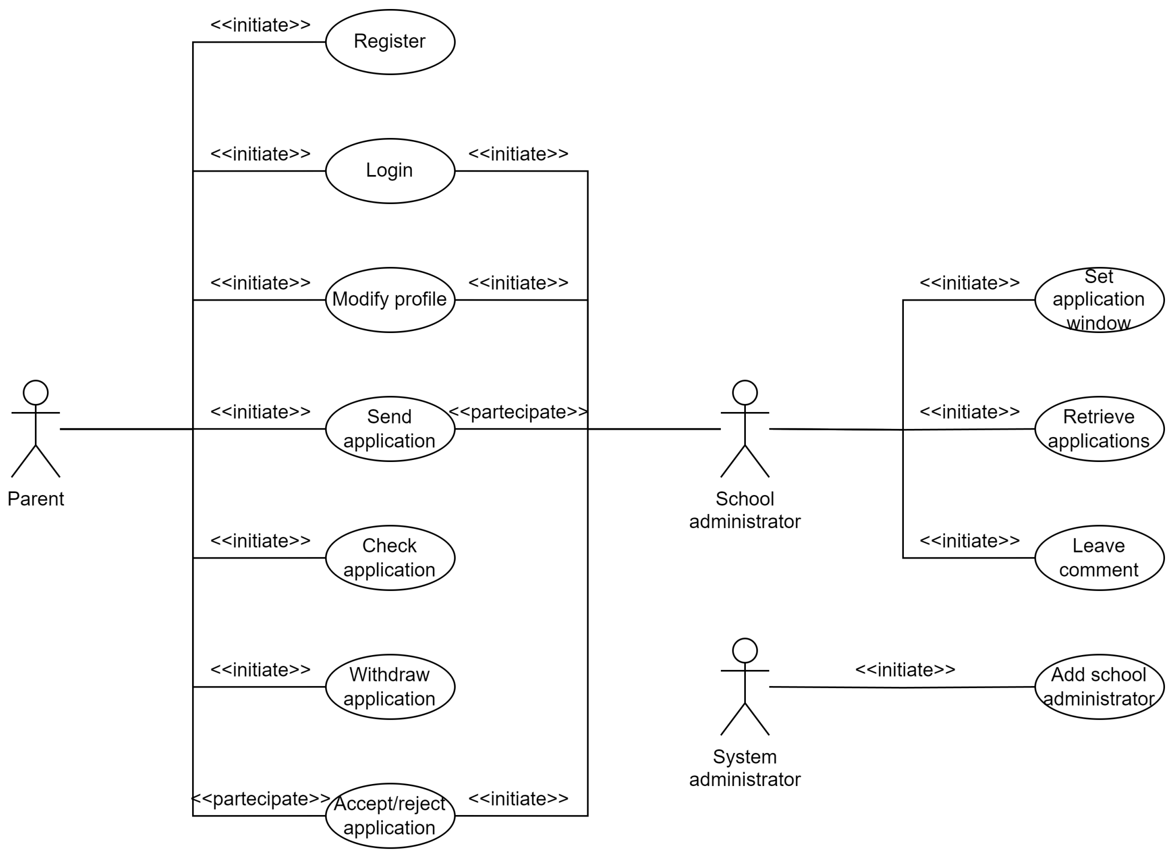
   - `AdmissionManager` allows system administrators to insert new administrators in the system and associate them with the corresponding school.

- `AdmissionManager` allows school administrators to log into the system using the credentials assigned to them by system administrators.
- `AdmissionManager` allows school administrators to indicate in their profile that they want to be notified when new applications for their schools are received.
- `AdmissionManager` allows school administrators to retrieve applications (related to their schools) that have yet to be evaluated.
- `AdmissionManager` allows school administrators to select an application yet to be evaluated and leave a comment in it.
- `AdmissionManager` allows school administrators to accept/reject an application.

The assumption is the following:

- School administrators periodically evaluate applications and guarantee to explicitly accept/reject all applications arrived within the notification window.

3. The UML use case diagram is the following:



4. We select send application case. We have that:

Actor(s)	Parent, School Administrator
Entry Condition	Parent has registered child and is logged in with the corresponding account. He/she has all necessary information
Event Flow	1. Parent selects school to which application must be sent 2. If required by the school, parent fills out additional information concerning child 3. Parent clicks submit button 4. System checks application and responds with application number 5. If administrator of selected school has asked to receive a notification of the application, email is sent to school administrator
Exit Condition	Application is received by the system, and email is sent to school administrator if he/she asked to be notified
Exceptions	Data provided in application is invalid or missing, user is notified that it should be fixed. School does no longer accept application (the application window has expired), so the application is immediately rejected.
Notes	In case of exception the system will notify user with a human-readable message

1.2 PaasPopCoin

The private security and event organization company HSG from the Netherlands wants to build an application (`PaasPopCoin`) that handles the coin emission and transactions in the scope of a medium-size music festival they are organizing. The goal of the system is to allow festival-goers and operators to spend an allotted amount of money in relative safety and without the need to bring wallets and other assets around the event. The software in question needs to handle at least three scenarios:

- Emission of coins in exchange for money through appropriate cashier desks and ATMs.
- Cash-back, that is, exchange of coins with cash in the same locations (we assume that people at the festival may be willing to receive back the money corresponding to the coins they have not used).
- Tracking of coin expenditure transactions at the various festival shops.

In the scope of the above scenarios, there are several special conditions to be considered. First, in the scope of coin emissions, there exist four classes of coin buyers:

a. VIPs who receive a 30% discount on the coins they buy.

b. Event organization people who receive a 50% discount.

c. Event ticket holders class A, who receive a 20% discount.

d. Regular ticket holders who receive no discount.

When buying coins, users first need to authenticate themselves by inserting their own ID card in the ATM or by giving it to the cashier; this allows the system to determine the class to which each coin buyer belongs. After authentication, buyers get the coins upon inserting into the ATM or giving to the cashier the corresponding amount of money. Second, also in the context of cash-back, users need to authenticate with their ID card to make sure the appropriate amount of money is given back, considering their role and privileges. Third, during the event, every shop clerk keeps track through the `PaasPopCoin` system of the sales of products and the coins received. `PaasPopCoin` relies on a third-party analytics service to periodically check whether the festival is earning money or not (cost-benefit analysis). Such check is performed with respect to costs of products being sold during the event, as well as the overhead to cover all event organization and management expenses.

1. With reference to the Jackson-Zave distinction between the world and the machine, identify the relevant world phenomena for `PaasPopCoin`, including the ones shared with the machine, providing a short description if necessary. For shared phenomena specify whether they are controlled by the world or the machine. Focus on phenomena that are relevant to describe the requirements of the system.

2. Describe through a UML Class Diagram the main elements of the `PaasPopCoin` domain.

3. Define a UML Use Case Diagram describing the relevant actors and use cases for `PaasPopCoin`. You can provide a brief explanation of the Use Case Diagram, especially if the names of the use cases are not self-explanatory.

Solution

1. The world-only phenomena can be:

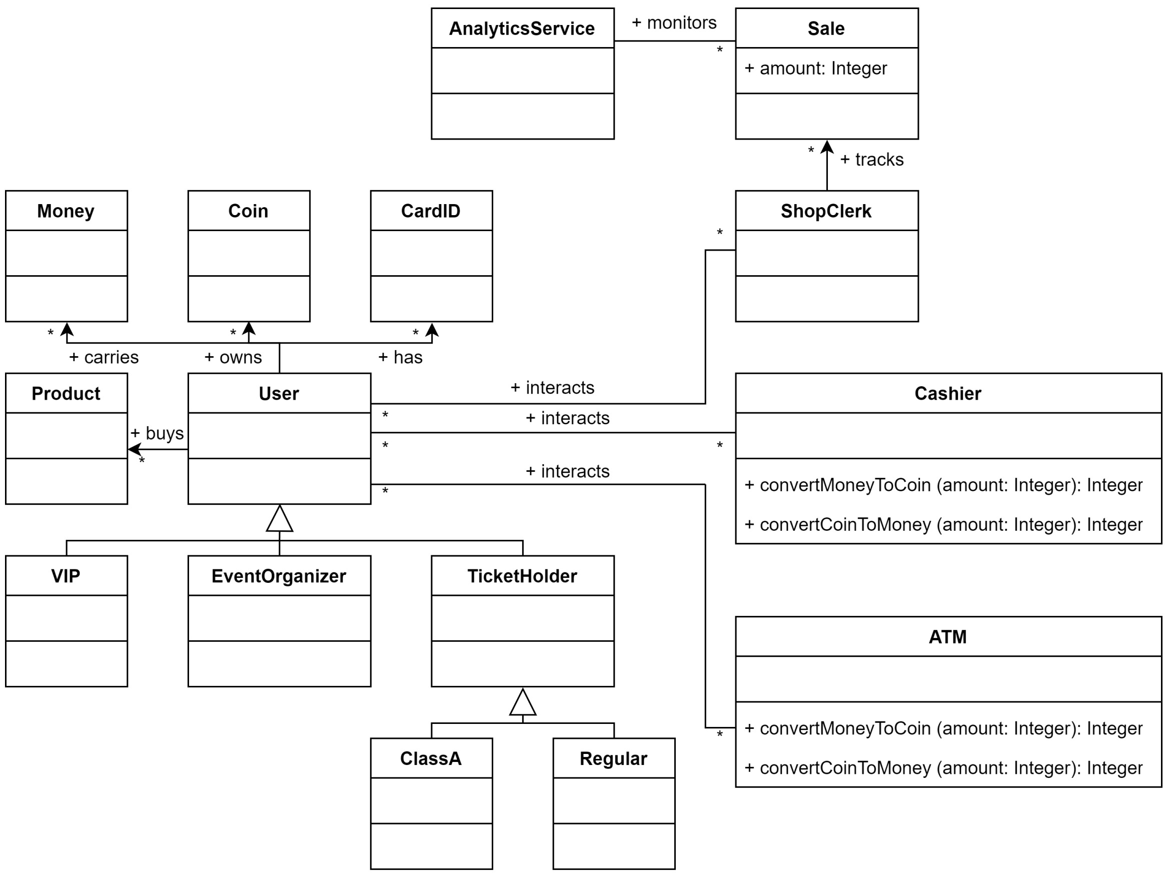
   - User buys Class A ticket.
   - User buys regular ticket.
   - VIP is contracted for event.
   - Event organization is started and contractors registered.
   - Event starts.
   - User gives money to cashier (to be converted in coins).
   - User gives coins to cashier (to be converted in money).
   - User gives ID card to cashier.
   - User buys some product at festival.
   - The external analytics service checks the success of an event.

   The shared phenomena can be the following:

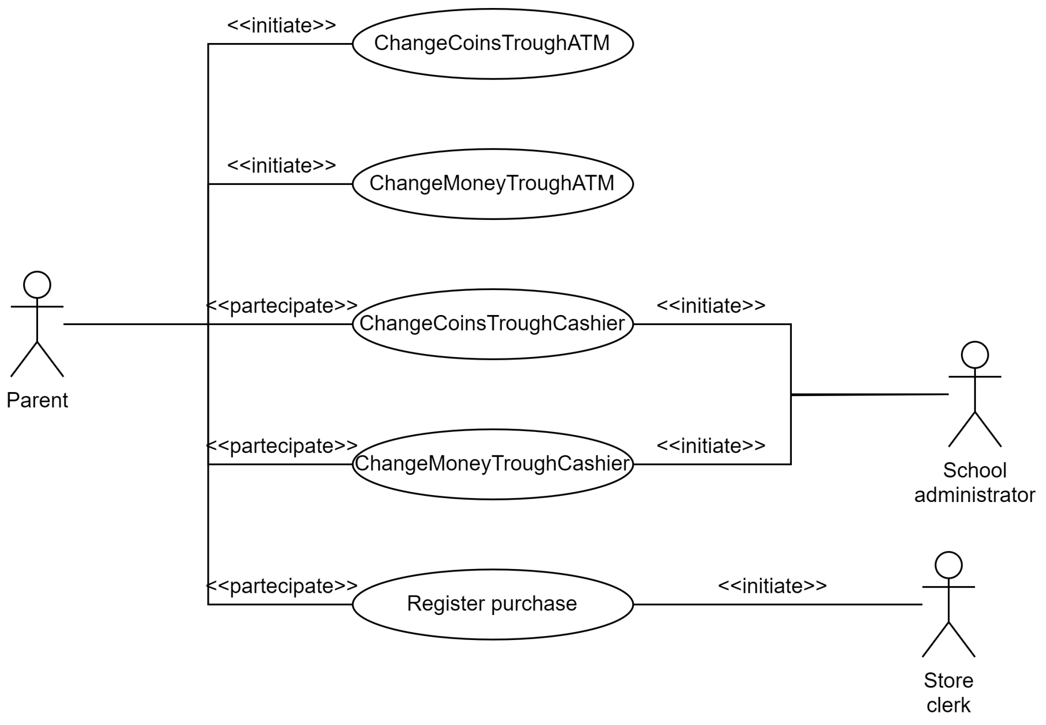
   - User inserts money into an ATM.

- ID Card is inserted into ATM.
- User inserts coins into an ATM.
- Cashier inserts in the system an ID card number.
- Cashier inserts in the system the amount of money handed by a certain user.
- Cashier inserts in the system the amount of coins returned by a certain user.
- Store clerk inputs in system the amount of coins spent by user in shop.
- The system enables coin emission after checking ID card and inserted amount of money.
- The system enables cash-back after checking ID card and inserted number of coins.
- The system sends data about purchases to the external analytics service.

2. The UML diagram of the given problem is:



3. The UML use case diagram of the given problem is:



<<initiate>>
ChangeCoinsTroughATM
<<initiate>>
ChangeMoneyTroughATM
<<partecipate>>
ChangeCoinsTroughCashier
<<initiate>>
Parent
<<partecipate>>
ChangeMoneyTroughCashier
<<initiate>>
School administrator
<<partecipate>>
Register purchase
<<initiate>>
Store clerk

CHAPTER 2

Exercise session II

2.1 Alloy

Consider construction cubes of three different sizes, small, medium, and large. You can build towers by piling up these cubes one on top of the other respecting the following rules:

- A large cube can be piled only on top of another large cube.
- A medium cube can be piled on top of a large or a medium cube.
- A small cube can be piled on top of any other cube.
- It is not possible to have two cubes, A and B, simultaneously positioned right on top of the same other cube C.

1. Model in Alloy the concept of cube and the piling constraints defined above.
2. Model also the predicate canPileUp that, given two cubes, is true if the first can be piled on top of the second and false otherwise.
3. Consider now the possibility of finishing towers with a top component having a shape that prevents further piling, for instance, a pyramidal or semispherical shape. This top component can only be the last one of a tower, in other words, it cannot have any other component piled on it. Rework your model to include also this component. You do not need to consider a specific shape for it, but only its property of not allowing further piling on its top. Modify also the canPileUp predicate so that it can work both with cubes and top components.

Solution

1. The concept of cube and its constraints can be defined in the following way.

```
abstract sig Size{}

sig Large extends Size{}

sig Medium extends Size{}

sig Small extends Size{}
sig Cube {
size: Size,
piledOn: lone Cube
}{piledOn ≠ this}

fact noCircularPiling {
no c: Cube | c in c.^piledOn
}

fact pilingUpRules {
all c1, c2: Cube | c1.piledOn = c2 implies (
c2.size = Large or
c2.size = Medium and (c1.size = Medium or c1.size = Small) or c2.size = Small and c1.size
    ↪  = Small)
}

fact noMultipleCubesOnTheSameCube {
no disj c1, c2: Cube | c1.piledOn = c2.piledOn
}
```

2. The predicate canPileUp can be defined as follows.

```
pred canPileUp[cUp: Cube, cDown: Cube] {
cUp.piledOn = cDown and
(cDown.size = Large or
cDown.size = Medium and (cUp.size = Medium or cUp.size = Small) or 2
cDown.size = Small and cUp.size = Small)
}

pred show {}

run show

run canPileUp
```

3. The now model become:

```
abstract sig Size{}

sig Large extends Size{}

sig Medium extends Size{}

sig Small extends Size{}

abstract sig Block{
piledOn: lone Cube
}

sig Cube extends Block {
size: Size
}{piledOn ≠ this}

sig Top extends Block { }

fact noCircularPiling {
no c: Cube | c in c.^piledOn
}

fact noMultipleBlocksOnTheSameCube {
no disj b1, b2: Block | b1.piledOn = b2.piledOn
}

fact pilingUpRules {
all c1, c2: Cube | c1.piledOn = c2 implies (
c2.size = Large or
c2.size = Medium and (c1.size = Medium or c1.size = Small) or c2.size = Small and c1.size
    ↪  = Small)
}

pred canPileUp[bUp: Block, cDown: Cube] {
```

```
bUp.piledOn = cDown and (bUp in Top or
(cDown.size = Large or
cDown.size = Medium and (bUp.size = Medium or bUp.size = Small) or cDown.size = Small and
    ↪  bUp.size = Small))
}

pred show {}

run show

run canPileUp
```

2.2 Alloy

The company `TravelSpaces` decides to help tourists visiting a city in finding places that can keep their luggage for some time. The company establishes agreements with small shops in various areas of the city and acts as a mediator between these shops and the tourists that need to leave their luggage in a safe place. To this end, the company wants to build a system, called `LuggageKeeper`, that offers tourists the possibility to: look for luggage keepers in a certain area; reserve a place for the luggage in the selected place; pay for the service when they are at the luggage keeper; and, optionally, rate the luggage keeper at the end of the service.

Given the scenario above, consider the following world phenomena:

- Every user owns various pieces of luggage.
- Every user can carry around various pieces of luggage.
- Each piece of luggage can be either safe, or unsafe.
- Small shops store the luggage in lockers, where each locker can store at most one piece of luggage.

Consider also the following shared phenomena:

- Each locker is opened with an electronic key that is associated with it (the electronic key is regenerated at each use of the locker; also, a locker that does not have an electronic key associated with it is free).
- Each user can hold various electronic keys.

Formalize through an Alloy model:

1. The world and machine phenomena identified above.
2. A predicate capturing the domain assumption D1 that a piece of luggage is safe if, and only if, it is with its owner, or it is stored in a locker that has an associated key, and the owner of the piece of luggage holds the key of the locker.
3. A predicate capturing the requirement R1 that a key opens only one locker.
4. A predicate capturing the goal G1 that for each user all his/her luggage is safe.
5. A predicate capturing the operation GenKey that, given a locker that is free, associates with it a new electronic key.

Solution

The model requested is:

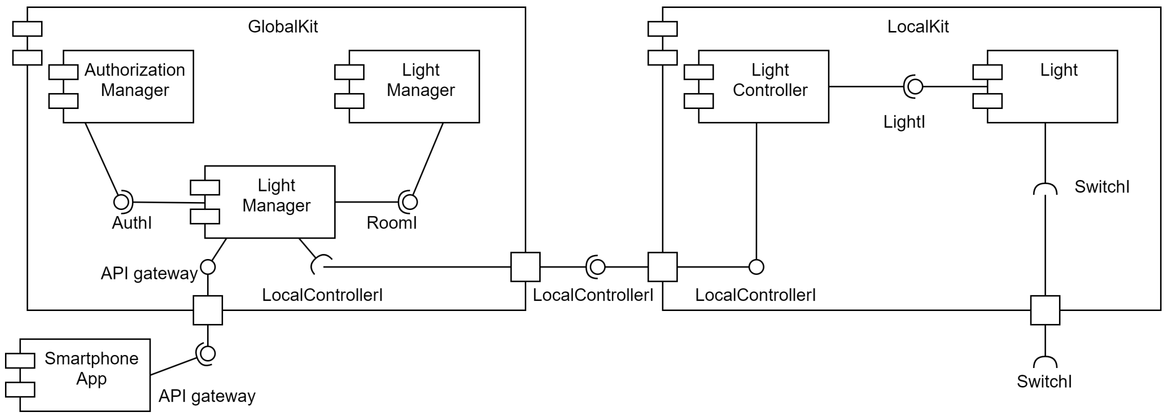
```
abstract sig Status{}

one sig Safe extends Status{}

one sig Unsafe extends Status{}

sig Luggage{
luggageStatus : one Status
}

sig EKey{}

sig User{
owns : set Luggage, carries : set Luggage, hasKeys : set EKey
}

sig Locker{
hasKey : lone EKey, storesLuggage : lone Luggage
}

sig Shop{
lockers : some Locker
}

pred D1 {
all lg : Luggage | lg.luggageStatus in Safe
iff
all u : User | lg in u.owns implies ( lg in u.carries or some lk : Locker | lg in lk.
    ↪ storesLuggage and lk.hasKey ≠ none and lk.hasKey in u.hasKeys )
}

pred R1 {
all ek : EKey | no disj lk1, lk2: Locker | ek in lk1.hasKey and ek in lk2.hasKey
}

pred G1 {
all u : User | all lg : Luggage | lg in u.owns implies lg.luggageStatus in Safe
}

pred GenKey[lk, lk' : Locker] {
lk.hasKey = none
lk'.storesLuggage = lk.storesLuggage
one ek : EKey | lk'.hasKey = ek
}
```

CHAPTER 3

Exercise session III

3.1 SmartLightingKit

SmartLightingKit is a system expected to manage the lights of a potentially big building composed of many rooms (e.g., an office space). Each room of the building, may have one or more lights. The system shall allow the lights to be controlled (either locally or remotely) by authorized users. Local control is achieved through terminals installed in the rooms (one terminal per room). Remote control is realized through a smartphone application, or through a central terminal installed in the control room of the building. While controlling the lights of a room, the user can execute one or more of the following actions: turn a light on/off at the current time, at a specified time, or when certain events happen (e.g., a person enters the building or a specific room). Moreover, users can create routines, that is, scripts containing a set of actions. SmartLightingKit manages remote control by adopting a fine-grained authorization mechanism. This means there are multiple levels of permissions that may even change dynamically. System administrators can control the lights of every room, install new lights, and remove existing ones. This diagram describes the portion of the SmartLightingKit system supporting lights turning on/off and lights status checking.



The diagram is complemented by the following description. `GlobalKit` is the component installed in the central terminal. It can be contacted through the `APIGateway` interface by the `SmartphoneApp` component representing the application used for remote control. `GlobalKit` includes:

- `RoomMapping`, which keeps track, in a persistent way, of lights' locations within the building's rooms.
- `AuthorizationManager`, which keeps track, in a persistent way, of the authorizations associated with each user (for simplicity, we assume that users exploiting the operations offered by the `APIGateway` are already authenticated through an external system and include in their operation calls a proper token that identifies them univocally);
- `LightManager`, which coordinates the interaction with all `LocalKit` components.

Each `LocalKit` component runs on top of a local terminal. Also, each `LocalKit` exposes the `LocalControllerI` interface that is implemented by its internal component `LightController`, which controls the lights in the room. Each light is represented in the system by a Light component which interacts with the external system operating the light through the `switchCommand` operation offered by the latter.

1. Analyze the operations offered by the components shown in the diagram and identify proper input and output parameters for each of them.
2. Write a UML Sequence Diagram illustrating the interaction between the software components when a regular user wants to use the smartphone application to check whether he/she left some lights on (among the lights he/she can control).
3. Assume that, at a certain point, the following new requirement is defined: "The smartphone application should allow users to activate and deactivate the receival of real-time updates about the state (on/off) of all the lights they can control." Define a high-level UML Sequence Diagram to describe how the current architecture could accommodate this requirement. Highlight the main disadvantage emerging from the sequence diagram.

Solution

1. The `APIGateway` has the following methods:
   - `getLights`. The input of this method is the `userID`. The output of this method is `list[lightID]`.
   - `getState`. The input of this method is the `userID`. The output of this method is `list[(lightID, state)]`.
   - `setState`. The inputs of this method are the `userID`, `lightID`, and `state`. The output of this method is `none`.

   The `AuthI` has the following method:

   - `getAuthorizations`. The input of this method is the `userID`. The output of this method is `list[(roomID, localKitID)]`.

   The `RoomI` has the following method:

   - `findLight`. The input of this method is the `roomID`. The output of this method is `list[lightID]`.

   The `LocalControllerI` has the following methods:

- `switch`. The input of this method is the `lightID`. The output of this method is `none`.
- `getState`. The input of this method is the `lightID`. The output of this method is `state`.

The `LightI` has the following methods:

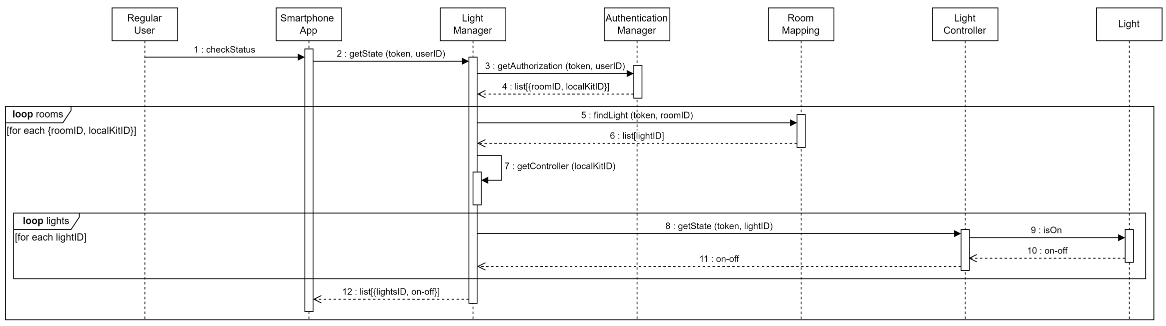
- `isOn`. The input of this method is the `none`. The output of this method is `true` or `false`.
- `switch`. The input of this method is the `none`. The output of this method is `none`.

The `SwitchI` has the following methods:

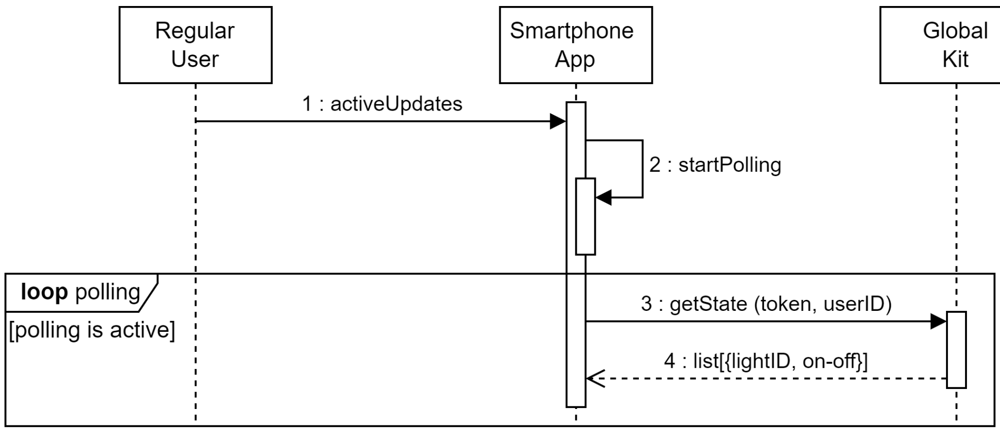
- `switchCommand`. The input of this method is the `none`. The output of this method is `none`.

Note that the state can be `on` or `off`. Additionally, all the operations of `APIGateway`, `AuthI`, `RoomI`, `LocalKitI` receive as input also the authentication token.

2. The requested UML Sequence Diagram is depicted below:


Regular User
Smartphone App
Light Manager
Authentication Manager
Room Mapping
Light Controller
Light
1 : checkStatus
2 : getState (token, userID)
3 : getAuthorization (token, userID)
4 : list[{roomID, localKitID}]
loop rooms
[for each {roomID, localKitID}]
5 : findLight (token, roomID)
6 : list[lightID]
7 : getController (localKitID)
loop lights
[for each lightID]
8 : getState (token, lightID)
9 : isOn
10 : on-off
11 : on-off
12 : list[{lightsID, on-off}]

3. The requested UML Sequence Diagram is depicted below:



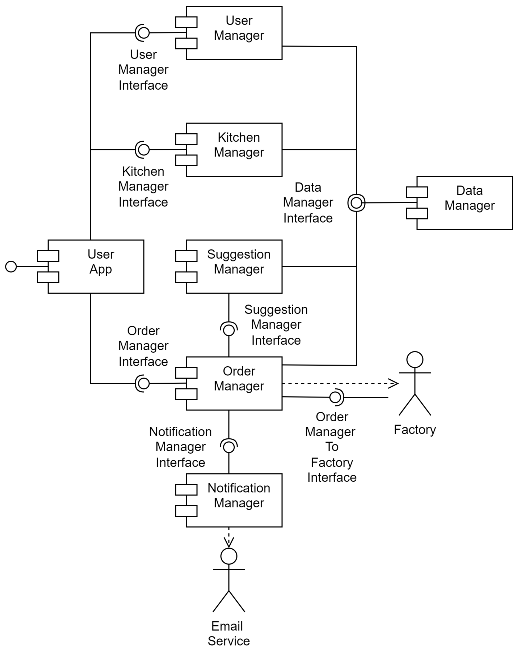
The problems of this new feature are:

- The current architecture does not support updates in push mode.
- The *SmartphoneApp* carries out a continuous polling process to retrieve the status of all the lights even in case it does not change.
- This propagates also internally to *GlobalKit* and the involve *LocalKits*, thus resulting in a potentially significant communication overhead.

3.2 KitchenDesigner

We want to build an application, KitchenDesigner, that allows users to define the layout of kitchens and to insert in such layouts the furniture and appliances (refrigerators, stoves, dishwashers, etc.) that go in them. For simplicity, we consider only rooms that have rectangular shape. Users can define the physical features of the room (length, width, height). Moreover, they can add pieces of furniture and appliances, and move them around. The position of each piece of furniture/appliance is given by the 3D coordinates of the lower left corner of the bottom side of the item, and by its orientation (which is the angle with respect to the x axis, and which can only be a multiple of 90 degrees). Users register with the application to be able to store and retrieve their designs. After a user finalizes his/her kitchen design, he/she can ask to have the kitchen delivered to a desired address; in this case, the kitchen is sent to production, and the user is given a probable date of delivery (producing a kitchen can take a few days, or even weeks). For simplicity, we do not consider payment. When the kitchen is ready to be delivered, the user is notified of the confirmed date of delivery. The system keeps track of the designs created by users, to identify the most common combinations of pieces of furniture and appliances. Hence, upon request by a user, given a draft layout for the kitchen, the system returns a list of possible pieces of furniture and appliances that might be added to that kitchen. Assuming you need to implement system KitchenDesigner analyzed above, identify the most relevant components and interfaces describing them through UML Component or Class Diagrams. Provide a brief description of each component. For each interface identified in the previous point, list the operations that it provides. You do not need to precisely specify operation parameters; however, you should give each operation a meaningful enough name to understand what it does; you can also briefly describe what information operations use/produce. Define a runtime-level Sequence Diagram describing the interaction that occurs among the KitchenDesigner components when the user asks for a list of suggested elements to be added to the kitchen. If useful, provide a brief description of the defined Sequence Diagram.

Solution The requested UML Component diagram is as follows:



The component `UserApp` is the front-end for users. It allows them to interact with the system by offering the following functions through interface *UserAppI*:

- Register, which has as input the user data.
- Login, which has as inputs the userId and the password.
- Create a new kitchen project, which has as input the name.
- Set the dimensions of the kitchen, which has as input the dimensions.
- Add an item to kitchen, which has as input the item to be added.
- Move item in kitchen, which has as input the item to be moved and the new position.
- Remove item from kitchen, which has as input the item to be removed.
- Ask for a suggestion, which returns the suggested elements.
- Finalize kitchen.
- Place order.

The module can keep track of the current kitchen being designed, so the user operates on the open project, and the information does not need to be included in the calls each time.

The component `UserManager` offers, through interface `UserManagerI`, the basic functions for handling users:

- Register, which has as input the user info.
- Login, which has as inputs the userId and password.

The component `KitchenManager` provides the interface `KitchenManagerI`, which handles functions related to the management of kitchens (excluding placing the order, which is handled by another component):

- Create a new kitchen project, which has as input the name.
- Set the dimensions of the kitchen, which has as inputs the kitchenId and the dimensions.
- Add an item to kitchen, which has as inputs the kitchenId and the item to be added.
- Move item in kitchen, which has as input the kitchenId, the item to be moved and the new position.
- Remove item from kitchen, which has as inputs the kitchenId and the item to be removed.
- Ask for a suggestion, which has as input the kitchenId and returns the suggested elements.
- Finalize kitchen, which has as input the kitchenId.

These operations are similar to those offered by the UserApp, but they also include the Id of the kitchen which should be modified.

The component `SuggestionManager` provides, through interface `SuggestionsManagerI`, functions related to the retrieval of suggestions:

- Get suggestion, which has as input the kitchenId.

The idea is that the component periodically retrieves kitchen designs from the `DataManager`, mines them, and identifies which combinations of items are most common. Hence, it only provides a single function, for getting the outcome of this mining. Hence, the computation is mostly offline, the get suggestion function compares what is present in the kitchen with what is most common, and suggests additional elements.

The component `OrderManager` provides, through interfaces `OrderManagerI` and `OrderManager2Factor` functions related to the management of orders. These are used by two different clients. The interface `OrderManagerI` is used by `UserApp` and provides the following function:

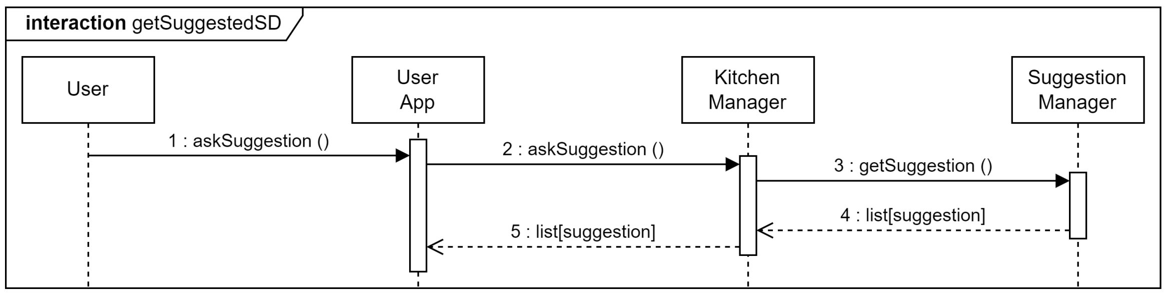
- Place order, which has as input the kitchenId. It is used to start the process to produce a kitchen. To handle the order the OrderManager needs to notify the factory of the new order. The handling of the order, and in particular its creation, is outside the scope of the `KitchenDesigner` application; this is represented in the diagram by the fact that there is an interaction with the factory. When the order is complete, the `OrderManager` is informed of this through interface `OrderManager2FactoryI` (to be used by the external actor factory) which provides the following operation:
  - Kitchen completed, which has as input the kitchenId. Which simply informs the OrderManager that the kitchen is indeed ready (hence the user can be notified of the date of its actual delivery).

The component `NotificationManager` handles the notification to users, in particular updates on when the kitchen will be delivered. It provides the following function through interface `NotificationManagerI`:

- Notify user, which has as inputs the message to be sent and the recipient. The notification is sent as an email, and for this reason it goes through an email server, which is an external component, outside the system.

The component `DataManager` handles the data of the system, which consists essentially of users and kitchens. It provides interface `DataManagerI`, which includes all necessary functions to handle CRUD operations on data. The only backend component that does not need to interact with `DataManager` is `NotificationManager`, because it relies on information provided by `OrderManager`.

The requested UML Sequence diagram is depicted below:



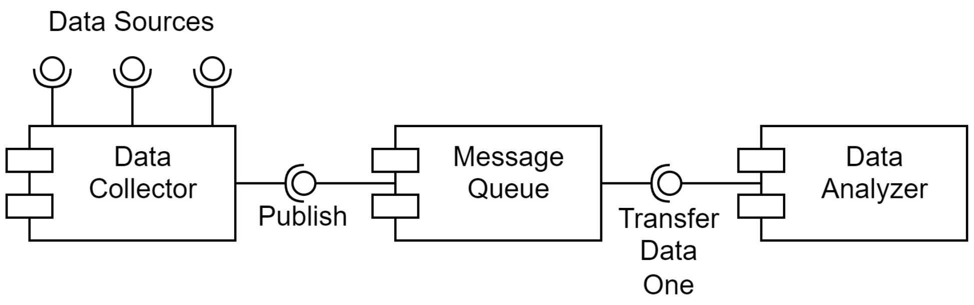
Mining of the designs is done asynchronously, not when a suggestion is requested, but offline, so it is not represented in this interaction.

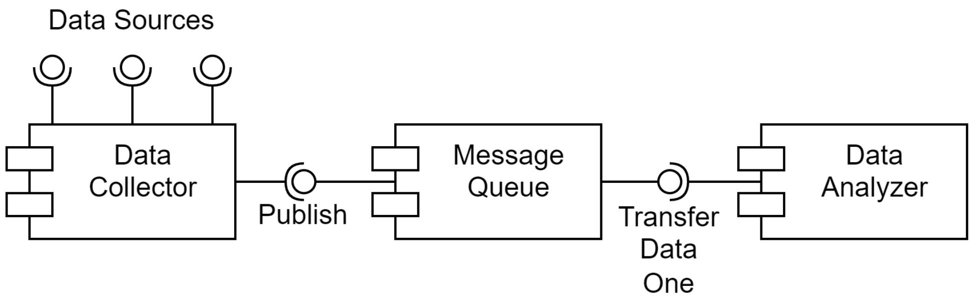
CHAPTER 4

Exercise session IV

4.1 Data analysis architecture

Consider the following two versions of the same system:





1. What is the difference between the two?

2. Define two sequence diagrams that describe how data flow through the system in the two versions of the architecture.

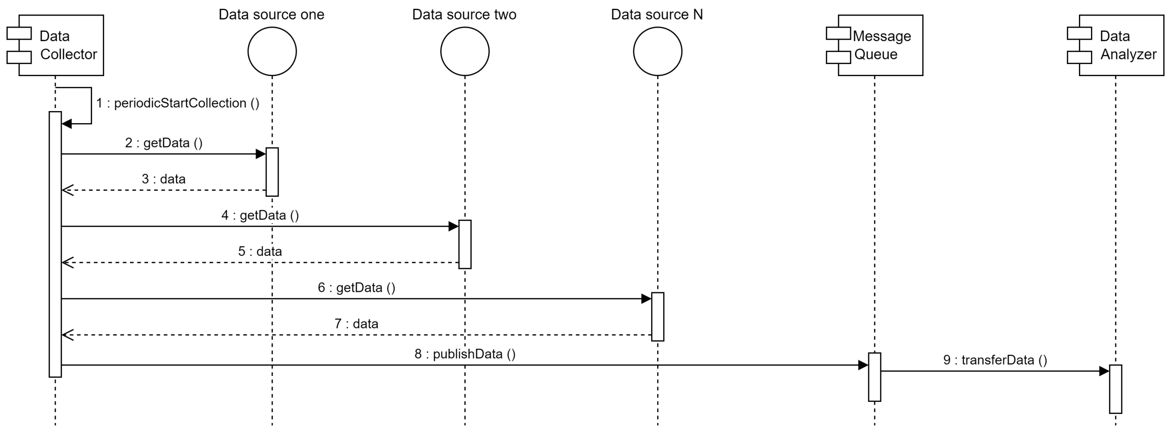
3. Assume that the components of your system offer the following availability: `DataCollector` (0.99), `MessageQueue (0.9999)`, and `DataAnalyzer` (0.995). Provide an estimation of the total availability of your system (you can provide a raw estimation of the availability without computing it completely).

4. Assuming that you wanted to improve this total availability by exploiting replication, which component(s) would you replicate?

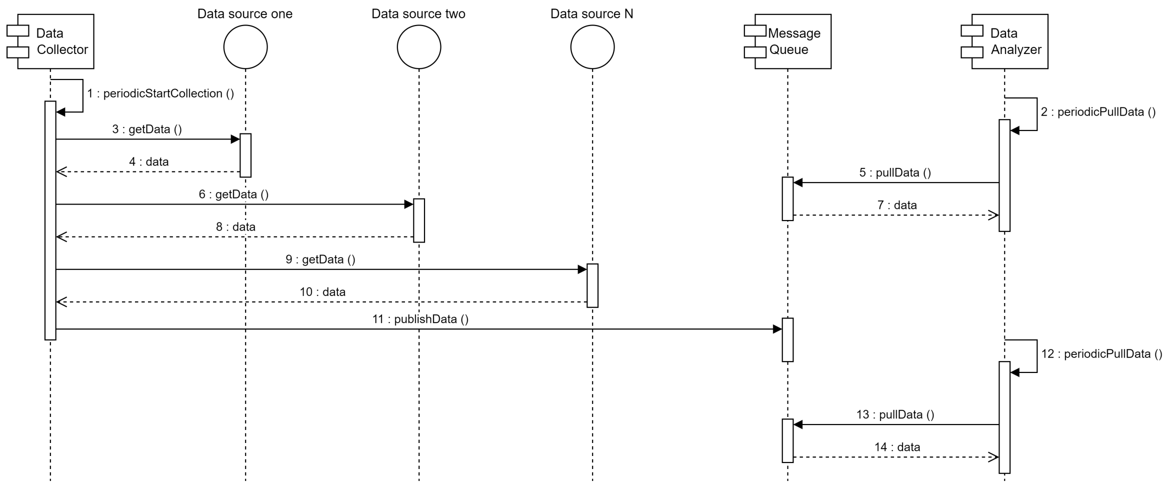
Solution

1. In the second case, the `MessageQueue` does not actively push the data to the `DataAnalyzer`, but it offers interface TransferData2 so that the `DataAnalyzer` can pull data as soon as it is ready to process them. Also in this case, both a batch or a per data approach is possible. The rest of the system behaves as first one.

2. The sequence diagram compatible with the first component diagram is as follows:



And the sequence diagram compatible with the second component diagram is as follows:



3. Data flow through the whole chain of components to be processed. Thus, we have a series of component. In this scenario, the total availability of the system is determined by the weakest element, that is, the `DataCollector`:

$$A_{\text{Total}} = 0.99 \cdot 0.9999 \cdot 0.995 = 0.985$$

4. If we parallelize `DataCollector` by adding a new replica, we can achieve the following availability:

$$(1 - (1 - 0.99) \cdot 2) \cdot 0.9999 \cdot 0.995 = 0.995$$

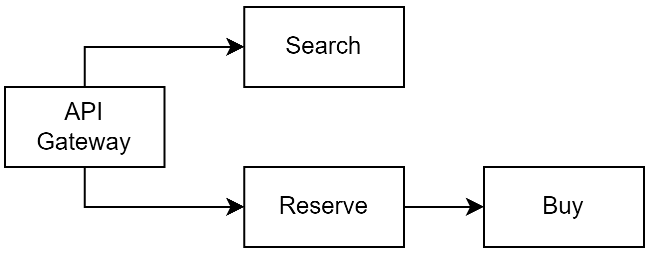
If we increase the number of `DataCollector` replica, we do not achieve an improvement as the weakest component becomes the `DataAnalyzer`. We can parallelize this component as well to further improve the availability of our system.

4.2 TrainTicket

Consider a microservices application called `TrainTicket` composed of 3 domain microservices (search, reserve, buy) and 1 additional microservice that acts as the API gateway. TrainTicket supports two basic operations invoked using the exposed RESTFul APIs:

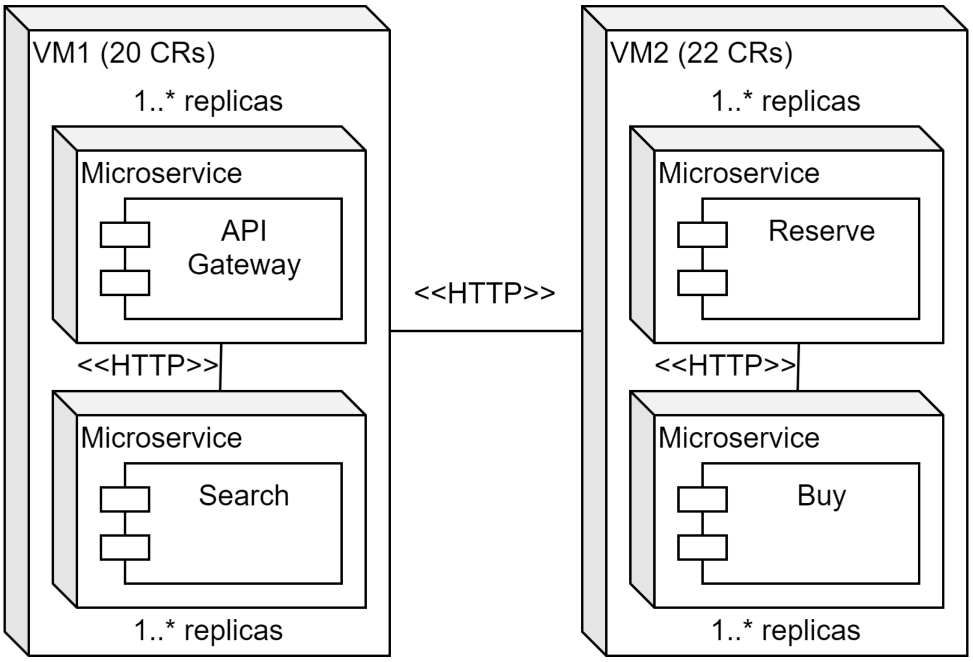
1. Search: /APIv1/search/args.
2. Reserve: /APIv1/reserve/args.

Requests (both search and reserve) are received and then dispatched by the API gateway. In particular, the following high-level schema shows how requests propagate from the gateway to internal microservices. Note that in this example reserve includes also the purchase of reserved items.



Microservices run in units deployed onto 2 different Virtual Machines, VM1 and VM2 as shown in the following UML deployment diagram. The available VMs have Computational Resources (CRs) that can be allocated to run microservices. Each VM has a maximum number of CRs and each microservice requires a certain number of CRs, according to the executed artifact. As shown in the schema, available CRs are as follows:

- VM1: 20 CRs.
- VM2: 22 CRs.



The mapping between microservices and required CRs is as follows:

- API gateway: 2 CRs.
- Search: 5 CRs.
- Reserve: 4 CRs.
- Buy: 5 CRs.

The deployment diagram shows that each microservice can be replicated to have redundant business-critical components. In the latter case, requests are directed to all the replicas rather than to an individual instance and the first answer received from a replica is returned to the caller, while the others are simply ignored. The number of replicas for each microservice shall be defined so that the following nonfunctional requirement is satisfied, and the deployment constraints defined in the deployment diagram and above are fulfilled. We have also the following requirement: both search and reserve services exposed through API gateway shall have availability greater than or equal to 0.99.

1. Considering the constraints of the execution environment represented above, determine whether requirement R1 can be satisfied or not assuming the following availability estimates for each microservice: API gateway (0.99), search (0.98), reserve (0.95), and buy (0.91).
2. Consider the problem of resource allocation taking into account the operational profile, that is, the behavior of the users. Assume the following workload in terms of average number of concurrent users for each request:
   - Search: 50 users.
   - Reserve: 90 users.

   Assume also that for reserve, only 20% of users complete the purchase at reservation time. This means that 20% of reserve requests get through and reach the buy microservice, while 80% of them terminate the execution without calling buy.

Solution

1. Considering the execution environment, we can derive the following equations constraining the number of replicas:
   - $2x + 5y \leq 20$.
   - $4u + 5z \leq 22$.
   - $(1 - (1 - 0.99)x) \cdot (1 - (1 - 0.98)y) \geq 0.99$.
   - $(1 - (1 - 0.99)x) \cdot (1 - (1 - 0.95)u) \cdot (1 - (1 - 0.91)z) \geq 0.99$.

   Where variables x, y, u, and z represent the number of replicas for the microservices API gateway, search, reserve, and buy, respectively. The requirement R1 can be satisfied since there exists a valid assignment to variables that satisfies all constraints. For instance: $x = 2$, $y = 2$, $u = 3$, and $z = 2$.

2. After a preliminary analysis, we realize that availability depends on the workload according to the following new estimates:

	Low workload	**High workload**
Microservice	*Availability*	
API gateway	0.99	0.98
Search	0.98	0.95
Reserve	0.95	0.93
Buy	0.91	0.90

The expected workload for each microservice is as follows:

- API gateway: 140 users (high).
- Search: 50 users (low).
- Reserve: 90 users (high).
- Buy: 18 users (low).

The constraints extracted from requirement R1 become as follows:

$$(1-(1-0.98)x)\cdot(1-(1-0.98)y)\geq 0.99$$

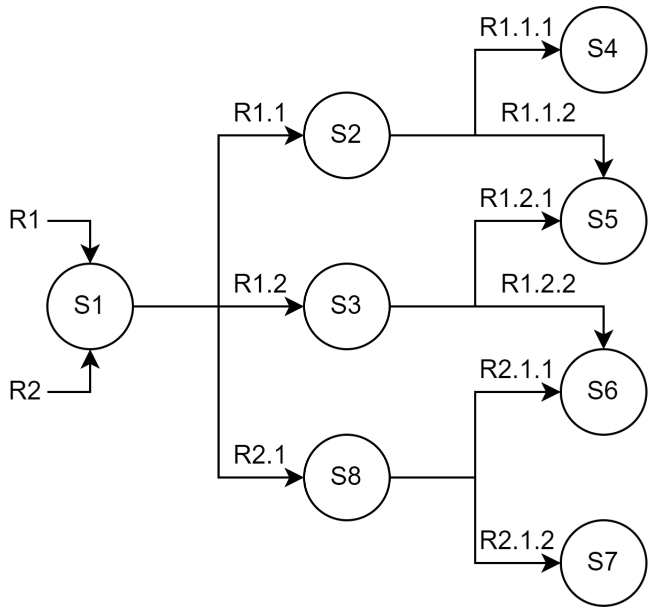
$$(1-(1-0.98)x)\cdot(1-(1-0.93)u)\cdot(1-(1-0.91)z)\geq 0.99$$

An optimal resource allocation is represented by the assignment $x=2$, $y=2$, $u=3$, and $z=2$ that is again feasible according to environment constraints.

4.3 Microservices and availability

Consider the microservice-based architecture shown in the figure below. The architecture is organized in eight stateless microservices that collaborate to fulfill requests R1 and R2. S1 is the front-end service that receives both requests. The fulfillment of request R1 requires the interaction with services S2 and S3 (through sub-requests R1.1 and R1.2, respectively), which, in turn, need to interact with other services. In particular, S2 interacts with S4 and S5 and S3 with S5 and S6. The fulfillment of R2 requires that S1 interacts with S8 which, in turn, interacts with S6 and S7.

Service	**Availability**
S1	0.99
S2	0.9
S3	0.9
S4	0.95
S5	0.999
S6	0.99
S7	0.99
S8	0.95



1. Assuming that the availability of services S1-S8 is the one reported in the table above, what is the availability of the system when answering to request R1?
2. If each of the services S1-S8 is duplicated, what is the new value of the availability computed at point 1?

Solution

1. Note that, regardless of the way the interaction between the `MessageQueue` and the Data-Analyzer works, data have to flow through the whole chain of components to be processed. This implies that we can model the system as a series of component. The total availability of the system is determined by the weakest element, that is, the `DataCollector`.

$$A_{\text{Total}} = 0.99 \cdot 0.9999 \cdot 0.995 = 0.985$$

If we parallelize the data collector adding a new replica, we can achieve the following availability:

$$(1 - (1 - 0.99)2) \cdot 0.9999 \cdot 0.995 = 0.995$$

At this point, even if we increase the number of `DataCollector` replica, we do not achieve an improvement as the weakest component becomes the DataAnalyzer. We can parallelize this component as well to further improve the availability of our system.

2. Let's consider the impact of the `DataCollector` parallelization on the rest of the system. If both replicas acquire information from the same sources in order to guarantee that all data are offered to the rest of the system, then the other components will see all data duplicated and will have to be developed considering this situation. For instance, the `MessageQueue` could discard all duplicates. Another aspect to be considered is that both `DataSources` and `MessageQueue` have to implement mutual exclusion mechanisms that ensure the communication between them and the two `DataCollector` replicas does not raise concurrency issues. Another option could be that only one `DataCollector` replica at a time is available, and the other is activated only when needed (for instance, if the first one does not send feedback within a certain timeout).

CHAPTER 5

Exercise session V

5.1 Testing

Consider the function `foo`, written in a C-like language:

```
0: int foo(int a, int b) {
1:     a++;
2:     while (a < b) {
3:         if (a != b)
4:             a++;
5:     }
6:     return a;
7: }
```

1. Execute `foo` symbolically limiting the execution of the loop statement to exactly two iterations. Show, for each non-conditional statement:

   path condition, symbolic state

2. Define the precondition to the execution of `foo` such that the while loop is executed exactly twice.
3. Generate three possible test cases to run this path.

Solution

1. The path needed to limit the execution of the loop exactly two time is composed by the following lines:

   0: $a = A, b = B$

   1: $a = A + 1$

   2: $A + 1 < B$

   3: $A + 1 \neq B$

   4: $a = A + 2$

2: $A + 2 < B$

3: $A + 2 \neq B$

4: $a = A + 3$

2: $A + 3 \geq B$

6:

2. The condition on the variables are:

$$\begin{cases} A + 1 < B \\ A + 1 \neq B \\ A + 2 < B \\ A + 2 \neq B \\ A + 3 \geq B \end{cases}$$

   The final condition for this path is that $A + 3 = B$.

3. Three possible test cases are:

   - $\{a = 1, b = 4\}$
   - $\{a = 0, b = 3\}$
   - $\{a = -3, b = 0\}$

5.2 Testing

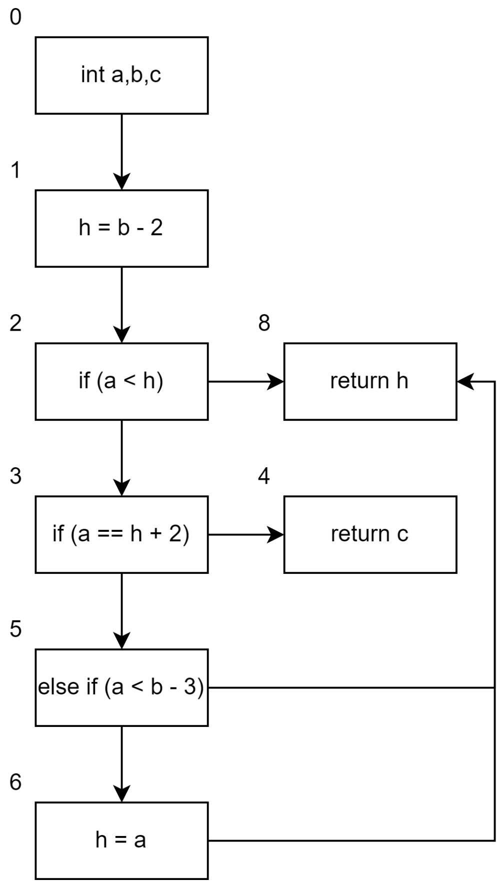
Consider the function `bar`, written in a C-like language:

```
0: int bar(int a, int b, int c) {
1:     int h = b-2;
2:     if (a < h) {
3:         if (a == h+2)
4:             return c;
5:         else if (a < b-3)
6:             h = a;
7:     }
8:     return h;
9: }
```

1. Derive the Control Flow Graph of the given function.
2. Derive the set of live variables at the exit of each block. Are there dead variables after definition at block 0?
3. Use symbolic execution to explore all paths in the function.

Solution

1. The Control Flow Graph of the function is derived from the code and is the following:



2. The definition of live variable is the following: "Given a CFG, a variable v is live at the exit of a block b if there is some path (on the CFG) from block b to a use of v that does not redefine v". By exploiting this definition we have to check all variables that are live in the node by checking if there is a path from the node that uses the variable without redefining it. With this analysis we find the following:

   - LV(0) $= \{a, b, c\}$
   - LV(1) $= \{a, b, c, h\}$
   - LV(2) $= \{a, b, c, h\}$
   - LV(3) $= \{a, b, c, h\}$
   - LV(4) $= \{\}$
   - LV(5) $= \{a, h\}$
   - LV(6) $= \{h\}$
   - LV(8) $= \{\}$

3. The possible paths are:

   - $\langle 0, 1, 2, 3, 5, 6, 8\rangle$: that is feasible with the condition $A < B - 3$.
   - $\langle 0, 1, 2, 3, 5, 8\rangle$: that is feasible with the condition $A = B - 3$.
   - $\langle 0, 1, 2, 8\rangle$: that is feasible with the condition $A \geq B - 2$.

5.3 Testing

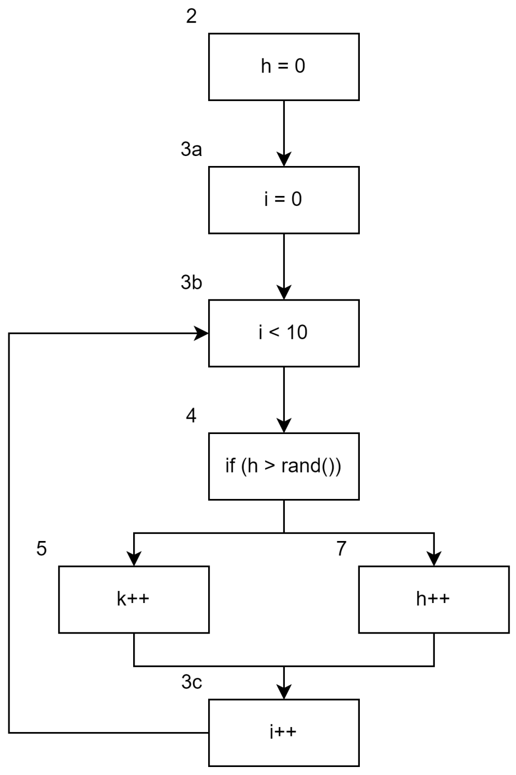
Consider the following function, written in a C-like language, where `rand()` returns a pseudo-random (integer) number:

```
0: void foo() {
1:     int h, k;
2:     h = 0;
3:     for (int i=0; i<10; i++) {
4:         if (h > rand())
5:             k++;
6:         else
7:             h++;
8:     }
9: }
```

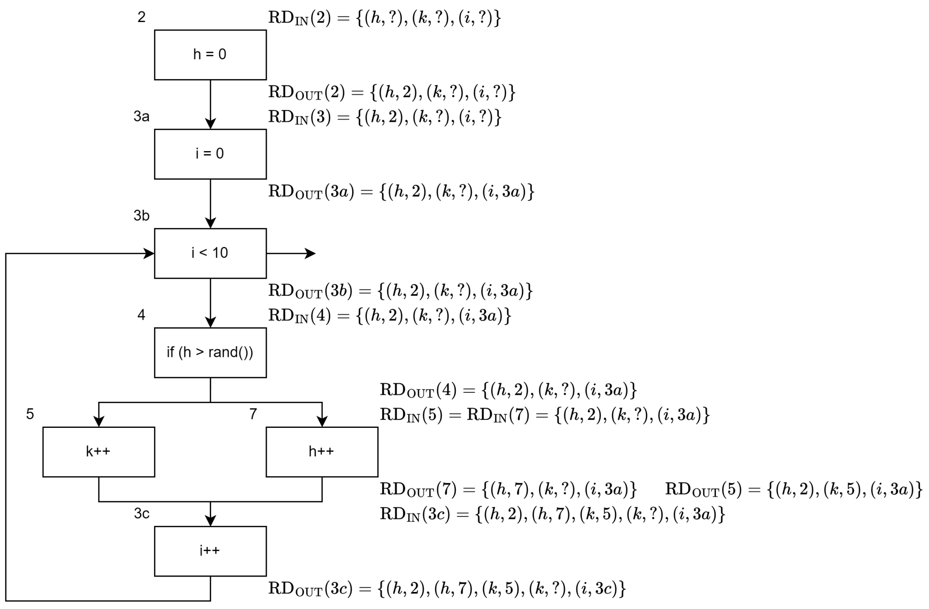
1. Build the Control Flow Graph of foo
2. Derive all the reaching definitions at the entry and the exit of each block.
3. According to the reaching definitions, derive all the UD chains and then def-use pairs for variables h and k.
4. According to the previous results, what are the potential problems of foo?

Solution

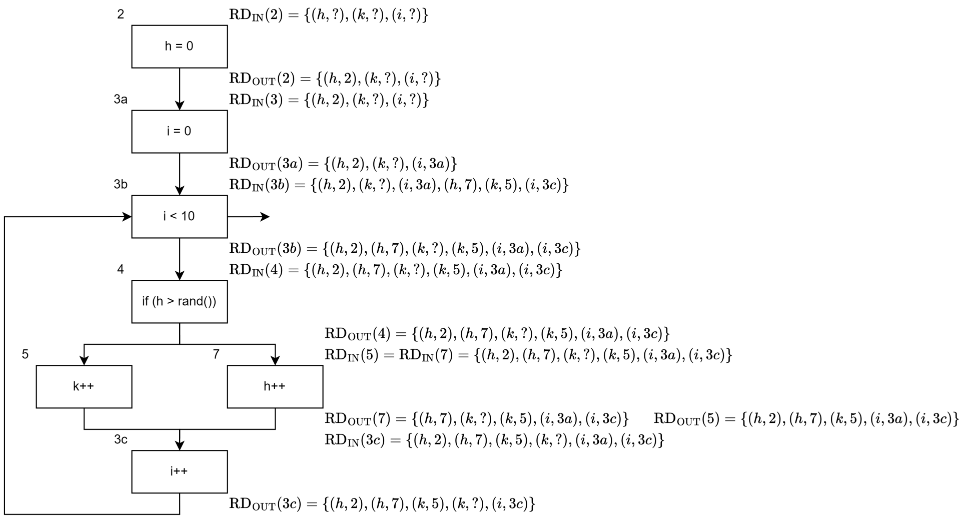
1. The Control Flow Graph of the function is derived from the code and is the following:



2. After executing once the loop we have the following reaching definitions:


2
$\mathrm{RD_{IN}}(2) = \{(h,?),(k,?),(i,?)\}$
h = 0
$\mathrm{RD_{OUT}}(2) = \{(h,2),(k,?),(i,?)\}$
3a
$\mathrm{RD_{IN}}(3) = \{(h,2),(k,?),(i,?)\}$
i = 0
$\mathrm{RD_{OUT}}(3a) = \{(h,2),(k,?),(i,3a)\}$
3b
i < 10
$\mathrm{RD_{OUT}}(3b) = \{(h,2),(k,?),(i,3a)\}$
4
$\mathrm{RD_{IN}}(4) = \{(h,2),(k,?),(i,3a)\}$
if (h > rand())
$\mathrm{RD_{OUT}}(4) = \{(h,2),(k,?),(i,3a)\}$
5
7
$\mathrm{RD_{IN}}(5) = \mathrm{RD_{IN}}(7) = \{(h,2),(k,?),(i,3a)\}$
k++
h++
$\mathrm{RD_{OUT}}(7) = \{(h,7),(k,?),(i,3a)\}$
$\mathrm{RD_{OUT}}(5) = \{(h,2),(k,5),(i,3a)\}$
3c
$\mathrm{RD_{IN}}(3c) = \{(h,2),(h,7),(k,5),(k,?),(i,3a)\}$
i++
$\mathrm{RD_{OUT}}(3c) = \{(h,2),(h,7),(k,5),(k,?),(i,3c)\}$

After the second iteration we have the following sets:


2
$\mathrm{RD_{IN}}(2) = \{(h,?),(k,?),(i,?)\}$
h = 0
$\mathrm{RD_{OUT}}(2) = \{(h,2),(k,?),(i,?)\}$
3a
$\mathrm{RD_{IN}}(3) = \{(h,2),(k,?),(i,?)\}$
i = 0
$\mathrm{RD_{OUT}}(3a) = \{(h,2),(k,?),(i,3a)\}$
3b
$\mathrm{RD_{IN}}(3b) = \{(h,2),(k,?),(i,3a),(h,7),(k,5),(i,3c)\}$
i < 10
$\mathrm{RD_{OUT}}(3b) = \{(h,2),(h,7),(k,?),(k,5),(i,3a),(i,3c)\}$
4
$\mathrm{RD_{IN}}(4) = \{(h,2),(h,7),(k,?),(k,5),(i,3a),(i,3c)\}$
if (h > rand())
$\mathrm{RD_{OUT}}(4) = \{(h,2),(h,7),(k,?),(k,5),(i,3a),(i,3c)\}$
5
7
$\mathrm{RD_{IN}}(5) = \mathrm{RD_{IN}}(7) = \{(h,2),(h,7),(k,?),(k,5),(i,3a),(i,3c)\}$
k++
h++
$\mathrm{RD_{OUT}}(7) = \{(h,7),(k,?),(k,5),(i,3a),(i,3c)\}$
$\mathrm{RD_{OUT}}(5) = \{(h,2),(h,7),(k,5),(i,3a),(i,3c)\}$
3c
$\mathrm{RD_{IN}}(3c) = \{(h,2),(h,7),(k,5),(k,?),(i,3a),(i,3c)\}$
i++
$\mathrm{RD_{OUT}}(3c) = \{(h,2),(h,7),(k,5),(k,?),(i,3c)\}$

3. The variable h is defined in block four and in block seven. In those two block the variable can be defined in different blocks:

   - $\mathrm{UD}(h, 4) = \{2, 7\}$
   - $\mathrm{UD}(h, 7) = \{2, 7\}$

   The variable k is only used in block five and can be defined only in this block, as a result we have:

   - $\mathrm{UD}(k, 5) = \{5, ?\}$

   From the use-definition chains it is possible to define the def-use pairs as follows:

- $h : \langle 2, 4\rangle , \langle 7, 4\rangle , \langle 2, 7\rangle , \langle 7, 7\rangle$
- $k : \langle 5, 5\rangle , \langle ?, 5\rangle$

4. We may have a problem with the variable k since in one case it has a use without definition ($\langle ?, 5\rangle$).

CHAPTER 6

Exercise session VI

6.1 Testing

Consider the function `foo`, written in a C-like language:

```
0: void foo(int a, int b) {
1:     for (int i=a; i<b; i++) {
2:         if (i % 5 == 0) {
3:             print(i)
4:         }
5:     }
6: }
```

1. Run a concolic execution starting from the following input $\{a = 1, b = 3\}$.
2. Run a concolic execution with concrete values that allow no execution of the loop.
3. Run a concolic execution with concrete values that allow the execution of the loop two times. Line three must be executed only in the second iteration.

Solution

1. The path executed with the given input is the following:

   0: $a = A(1), b = B(3)$

   1: $i = A(1), A < B$

   2: $A\%5 \neq 0$

   1: $i = A + 1(2), A + 1 < B$

   2: $A + 1\%5 \neq 0$

   1: $i = A + 2(3), A + 2 = B$

In this case we have that the test executes the loop 2 times without executing the location 3 as requested. The condition for this path are:

$$\begin{cases} A\%5 \neq 0 \\ A+1\%5 \neq 0 \\ A+2 = B \end{cases}$$

2. To avoid the loop we have to negate the condition to enter in it, that is:

$$\neg(A < B) \Rightarrow A \geq B$$

One test case that satisfies this condition is:

$$\{a = 3, b = 1\}$$

In this case the final path will be $\langle 0, 1 \rangle$.

3. Starting from the execution of the first test case we have only to negate the condition:

$$A + 1\%5 \neq 0$$

This results in the following conditions:

$$\begin{cases} A\%5 \neq 0 \\ A+1\%5 = 0 \\ A+2 = B \end{cases}$$

One possible concrete set is $\{a = 4, b = 6\}$. After running the concrete execution we have that the request is satisfied.

6.2 Testing

Consider the following function `foo2`, written in a C-like language:

```
0: void foo2(int a, int b) {
1:     int x = 1;
2:     if (a>b && a>0 && b>1) {
3:         for (int i=0; i<b; i++)
4:             x = x*a;
5:     if (x < rand(1,10))
6:         print("Log message");
7:     }
8: }
```

1. Run concolic execution to explore all possible branches.
2. Write the pseudocode of a possible generational fuzzer that should test `foo2`.
3. Given the fuzzer defined in the previous point, what is the chance of executing a path that includes location six?

4. How much time do you have to wait on average assuming that a single execution takes ≈ 1 ms?

5. Consider an SBST procedure with the objective of executing paths that reach location six. Define proper search space, neighborhood relation, and fitness function to use such approach.

Solution

1. For the concolic execution we have to run the following test cases to cover all paths:

   - Test case $\{a = 1, b = 2\}$ executes the false branch of condition two.
   - Test case $\{a = 4, b = 2\}$ executes the true branch of condition two.
   - In this case, the test case $\{a = 3, b = 2\}$ executes true branch for both conditions two and five.

   In all this cases we assumed that the `rand()` function returns always ten.

2. A possible generational fuzzer is as follows:

```
0: int[] fuzzer() {
1:     int b = rand();
2:     int a = rand();
3:     return [a,b];
4: }
```

3. Assume that `rand()` returns a random integer without constraints. To execute location six, we need to generate inputs that satisfy the following conditions:

$$A > B \wedge B = 2 \wedge A * A < rand(1, 10)$$

   The only possible solution is $\{a = 3, b = 2\}$. Assuming 32-bit encoding for integers, the range is $[-2147483648, 2147483647]$. The total chance is:

$$P(a = 3 \wedge b = 2 \wedge \texttt{rand}(1, 10) = 10) = \left(\frac{1}{4,3 \cdot 10^9}\right) \cdot \left(\frac{1}{4,3 \cdot 10^9}\right) \cdot \left(\frac{1}{10}\right) = 5,4 \cdot 10^{-21}$$

   The average number of runs is given by the inverse of the probability, that is:

$$\#\text{runs} = \frac{1}{P(a = 3 \wedge b = 2 \wedge \texttt{rand}(1, 10) = 10)} = \frac{1}{5,4 \cdot 10^{-21}} = 1,8 \cdot 10^{20}$$

4. The execution time is given by the runs multiplied by the execution of each run, that is:

$$\text{execution time} = \#\text{runs} \cdot \text{time} = 10^{20} \cdot 0,001 = 5.7 \cdot 10^9 \text{ years}$$

5. By looking at the source code we need to generate values for `a` and `b` such that both conditions two and five are true. We can now define

   - Search space: all possible pairs of integer values.

- Neighborhood relation: given a pair, we possibly modify each element of the pair by one. In this case for each pair we have eight pairs in total. For example we may have $\langle 1, 1\rangle \rightarrow \langle 0, 0\rangle, \langle 0, 1\rangle, \langle 0, 2\rangle, \langle 1, 0\rangle, \langle 1, 2\rangle, \langle 2, 0\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle$.
- Fitness function: consider the distance from the branches we want to execute. For the `if` in line two the total distance is the sum of all distances from the three conditions. Each distance should be normalized to avoid biased search (some distances may depend on very large values, some other distances on very small values):

  ```
  int dist1(int a, int b) {
      return norm(dist_subb1(a,b))+ norm(dist_subb2(a)) + norm(dist_subb3(b));
  }
  ```

  We have to do the same for the `if` in the fifth line:

  ```
  int dist2(int x, int rand(1,10)) {
      return norm(dist_subb1(rand(1,10),x));
  }
  ```

  Now we have that the fitness can be defined as the sum of `dist1` and `dist2`. When the sum is zero it means we execute the true branch of both conditions. The test case generation shall minimize the sum.