SE4HPC Exercise Book

Politecnico di Milano

First Edition

Simone Reale, Matteo Camilli, Elisabetta Di Nitto
Milano, Italia

SE4HPC Exercise Book

Politecnico di Milano

First Edition

Simone Reale, Matteo Camilli, Elisabetta Di Nitto
Milano, Italia

Self Publishers Worldwide
Seattle San Francisco New York Siziano
London Paris Campobasso Beijing Ferrazzano Madrid

This book was typeset using LaTeX software.

Preface

This document contains exercises designed to support your preparation for the written exam in the Software Engineering for HPC course. It has been compiled in response to requests from students in the 2023-2024 academic year.
A furor di popolo!

Table of Contents

Chapter 1

Availability

1.1 Brief theoretical recap

All exercises in this chapter focus on the availability analysis discussed in class. This verification technique aims to estimate the potential availability of a software system based on its architecture.

We compute availability for each operation offered by a software system to its external environment. For instance, consider a system that provides operation `x` and is composed of a component `A` and a component `B`. If the execution of `x` involves both `A` and `B`, then these components are considered to be *organized in series* with respect to the execution of `x`. Consequently, the availability of the entire system for `x` is given by:

$$A_{\texttt{A}} \cdot A_{\texttt{B}} \tag{1.1}$$

where $A_{\texttt{A}}$ and $A_{\texttt{B}}$ represent the availability of `A` and `B`, respectively. Since availability is always a number between 0 and 1, this formula indicates that the availability of a series configuration cannot exceed the availability of its weakest component.

Conversely, if operation `x` is exposed by a system *organized in a parallel* composition of two components `C` and `D` that can be used interchangeably, then the availability of the whole system for `x` is:

$$1 - (1 - A_{\texttt{C}}) \cdot (1 - A_{\texttt{D}}) \tag{1.2}$$

where A_C and A_D represent the availability of `A` and `B`, respectively. Note that $1 - A_{\texttt{comp}}$ represents the *unavailability* of an individual component `comp`. By multiplying the unavailability values, we obtain a number lower than the unavailability of the weakest component. Consequently, the unavailability in a parallel composition decreases, leading to an increase in overall availability.

This approach can also be applied to systems with more than two components with mixed configurations, where some parts are arranged in series and others in parallel. In the following section, we will explore several examples.

1.2 Collection and analysis

Description

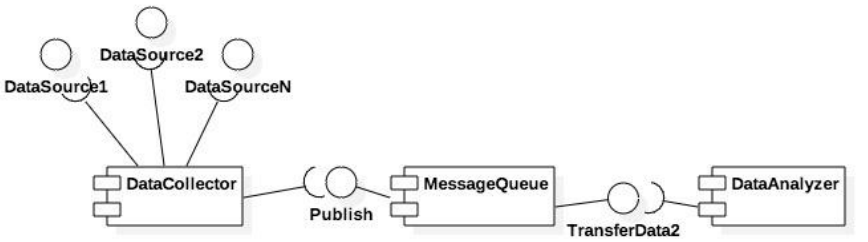


Figure 1.1: Collection and analysis system.

Consider the system above. Assume that the participating components offer the following availability:

- DataCollector: 99%
- MessageQueue: 99.99%
- DataAnalyzer: 99.5%

Questions

1. Provide an estimate of your system's total availability (a rough estimation is acceptable without complete calculations). Assume you want to improve this total availability through replication, which component(s) would you choose to replicate? Explain your reasoning.
2. How would such replication impact the way the system works and is designed?

Solution

1. Data must flow through the entire chain of components to be processed, implying that the system can be modeled as a series of components. Consequently, the total availability A_{Total} of the system is determined by its weakest element, which in this case is the `DataCollector`.

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

If we parallelize the `DataCollector` element by adding a new replica, we obtain the following availability:

$$A_{Total} = (1 - (1 - 0.99)^2) \cdot 0.9999 \cdot 0.995 = 0.995$$

2. If both `DataCollector` replicas acquire information from the same sources (hot spare approach), the other components will receive duplicate data and must be designed to handle this. For instance, the `MessageQueue` could discard all duplicates. Additionally, both the `DataSources` and the `MessageQueue` need to implement mutual exclusion mechanisms to ensure that communication between them and the two `DataCollector` replicas does not cause concurrency issues. Alternatively, only one `DataCollector` replica could be active at a time, with the second replica activated only when needed, such as if the first one fails to send feedback within a certain timeout period (warm or cold spare).

1.3 Train Ticket

Description

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.

- Search request: `/APIv1/search/{args}`
- Reserve request: `/APIv1/reserve/{args}`

Requests (search and reserve) are received and dispatched by the API gateway. In particular, Figure 1.2 shows how requests propagate from the gateway to internal microservices. Note that in this example, reserve also includes the purchase of reserved items.

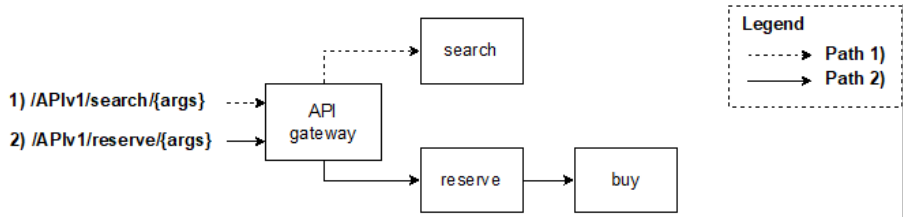


Figure 1.2: train ticket system

Microservices run in units deployed onto 2 different Virtual Machines (VMs), 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

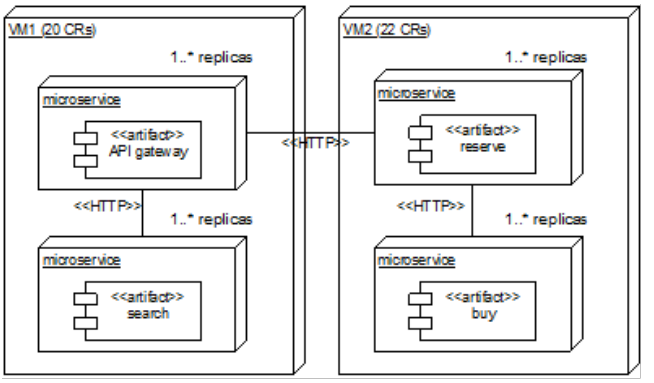


Figure 1.3: Deployment

- 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 in Figure 1.3 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.

R_1: *"Both search and reserve services exposed through API gateway shall have availability greater than or equal to 0.99."*

Questions

1. Considering the constraints of the execution environment represented above, determine whether requirement R_1 can be satisfied or not assuming the following availability estimates for each microservice.
    - API gateway: 0.99
    - search: 0.98

- reserve: 0.95
- buy: 0.9

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.

   (a) search: 50 users

   (b) reserve: 90 users

   Assume 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. After a preliminary analysis, we realize that availability depends on the workload according to the estimates presented in Table 1.1. Does the execution environment have enough computational resources to support the workload defined above still fulfilling requirement R1 and the defined constraints? Justify your answer.

LOW workload 0-60 concurrent users		**HIGH workload** 60-150 concurrent users	
Microservice	**Availability**	**Microservice**	**Availability**
`API gateway`	0.99	`API gateway`	0.98
`search`	0.98	`search`	0.95
`reserve`	0.95	`reserve`	0.93
`buy`	0.91	`buy`	0.90

Table 1.1: Microservices availability under different workloads.

Solution

1. Considering the execution environment, we can derive the following inequations constraining the number of replicas:

   - Constraints extracted from environment:

   $$2x + 5y \leq 20$$
   $$4u + 5z \leq 22$$

   - Constraints extracted from requirement R_1:

   $$(1 - (1 - 0.99)^x) \cdot (1 - (1 - 0.98)^y) \geq 0.99$$
   $$(1 - (1 - 0.99)^x) \cdot (1 - (1 - 0.95)^u) \cdot (1 - (1 - 0.91)^z) \geq 0.99$$

   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:

$$x = 2, \quad y = 2, \quad u = 3, \quad z = 2$$

2. 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 $x = 2$, $y = 2$, $u = 3$, $z = 2$ that is again feasible according to environment constraints.

1.4 Circuit breaker

Description

Consider the following high-level component structure that is a static description of a possible microservices-based system implementing the business logic for two applications (App A, and App B). The UML Component Diagram below highlights the interfaces (both provided and required ones) of microservices `MS1`,..., `MS6`. Moreover, the diagram shows that MS5 writes data on a Network-Attached Storage (NAS) device, and MS6 reads and writes data on a local cache.

Questions

1. Based exclusively on the static information presented in the Component Diagram, enumerate all possible execution paths (in terms of sequences of microservices) triggered by requests generated by App A and App B, respectively. Notice that we are interested only in paths that terminate with a microservice that does not require any further interface. Then, describe the ripple effect possibly generated by a sudden slowdown of the NAS on each identified path. Assume every request to `MS5` causes interactions with the NAS. Identify the application(s) affected by this disruptive event. Justify your answer.

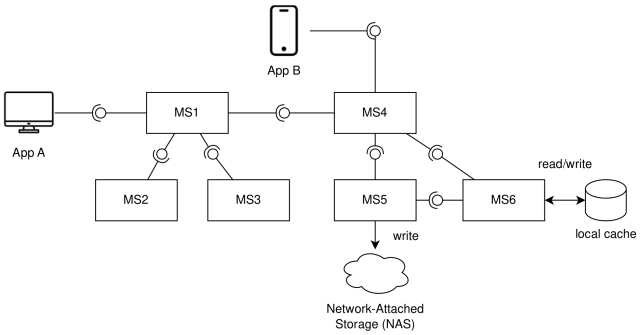


Figure 1.4: A high-level component diagram of the microservice-based system.

Application	Workload
App A	100
App B	150

Table 1.2: Workload for applications A and B.

2. Assume now that the expected number of requests per second (workload) for the two applications is as follows:

   You can also assume the following behavior:

   (a) Requests directed to `MS1` are always distributed evenly among all microservices used by `MS1`.

   (b) 25% of the requests to MS4 are write operations and are transferred to `MS5`, which, in turn, executes them into the NAS and transfers them to MS6 for local cache updates.

   (c) 75% of the requests to `MS4` are read-only operations and are transferred to `MS6`.

   (d) The number of requests generated by `MS2` is always negligible.

   Address the following points:

   - Determine the number of requests per second that reach each microservice.
   - Use the numbers you have computed to determine the availability of each microservice replica according to the following rules: if the total number of requests a microservice receives is less than 60, the availability estimate is 0.99; if the number of requests is between 60 and 80 the estimate is 0.98, 0.97 otherwise. These estimates apply to each of the considered microservices.
   - Focus now on the write operations generated by App B and on the corresponding path. What is the availability shown by App B for write operations when a single replica of each microservice is used?

- If we want to satisfy the following nonfunctional requirement: "*The total availability of App B for write operations shall be at least 0.999, under the condition that any circuit breaker is closed*", what is the required minimum number of replicas for each of the involved microservices?

Solution

1. According to static information only, the possible paths triggered by either App A or App B are as follows:

   - **App A**
     - App A → `MS1` → `MS3`
     - App A → `MS1` → `MS4` → `MS5` → `MS6`
     - App A → `MS1` → `MS4` → `MS6`
   - **App B**
     - App B → `MS4` → `MS5` → `MS6`
     - App B → `MS4` → `MS6`

   A sudden slowdown of the NAS could cause a growing number of pending requests to `MS5` due to the high number of concurrent (slow) operations of the NAS. This could cause a saturation of the resources of `MS5`. Therefore, pending requests to `MS4` also grow, potentially saturating its available resources.

   Essentially, the ripple effect follows all the paths listed above that use `MS5` or `MS4` in a backward manner. Thus, according to the paths, the effect potentially propagates back to App A and B.

2. Given the defined workload, we have the following concurrent requests per each microservice.

   - `MS1` = 100 requests
   - `MS2` negligible
   - `MS3` = 50 requests
   - `MS4` = $150 + 50 = 200$ requests
   - `MS5` = $200 \cdot 0.25 = 50$ requests
   - `MS6` = $200 \cdot 0.75 + 50 = 200$ requests

   Under such conditions, individual availability estimates are as follows.

   - `MS1` = 0.97
   - `MS2` = 0.99
   - `MS3` = 0.99

- `MS4` = 0.97
- `MS5` = 0.99
- `MS6` = 0.97

The path followed by write operations incoming from App B is `MS4` → `MS5` → `MS6`, which, for this purpose, are organized in a series. The availability is then:

$$0.97 \cdot 0.99 \cdot 0.97 = 0.93$$

We need to introduce additional replicas for all three involved microservices to satisfy the non-functional requirement. In general, we should satisfy the following constraint:

$$(1 - (1 - 0.97)^x) \cdot (1 - (1 - 0.99)^y) \cdot (1 - (1 - 0.97)^z) > 0.999$$

where x, y, z represent the number of replicas for `MS4`, `MS5`, and `MS6`, respectively.

Assignments satisfying the previous condition are:

- $x = 2, y = 3, z = 3$ and $x = 3, y = 3, z = 2$ that lead to Avail = 0.9991
- $x = 3, y = 2, z = 3$ that leads to Avail = 0.9998

Instead, instantiating only two replicas per service leads to availability equal to 0.9981, while two replicas for `MS5` and `MS6` or `MS4` and three replicas for the remaining service leads to availability equal to 0.99897312, which is still slightly lower than 0.999.

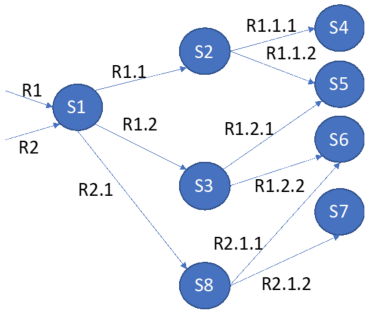
1.5 Microservice architecture

Description

Consider the microservice-based architecture shown in the Figure 1.5 below. The architecture is organized into eight stateless microservices collaborating 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 $S7$.

Questions

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 request $R1$?



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

Figure 1.5: Microservice-based architecture.

2. If each of the services $S1$-$S8$ is duplicated, what is the new availability value computed at point 1?

Solution

1. Since we have no information that suggests the opposite, we can assume that the availability of each service can be associated with any of the requests the service is able to answer. This implies that, for instance, the availability associated with request $R1.1.2$ is the same as request $R1.2.1$ and corresponds to the availability of service $S5$. Considering that the fulfillment of $R1$ requires, besides the execution of $S1$, the delegation of the corresponding sub-requests to the other services, we can write:

$$\begin{aligned} AR1 &= AS1 \cdot AR1.1 \cdot AR1.2 \\ &= AS1 \cdot (AS2 \cdot AS4 \cdot AS5) \cdot (AS3 \cdot AS5 \cdot AS6) \\ &= 0.753 \end{aligned}$$

   Notice that $S5$ is involved in two sub-requests and, therefore, its availability occurs twice in the expression above.

2. If all services are duplicated, we can replace in the formula above the availability of each service with the one obtained by duplicating it, as follows:

$$A'_{Si} = 1\text{–}(1 - A_{Si})^2$$

$$A_{R1} = A'_{S1} \cdot (A'_{S2} \cdot A'_{S4} \cdot A'_{S5}) \cdot (A'_{S3} \cdot A'_{S5} \cdot A'_{S6}) = 0.977$$

1.6 Image recognition workflow

Description

Consider a workflow made of the following 4 steps:

1. image acquisition
2. image recognition of feature 1
3. image recognition of feature 2
4. decision

where steps 2 and 3 (the recognition of specific features of the image) are executed in parallel, but the results of both steps are necessary to make the final decision. We have several components that can perform the various steps of the workflow. In particular, step 1 is executed by components whose availability is 95%; step 2 is executed by components whose availability is 80%; step 3 is executed by components whose availability is 90%; step 4 is executed by components whose availability is 99%.

Questions

1. Design an architecture of the system whose availability is 99%.

Solution

1. Although steps 2 and 3 are executed in parallel since both of their results are necessary for the execution of step 4, the workflow is to be considered, from the availability point of view, a series of the 4 steps. Since we need an overall availability of 99%, and each component has an availability of 99% or less, we must replicate each component to increase the availability of each individual part.

   In particular, we have the following:

   - The parallel of 2 components for step 1 achieves an availability, for that step, of 99.7%.
   - The parallel of 4 components for step 2 achieves an availability, for that step, of 99.8%.
   - The parallel of 3 components for step 3 achieves an availability, for that step, of 99.9%.
   - The parallel of 2 components for step 4 achieves an availability, for that step, of 99.99%.

   All in all, the resulting total availability is $0.997 \cdot 0.998 \cdot 0.999 \cdot 0.9999 = 0.994$, which is enough.

   Note that if we decide to have only one component for step 4, we have $0.997 \cdot 0.998 \cdot 0.999 \cdot 0.99 = 0.985$, which is not enough, instead.

1.7 Publish-Subscribe layer

Description

Consider the following UML Component Diagram, describing the architecture of the core of a publish-subscribe layer (Figure 1.7): The layer is in charge of

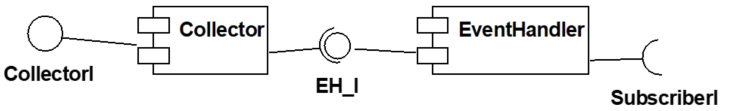


Figure 1.6: The publish-subscribe layer UML component diagram.

receiving published events through the Collector and of sending these events to subscribers through the EventHandler (which assumes subscribers offer the SubscriberI interface to allow for the delivery of events). The EventHandler stores the table internally, keeping track of event subscribers; we assume that the table does not change over time (or it changes so infrequently that, for our purposes, we can consider it fixed). The Collector component is a lightweight worker node that simply takes incoming requests for event publication and dispatches them to the handler.

Questions

1. After the first version of this system is delivered, developers realize that there are 2 kinds of events: high-priority ones (e.g., alarms) and medium-priority ones (e.g., warnings). Describe how developers could change the architecture above to make sure that the handling of high-priority events is not negatively affected by the presence of medium-priority ones. Provide a new version of the component diagram with your solution and provide a motivation for your choice.

2. Suppose that we want the delivery of high-priority messages to have a level of availability of 99.99% and that of medium-priority messages to have a level of availability of 99%. Suppose also that we have at our disposal 2 servers that have 99% availability, 2 servers that have 95% availability, 2 servers that have 90% availability, and 4 servers that have 80% availability. How would you deploy the components of your component diagram to achieve the desired levels of availability?

Solution

1. A possible way to solve the issue is to introduce in the architecture separate components to handle the different types of events (high-priority, or HP, ones, and medium-priority, or MP, ones). This architecture (fig-

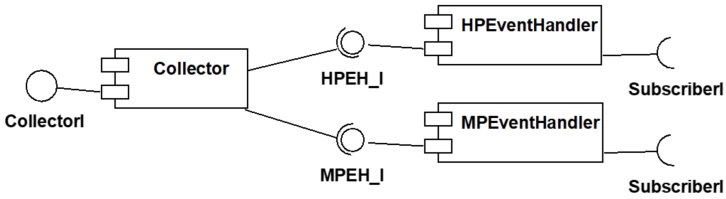


Figure 1.7: Possible architecture

ure 1.7) fosters scalability since each component can be replicated to handle increasing loads of requests. The replication requires the creation of a new copy of the EventHandlers subscription table, but the assumption about the stability of this table can make this copy less critical. One can indeed imagine that there is a fixed pool of Collector components, but the number of handlers for each type of event can increase or decrease based on the current load.

2. The delivery of each type of message involves the series of the Collector component(s) and the corresponding handler (so, the chain for the delivery of HP events is given by the replicas of the Collector component and the replicas of the HPEventHandler component). The Collector component(s) belongs to each chain, so it must have an availability that allows us to meet all constraints, in particular those for the delivery of HP events. Hence, it is reasonable to use the most available servers, but we also need to leave some highly available servers for the second part of the chain of HP events, i.e., HPEventHandler. Hence, if we use (in parallel) 1 server with 99% availability and 2 servers with 95% availability, we reach an availability for Collector that is 99.998%. Similarly, if we have in parallel 1 server with 99% availability, 2 with 90% availability, and 1 with 80% availability for HPEventHandler, we obtain, for this component, 99.998% availability, and the chain of the two components has 99.996 availability, which meets the desired threshold. Then, concerning the chain for delivering MP events, we use the remaining 3 servers with 80% availability to replicate component MPEventHandler. The parallel of the 3 servers gives an availability of 99.2%, and the series with the Collector component gives an availability of 99.198%, which is enough for our purposes.

Chapter 2

Petri Nets

2.1 Brief theoretical recap

Petri nets is a formal modeling language to describe distributed and parallel systems.

A Petri net, or place-transition (P/T) net, is defined as follows: $P/T = (P, T, F, W, M_0)$, where P is a set of places, T is a set of transitions, F is a flow relation between places and transitions and vice versa, W is a weight function that assigns a strictly positive integer to elements of the flow relation, and, finally, M_0 is an initial marking which associates to each place in the net a number of *tokens*.

In the modeling process, subsets of places and a corresponding marking can represent states of the system, while transitions can represent the evolution of the system from a state to the next one. For instance, a one-position buffer can be represented by:

- two places, let's call them `empty` and `full`;
- two transitions, `getFromBuffer` having as input place `full` and output place `empty` and `insertInBuffer`, having as input place `empty` and output place `full` (notice that the flow relation F is the one that defines the connection between places and transitions and vice versa);
- a weight function assigning 1 to all elements of the flow relation;
- an initial marking where `empty` includes one token and `full` none.

Given this model, the places `empty` with one token and `full` with no token represent the empty buffer state, while the same two places with a single token in `full` represent the full buffer state. Moreover, the two transitions represent the processes that determine the insertion and extraction of a new token in the buffer. Remember that a transition can *fire* only if each input place has a number of tokens greater or equal to the weight of the arc from the place to

the transition. Upon firing, the transition eliminates from the input places a number of tokens equal to the weights of the corresponding arcs and, at the same time, inserts in the output places a number of tokens depending on the weights of the outgoing arcs.

The reachability graph of a P/T net is a directed graph, G=(V, E), where each node, $v \in V$, represents a reachable marking, and each edge, $e \in E$, represents a transition between two reachable markings. The main steps for the construction of the reachability graph of a marked net (N,m0) are summarized in the following algorithm.

1. **Initialization:** The initial node of the graph is the initial marking m_0. This node is initially unlabeled.
2. **Consider an unlabeled node m of the graph.**
   (a) For each transition t enabled at m, i.e., such that $m \geq \text{Pre}[\cdot, t]$:
      i. Compute the marking $m' = m + C[\cdot, t]$ reached from m firing t.
      ii. If no node m' is on the graph, add a new node m' to the graph.
      iii. Add an arc t from m to node m'.
   (b) Label node m as "old".
3. If there exist nodes with no label, go to Step 2.

2.2 Logical And

Description

Define a P/T net model for calculating the logical conjunction of two variables, x, and y, each of which takes the values *true* or *false*, with each variable's value being independent of the other.

Questions

1. Draw the diagram describing the aforementioned P/T net model.
2. Define a P/T net that computes the negation of x, which again takes only the values *true* and *false*.
3. Define a P/T net that computes $!(x \wedge y)$ by composing the previous models.

Solution

1. To see how the P/T net works, we walk through the firing sequence corresponding to $x = true$ and $y = false$. We start with one token in the places corresponding to $x = true$ and $y = false$, the leftmost and rightmost places on the top line of Figure 2.1. Only one transition is enabled:

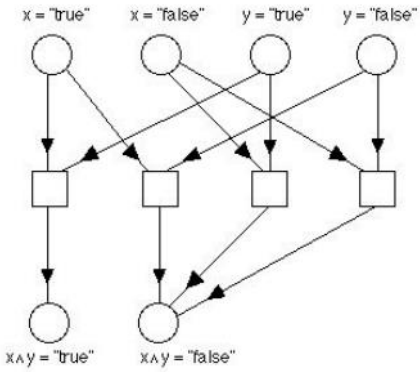


Figure 2.1: P/T net modeling the logical and.

the second one from the left. Firing that transition puts one token into place labeled $x \wedge y = \mathit{false}$. No other transitions can fire, and the desired result is obtained.

2. If we want to know the value of $!x$ when x has a certain value, we put one token in place for that value, carry out the firing, and observe which place has a token in it.

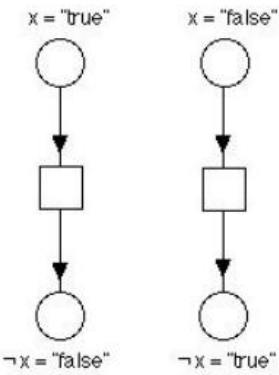


Figure 2.2: P/T net modeling the not operator

3. Figure 2.3 shows the two previous nets combined to model a logical nand.

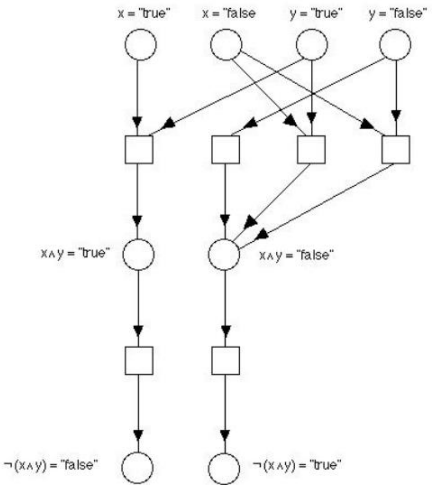


Figure 2.3: Petri net modelling $!(x \wedge y)$.

2.3 Reachability graph

Description

Consider the P/T net $(N, M0)$ presented in Figure 2.4.

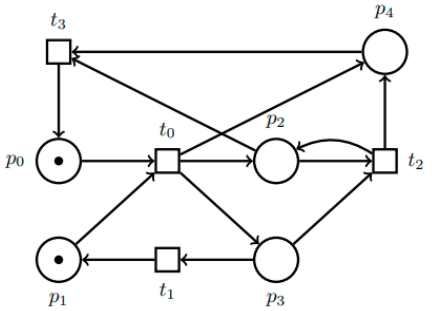


Figure 2.4: An example of P/T net model.

Questions

1. Define the reachability graph of $(N, M0)$.

Solution

1. Figure 5.5 shows the reachability graph of the P/T net.

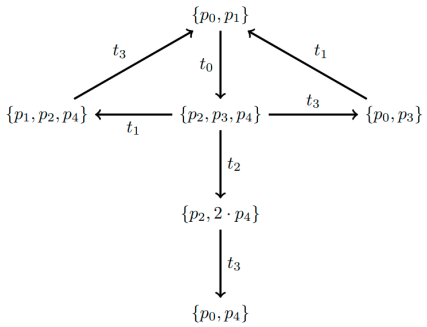


Figure 2.5: Reachability graph

2.4 Wittgenstein and friends

Description

Four philosophers, Simone, Ludwig, Elisabetta, and Bertrand, sit around a round table. There are forks on the table, one between each pair of philosophers. The philosophers want to eat spaghetti "alla carbonara" from their plats (situation depicted in Figure 2.6). Unfortunately, plates are full of egg yolk and are particularly slippery, so a philosopher needs both forks to eat the carbonara. The philosophers have agreed on the following protocol:

- Initially, philosophers think about philosophy.
- When they get hungry, they do the following:
    1. Take the left fork.
    2. Take the right fork and start eating.
    3. Return both forks simultaneously and repeat from the beginning.

Questions

1. Define a P/T net modeling the behavior of each philosopher as it was alone.
2. Starting from the model defined in the first step, define a P/T model of the whole system.

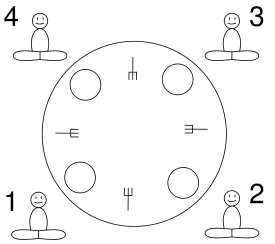


Figure 2.6: Philosophers' dining table

3. Will the philosophers starve to death (because the system reaches a deadlock)?

4. Will an individual philosopher eventually eat, assuming he or she wants to?

Solution

Solution to point 1: Figure 2.7 shows a P/T net modeling the behaviour of a single phisolopher. `Fork1 available` and `Fork2 available` represent the availability of the needed resources (the two forks). Following the description in the text, from the initial marking the enabled transition is `Take fork`. Its firing consumes the tokens in `Fork1 available` and `Thinking` and produces a token in the place `Ready to take the next fork`. At this point, transition `Take fork & start eating` is enabled. Its firing consumes the tokens from `Fork2 available` and `Ready to take the next fork` and produces a token in the place `Eating`. While eating, the two forks are not available for other uses. Finally, when transition `Finish eating` fires, the two forks become both available and the philosopher enters again in the thinking state.

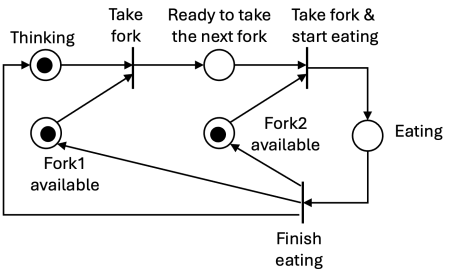


Figure 2.7: P/T net modeling the dining philosophers problem.

Solution to point 2: Figure 2.8 shows the P/T net representing the case of the four philosophers. You can note the structure of the net in Figure 2.7 is replicated four times. We have four places representing the availability of the four forks. Each of such places is shared between two net fragments representing two different philosophers.

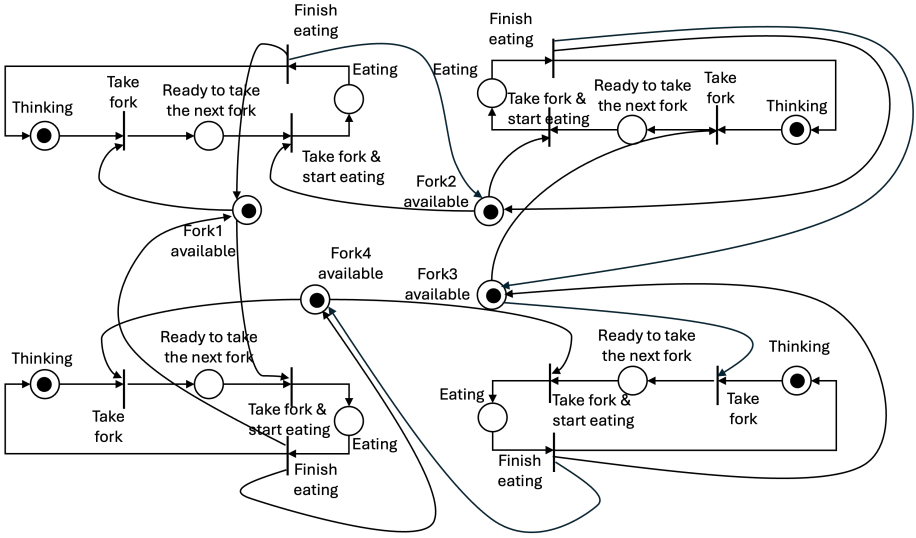


Figure 2.8: Alternative P/T net modeling the dining philosophers problem.

Solution to point 3: Yes, if all philosophers pick up their left fork simultaneously, the system will reach a deadlock, and all philosophers will starve. Try to define the reachability graph and verify this property.

Solution to point 4: Starvation (cf. Starvation concept in P/T nets) occurs if a philosopher is perpetually unable to acquire both forks and thus never gets to eat. In the deadlock scenario, all philosophers starve because none can transition to eating; without deadlocks, an individual philosopher could eventually eat.

2.5 Reachability graph 2

Description

Questions

1. Draw the reachability graph of the P/T net in Figure 2.9.

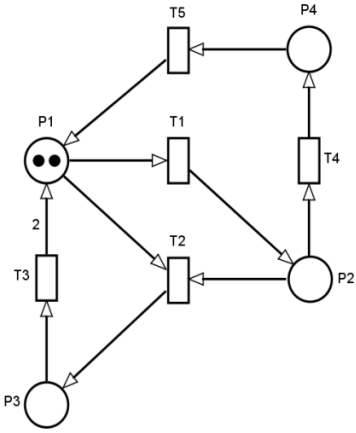


Figure 2.9: Example of a P/T net.

Solution

1. Figure 2.10 shows the reachability graph of the proposed example. Nodes are represented by tuples of numbers that represent the number of tokens in the individual places of the P/T net. In this case, places are four; the first position in each tuple corresponds to the number of tokens in `P1`, the second to the ones in `P2`, and so on.

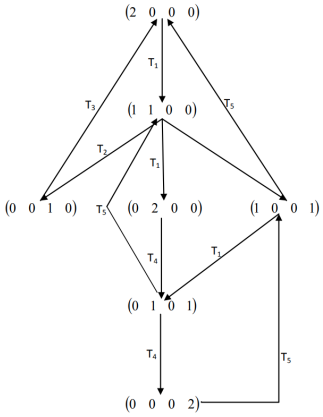


Figure 2.10: Reachability graph of the P/T net depicted in Figure 2.9.

2.6 Train tracks

Description

Four cities are connected by unidirectional train tracks building a circle. Two trains circulate on the tracks. We model the four tracks by places s_1,..., s_4. A token on s_i means a train is in the i-th track.

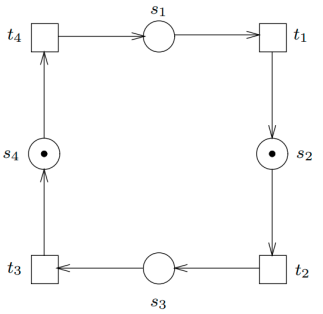


Figure 2.11: A P/T net modeling the train tracks problem.

Questions

1. Modify the P/T net model so that two trains can never occupy the same track.
2. How can we prove that everything works?

Solution

1. The four control places l_1, ... , l_4 guarantee that no reachable marking puts more than one token on s_i.

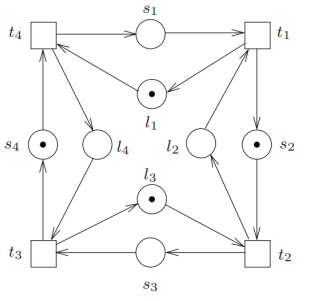


Figure 2.12: The modified P/T net.

2. This property can be proven using the reachability graph. Since every reachable marking puts at most one token on a place, we denote a marking by the set of places marked by it.

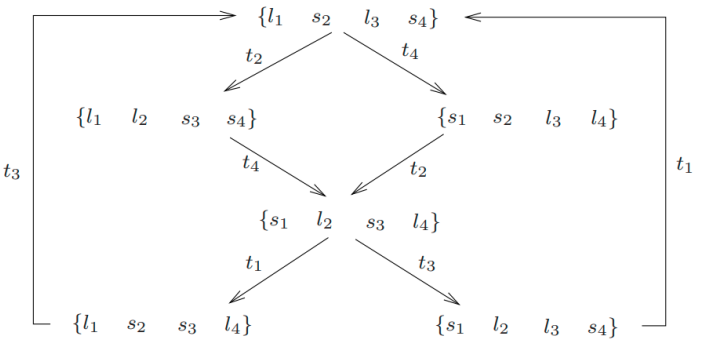


Figure 2.13: Reachability graph

Chapter 3

Symbolic Execution

3.1 Brief theoretical recap

Symbolic execution is a code analysis technique aimed at verifying reachability and path feasibility properties. It involves assigning symbolic values to input variables within a code fragment and simulating the execution of the code with these assigned symbols. As the execution progresses, a *path condition* emerges. A point in the code is considered reachable if there exists at least one path leading to that point that corresponds to a satisfiable path condition. Similarly, a path is deemed feasible if its corresponding path condition is satisfiable.

In essence, a path condition is considered satisfiable if it is possible to identify an assignment of concrete values to the variables that renders the condition true.

3.2 Simple Computation

Description

Consider the following fragment of code:

```
0  int computation(int a[], int n){
1    int i, count, prod;
2    if (n < 2)
3        printf("%d", n);
4    i = 0;
5    count = 0;
6    prod = 1;
7    while (i < n){
8        if (a[i] == 0)
9            count++;
10       prod = prod * a[i];
11       i++;
12   }
13   if (count < 2 || prod == 0)
```

```
14      return -1;
15    else
16      return prod;
17  }
```

Questions

1. Derive the path condition corresponding to the execution of path 1, 2, 4, 5, 6, 7, 8, 10, 11, 12, 7, 8, 9, 10, 11, 12, 7, 13, 14.

2. Derive the path condition corresponding to the execution of path 1, 2, 4, 5, 6, 7, 13, 15.

Solution

Tables 3.1 and 3.2 present the symbolic values of all relevant variables. In the first column the code line numbers are shown while the last column includes the path condition.

As for the first path, we can say that it is feasible as we can identify concrete values for `a` and `n` such that the corresponding path condition is fulfilled.

As for the second path, it is certainly not feasible because the condition $N \geq 2 \wedge N \leq 0$ contains a clear contradiction. For this reason, we stopped the symbolic execution at line 7: it is impossible that, from that point, the program continues the execution on the specified path.

#	a	a[0]	a[1]	n	i	count	prod	π
0	A	A[0]	A[1]	N				
1	A	A[0]	A[1]	N				
2		A[0]	A[1]					$N \geq 2$
4	A	A[0]	A[1]	N	0			$N \geq 2$
5	A	A[0]	A[1]	N	0	0		$N \geq 2$
6	A	A[0]	A[1]	N	0	0	1	$N \geq 2$
7	A	A[0]	A[1]	N	0	0	1	$N \geq 2 \wedge 0 < N$
8	A	A[0]	A[1]	N	0	0	1	$N \geq 2 \wedge A[0] \neq 0$
10	A	A[0]	A[1]	N	0	0	A[0]	$N \geq 2 \wedge A[0] \neq 0$
11	A	A[0]	A[1]	N	1	0	A[0]	$N \geq 2 \wedge A[0] \neq 0$
12	A	A[0]	A[1]	N	1	0	A[0]	$N \geq 2 \wedge A[0] \neq 0$
7	A	A[0]	A[1]	N	1	0	A[0]	$N \geq 2 \wedge 1 < N \wedge A[0] \neq 0$
8	A	A[0]	A[1]	N	1	0	A[0]	$N \geq 2 \wedge A[0] \neq 0 \wedge A[1] = 0$
9	A	A[0]	A[1]	N	1	1	A[0]	$N \geq 2 \wedge A[0] \neq 0 \wedge A[1] = 0$
10	A	A[0]	A[1]	N	1	1	A[0]*A[1]	$N \geq 2 \wedge A[0] \neq 0 \wedge A[1] = 0$
11	A	A[0]	A[1]	N	2	1	A[0]*A[1]	$N \geq 2 \wedge A[0] \neq 0 \wedge A[1] = 0$
12	A	A[0]	A[1]	N	2	1	A[0]*A[1]	$N \geq 2 \wedge A[0] \neq 0 \wedge A[1] = 0$
7	A	A[0]	A[1]	N	2	1	A[0]*A[1]	$N \geq 2 \wedge N \leq 2 \wedge A[0] \neq 0 \wedge A[1] = 0$
13	A	A[0]	A[1]	N	2	1	A[0]*A[1]	$N = 2 \wedge A[0] \neq 0 \wedge A[1] = 0$
14	A	A[0]	A[1]	N	2	1	A[0]*A[1]	$N = 2 \wedge A[0] \neq 0 \wedge A[1] = 0$

Table 3.1: Symbolic execution of the Simple Computation example for the path 1, 2, 4, 5, 6, 7, 8, 10, 11, 12, 7, 8, 9, 10, 11, 12, 7, 13, 14.

#	a	a[0]	a[1]	n	i	count	prod	π
0	A	A[0]	A[1]	N				
1	A	A[0]	A[1]	N				
2	A	A[0]	A[1]	N				$N \geq 2$
4	A	A[0]	A[1]	N	0			
5	A	A[0]	A[1]	N	0	0		
6	A	A[0]	A[1]	N	0	0	1	
7	A	A[0]	A[1]	N	0	0	1	$N \geq 2 \wedge N \leq 0$
13								
15								

Table 3.2: Symbolic execution of the Simple Computation example for the path 1, 2, 4, 5, 6, 7, 13, 15.

3.3 Decode

Description

You can assume that function `hex_vals` takes as input a char representing a hexadecimal value (between 0 and F) and returns an int value, that is, its decimal representation (-1 if the input is not a hexadecimal value). Function unicode, instead, takes an int value as input and returns a char, that is, the corresponding unicode representation.

```
0  int decode(char s[], char t[]) {
1      char c, dh, dl;
2      int v, i = 0, j = 0;
3      while (i < strlen(s)) {
4          c = s[i];
5          if (c == '+') {
6              t[j++] = ' ';
7          } else if (c == '%' && i + 2 < strlen(s)) {
8              if (hex_vals(s[i + 1]) > 0 && hex_vals(s[i + 2]) > 0) {
9                  v = hex_vals(s[i + 1]) * 16 + hex_vals(s[i + 2]);
10                 t[j++] = unicode(v);
11             } else {
12                 return -1;
13             }
14             dh = s[i + 1];
15             dl = s[i + 2];
16             i += 2;
17         } else {
18             t[j++] = c;
19         }
20         i++;
21     }
22     t[j] = '\0';
23     return 0;
24 }
```

Questions

1. Consider the following execution path: 0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 14, 15, 16, 20, 3, 22, 23 Symbolically execute the function and check whether the

give path is feasible.

Solution

Table 3.3 shows the symbolic values associated to each variable during the execution of the given path. The first column lists all executed lines of code. To keep the table compact, we have used the symbols U and H to denote the values assumed by $t[1]$ and v, respectively, where we have:

$$U = unicode(hex\_vals(S[1]) \cdot 16 + hex\_val(S[2]))$$

$$H = hex\_vals(S[1]) \cdot 16 + hex\_val(S[2])$$

#	s	t	t[1]	c	dh	dl	v	i	j	π
0	S	T								
2	S	T						0	0	
3	S	T						0	0	$\|S\| > 0$
4	S	T		S[0]				0	0	
5	S	T		S[0]				0	0	$\|S\| > 0 \wedge S[0] \neq' +'$
7	S	T		S[0]				0	0	$S[0] =' \%' \wedge \|S\| > 2$
8	S	T		S[0]				0	0	$S[0] =' \%' \wedge \|S\| > 2 \wedge hex(S[1]) \wedge hex(S[2])$
9	S	T		S[0]			H	0	0	$S[0] =' \%' \wedge \|S\| > 2 \wedge hex(S[1]) \wedge hex(S[2])$
10	S	T	U	S[0]			H	0	1	$S[0] =' \%' \wedge \|S\| > 2 \wedge hex(S[1]) \wedge hex(S[2])$
14	S	T	U	S[0]	S[1]		H	0	1	$S[0] =' \%' \wedge \|S\| > 2 \wedge hex(S[1]) \wedge hex(S[2])$
15	S	T	U	S[0]	S[1]	S[2]	H	0	1	$S[0] =' \%' \wedge \|S\| > 2 \wedge hex(S[1]) \wedge hex(S[2])$
16	S	T	U	S[0]	S[1]	S[2]	H	2	1	$S[0] =' \%' \wedge \|S\| > 2 \wedge hex(S[1]) \wedge hex(S[2])$
20	S	T	U	S[0]	S[1]	S[2]	H	3	1	$S[0] =' \%' \wedge \|S\| > 2 \wedge hex(S[1]) \wedge hex(S[2])$
3	S	T	U	S[0]	S[1]	S[2]	H	3	1	$S[0] =' \%' \wedge \|S\| > 2 \wedge hex(S[1]) \wedge hex(S[2]) \wedge \|S\| <= 3$
22	S	T	\0	S[0]	S[1]	S[2]	H	3	1	$S[0] =' \%' \wedge hex(S[1]) \wedge hex(S[2]) \wedge \|S\| = 3$
23	S	T	\0	S[0]	S[1]	S[2]	H	3	1	$S[0] =' \%' \wedge hex(S[1]) \wedge hex(S[2]) \wedge \|S\| = 3$

Table 3.3: Symbolic execution of the Decode example.

In conclusion, the path condition that corresponds to the given path is $S[0] =' \%' \wedge hex(S[1]) \wedge hex(S[2]) \wedge |S| = 3$. It is satisfiable as we can identify actual input values that fulfill it.

3.4 Foo Fighters

Description

Consider the following C function foo:

```
0  int foo(int a[], int b, int c) {
1      int r, x = 0;
2      int y = 0;
3      if (a[b] >= 0) {
4          y = 7;
5      }
6      else {
7          y = 5;
8      }
9      a[y] = c;
10     if (a[b] < 3) {
11         x = 2;
12         if (a[b] <= 0 || c != 0) {
13             y = y - 3;
14         }
15     }
16     r = a[x + y - 2];
17     return r;
18 }
```

Note: you can assume that the array $a[\,]$ has 10 elements and $0 \leq b \leq 9$ holds.

Questions

1. Consider the function foo and carry out a symbolic execution for all paths (note that, in this case, the number of paths is finite). Given that this code fragment does not contain loops, you may be facilitated in your work if you create a binary tree to represent these paths. The nodes of the tree shall include at least the initial symbolic assignment in lines 0-2 (root), the final assignments (leaves), and all the conditional statements (internal nodes). You can annotate the arcs with the conditions and assignments that are relevant to the corresponding part of the code. For each path (i.e., each leaf of the previous tree structure), determine the symbolic condition that ensures the execution of the corresponding path. Then, identify the unfeasible path(s), if any.

   Example (tree structure): the following piece of tree (figure 3.1) represents a portion of the symbolic paths extracted from a program where lines h and $h + 2$ are conditions, between h and $h + 2$, there are no conditions, and there is a new assignment for variable y (where the symbolic value of variable x before line $h + 1$ is X).

2. Suppose that we wanted to guarantee that the result of the foo function is not zero. According to the previous analysis, try to determine a pre-condition among the possible ones that ensures the satisfaction of this property.

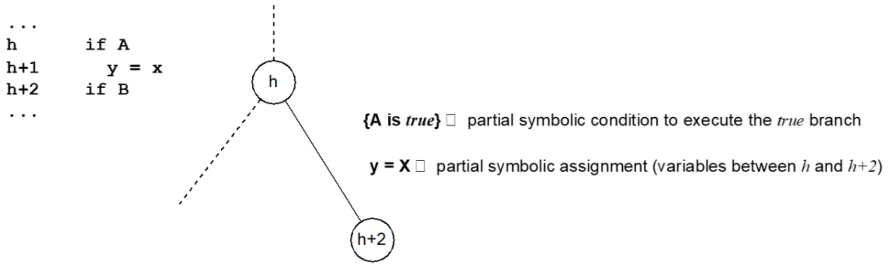


Figure 3.1: Example of tree.

Solutions

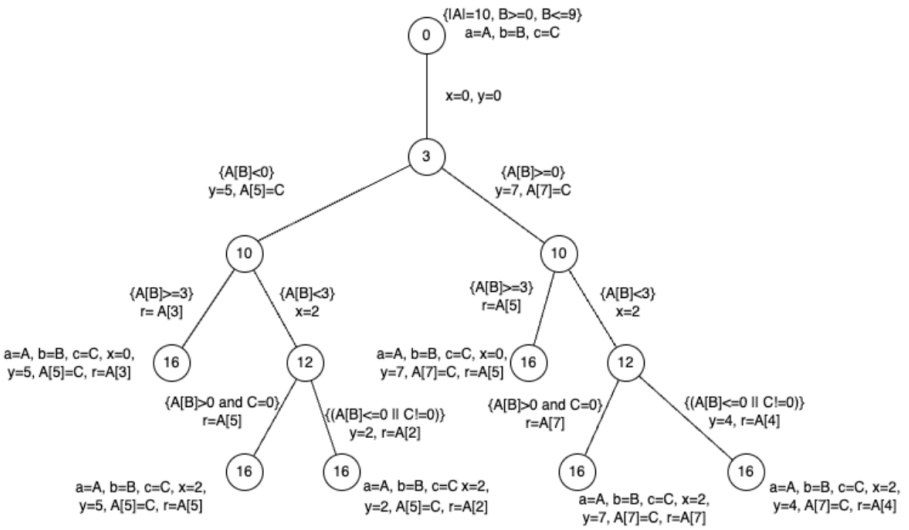


Figure 3.2: Tree structure

1. There are 6 leaves (from left to right) with the following path conditions:
   - $\{A[B] < 0$ and $A[B] \geq 3\}$
     Path: $0, 1, 2, 3, 6, 7, 8, 9, 10, 15, 16, 17$ **(unfeasible)**
   - $\{A[B] < 0$ and $A[B] < 3$ and $A[B] > 0$ and $C = 0\}$
     Path: $0, 1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17$ **(unfeasible)**
   - $\{A[B] < 0$ and $A[B] < 3$ and $(A[B] \leq 0$ or $C \neq 0)\}$
     Path: $0, 1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17$
   - $\{A[B] \geq 0$ and $A[B] \geq 3\}$
     Path: $0, 1, 2, 3, 4, 5, 9, 10, 15, 16, 17$

- $\{A[B] \geq 0$ and $A[B] < 3$ and $A[B] > 0$ and $C = 0\}$
  Path: $0, 1, 2, 3, 4, 5, 9, 10, 11, 12, 14, 15, 16, 17$
- $\{A[B] \geq 0$ and $A[B] < 3$ and $(A[B] \leq 0$ or $C \neq 0)\}$, that is $\{A[B] = 0$ or $(0 \leq A[B] < 3$ and $C \neq 0)\}$
  Path: $0, 1, 2, 3, 4, 5, 9, 10, 11, 12, 13, 14, 15, 16, 17$

2. Possible preconditions are the following (you were asked to provide only one of them):

   - $|a| = 10$ and $0 \leq b \leq 9$ and $a[b] < 0$ and $a[2] \neq 0$
   - $|a| = 10$ and $0 \leq b \leq 9$ and $a[b] \geq 3$ and $a[5] \neq 0$
   - $|a| = 10$ and $0 \leq b \leq 9$ and $0 < a[b] < 3$ and $c = 0$ and $a[7] \neq 0$
   - $|a| = 10$ and $0 \leq b \leq 9$ and $(a[b] = 0$ or $(0 \leq a[b] < 3$ and $c \neq 0))$ and $a[4] \neq 0$

   Considering the tree, the conditions above ensure the execution of feasible paths to the leaves and further constrain the return value r.

   Note that path $0, 1, 2, 3, 4, 5, 9, 10, 11, 12, 14, 15, 16, 17$ (path 5) cannot lead to the desired condition; indeed, it would give the following condition:

   $$(|a| = 10 \text{ and } 0 \leq b \leq 9 \text{ and } 0 < a[b] < 3 \text{ and } c = 0 \text{ and } a[7] \neq 0)$$

   which cannot hold because $a[7]$ is equal to c, which, in turn, should be 0. Therefore, the corresponding path must return 0 to the caller.

Chapter 4

Strategies for test case identification

4.1 Brief theoretical recap

Developers and quality assurance teams traditionally rely on manual identification of test cases. However, leveraging automated techniques can significantly enhance this process. In the forthcoming examples, we use two such techniques: *concolic* execution, which combines symbolic and actual software execution, and *metamorphic* testing.

Concolic testing aims to explore as many execution paths within the software as possible, identifying test cases that satisfy the path conditions established during this exploration.

Metamorphic testing, on the other hand, begins with preexisting test cases, their expected outcomes, and known metamorphic relations between inputs and outputs. Using this information, the technique guides the identification of subsequent test cases that, based on these established relations, should produce results consistent with the original test cases. This approach not only helps in identifying new test cases but also generates appropriate oracles for the identified cases.

4.2 Triangles

Description

Consider a function with the following signature: `bool isTriangle(float a, float b, float c)`. The function receives three integers representing the lengths of three segments. The function returns true if the three segments can constitute the sides of a triangle and false otherwise.

Questions

Assume you have the following test cases and corresponding output:

- $a = 10, b = 12, c = 9$, result `true`
- $a = 10, b = 12, c = 22$, result `false`

Identify one or more metamorphic relations between inputs and outputs and some corresponding test cases.

Solution

Metamorphic relations: we can identify at least the following two relations that predicate on the sides of a triangle:

1. *Relation between the sides of a triangle:* the length of each side of a triangle is lower than the sum of the lengths of the remaining two sides.
2. *Similarity:* given two triangles, A and B, they are said to be similar if the corresponding pairs of sides proportional.

Test cases: we can exploit the metamorphic relations to identify additional test cases. For instance:

1. By exploiting the first relation:
   - $a = 12, b = 9, c = 10$, result `true`
   - $a = 12, b = 22, c = 10$, result `false`
2. By exploiting the similarity:
   - $a = 5, b = 6, c = 4.5$, result `true`
   - $a = 100, b = 120, c = 90$, result `true`

4.3 Wittgenstein and friends part 2

Description

Consider the philosophers' problem described in Section 2.4. Assume that the philosophers are represented by four processes, `P1`, ..., `P4` accessing to four shared variables, `fork1`, ..., `fork4`, which contain information about the availability of forks. In particular, if the value of a fork is true, this means that the fork is available for use by any process (philosopher). If it is false, it means that it is not available being in the hands of a specific philosopher.

At startup time, all processes are spawned and work autonomously till the end of a single thinking and eating cycle.

We have tested the software while the processes access the forks variables as follows:

1. $R^{P1}_{fork1} \rightarrow true$
2. $W^{P1}_{fork1}(false)$
3. $R^{P1}_{fork2} \rightarrow true$
4. $W^{P1}_{fork2}(false)$
5. $W^{P1}_{fork1}(true)$
6. $W^{P1}_{fork2}(true)$
7. $R^{P2}_{fork2} \rightarrow true$
8. $W^{P2}_{fork2}(false)$
9. $R^{P2}_{fork3} \rightarrow true$
10. $W^{P2}_{fork3}(false)$
11. $W^{P2}_{fork2}(true)$
12. $W^{P2}_{fork3}(true)$
13. $R^{P3}_{fork3} \rightarrow true$
14. $W^{P3}_{fork3}(false)$
15. $R^{P3}_{fork4} \rightarrow true$
16. $W^{P3}_{fork4}(false)$
17. $W^{P3}_{fork3}(true)$
18. $W^{P3}_{fork4}(true)$
19. $R^{P4}_{fork4} \rightarrow true$
20. $W^{P4}_{fork4}(false)$
21. $R^{P4}_{fork1} \rightarrow true$
22. $W^{P4}_{fork1}(false)$
23. $W^{P4}_{fork4}(true)$
24. $W^{P4}_{fork1}(true)$

R^{Py}_{forkx} indicates a read operation on $forkx$ by philosopher Py and the value on the right of the arrow represents the value that the philosopher reads from the fork. $W^{Py}_{forkx}(booleanvalue)$ represents a write operation by process Py that assigns to $forkx$ the boolean value specified in the parentheses.

In this case, all philosophers have got the forks and eaten.

Questions

What could we discover by applying metamorphic testing for concurrent systems in this case?

Solution

In the considered test case, processes have accessed the variables in a strictly sequential order. By applying metamorphic testing we can explore different test cases corresponding to enforcing different sequences of R/W operations on the shared variables. We expect that the test will be successful if, in any sequence we select, the execution will end with all philosophers having eaten after thinking, as it happens in the case above.

	Data Access Pattern	Description
1.	$R_u(l)\ W_{u'}(l)\ W_u(l)$	Value read is stale by the time an update is made in u.
2.	$R_u(l)\ W_{u'}(l)\ R_u(l)$	Two reads of the same location yield different values in u.
3.	$W_u(l)\ R_{u'}(l)\ W_u(l)$	An intermediate state is observed by u'.
4.	$W_u(l)\ W_{u'}(l)\ R_u(l)$	Value read is not the same as the one written last in u.
5.	$W_u(l)\ W_{u'}(l)\ W_u(l)$	Value written by u' is lost.
6.	$W_u(l_1)\ W_{u'}(l_1)\ W_{u'}(l_2)\ W_u(l_2)$	Memory is left in an inconsistent state.
7.	$W_u(l_1)\ W_{u'}(l_2)\ W_{u'}(l_1)\ W_u(l_2)$	same as above.
8.	$W_u(l_1)\ W_{u'}(l_2)\ W_u(l_2)\ W_{u'}(l_1)$	same as above.
9.	$W_u(l_1)\ R_{u'}(l_1)\ R_{u'}(l_2)\ W_u(l_2)$	State observed is inconsistent.
10.	$W_u(l_1)\ R_{u'}(l_2)\ R_{u'}(l_1)\ W_u(l_2)$	same as above.
11.	$R_u(l_1)\ W_{u'}(l_1)\ W_{u'}(l_2)\ R_u(l_2)$	same as above.
12.	$R_u(l_1)\ W_{u'}(l_2)\ W_{u'}(l_1)\ R_u(l_2)$	same as above.
13.	$R_u(l_1)\ W_{u'}(l_2)\ R_u(l_2)\ W_{u'}(l_1)$	same as above.
14.	$W_u(l_1)\ R_{u'}(l_2)\ W_u(l_2)\ R_{u'}(l_1)$	same as above.

Figure 4.1: Data access patterns.

To select test cases in a smart way, we can rely on the Data access patterns table shown in Figure 4.1[1]. Since in this example we have multiple variables and multiple processes to be considered, all patterns in the table are applicable. For instance, by applying pattern number 1 in our case we could enforce the following sequence of operations:

[1]Figure from Hammer et al. 2008. Dynamic detection of atomic-set-serializability violations. In Proceedings of the 30th International Conference on Software Engineering (ICSE'08). 231–240.

1. $R^{P1}_{fork1} \rightarrow true$
2. $R^{P4}_{fork1} \rightarrow true$
3. $W^{P4}_{fork1}(false)$
4. $W^{P1}_{fork1}(false)$
5. $R^{P4}_{fork4} \rightarrow true$
6. $W^{P4}_{fork4}(false)$
7. $W^{P4}_{fork4}(true)$
8. $W^{P4}_{fork1}(true)$
9. $R^{P1}_{fork2} \rightarrow true$
10. $W^{P1}_{fork2}(false)$
11. $W^{P1}_{fork1}(true)$
12. $W^{P1}_{fork2}(true)$
13. $R^{P2}_{fork2} \rightarrow true$
14. $W^{P2}_{fork2}(false)$
15. $R^{P2}_{fork3} \rightarrow true$
16. $W^{P2}_{fork3}(false)$
17. $W^{P2}_{fork2}(true)$
18. $W^{P2}_{fork3}(true)$
19. $R^{P3}_{fork3} \rightarrow true$
20. $W^{P3}_{fork3}(false)$
21. $R^{P3}_{fork4} \rightarrow true$
22. $W^{P3}_{fork4}(false)$
23. $W^{P3}_{fork3}(true)$
24. $W^{P3}_{fork4}(true)$

where operations highlighted in red, may generate problems. In this case, the issue is that `P1` at line 4 may assume to have acquired `fork1` that, instead, has been acquired by `P4` at line 3.

Another possible test case could correspond to the following sequence, which corresponds to pattern 6. In this case, $W^{P4}_{fork1}(false)$ and $W^{P3}_{fork4}(false)$ correspond to the scenario where `P4` and `P3` may think they have acquired forks 1 and 4, respectively, while these are held by other philosophers (`P1` is holding `fork1` and `P4` `fork4`).

1. $R^{P1}_{fork1} \rightarrow true$
2. $R^{P3}_{fork3} \rightarrow true$
3. $W^{P3}_{fork3}(false)$
4. $R^{P4}_{fork4} -> true$
5. $R^{P3}_{fork4} -> true$
6. $R^{P4}_{fork1} -> true$
7. $W^{P1}_{fork1}(false)$
8. $W^{P4}_{fork4}(false)$
9. $W^{P4}_{fork1}(false)$
10. $W^{P3}_{fork4}(false)$
11. $W^{P3}_{fork3}(true)$
12. $W^{P3}_{fork4}(true)$
13. $R^{P1}_{fork2} \rightarrow true$
14. $W^{P1}_{fork2}(false)$
15. $W^{P1}_{fork1}(true)$
16. $W^{P1}_{fork2}(true)$
17. $W^{P4}_{fork4}(true)$
18. $W^{P4}_{fork1}(true)$
19. $R^{P2}_{fork2} \rightarrow true$
20. $W^{P2}_{fork2}(false)$
21. $R^{P2}_{fork3} \rightarrow true$

22. $W^{P2}_{fork3}(false)$
23. $W^{P2}_{fork2}(true)$
24. $W^{P2}_{fork3}(true)$

Finally, a third case not highlighted by the data access patterns but related to the deadlock condition is the following one:

1. $R^{P1}_{fork1} \rightarrow true$
2. $W^{P1}_{fork1}(false)$
3. $R^{P2}_{fork2} \rightarrow true$
4. $W^{P2}_{fork2}(false)$
5. $R^{P3}_{fork3} \rightarrow true$
6. $W^{P3}_{fork3}(false)$
7. $R^{P4}_{fork4} \rightarrow true$
8. $W^{P4}_{fork4}(false)$
9. $R^{P1}_{fork2} -> false$
10. $W^{P1}_{fork2}(false)$
11. $R^{P2}_{fork3} -> false$
12. $W^{P2}_{fork3}(false)$
13. $R^{P3}_{fork4} -> false$
14. $W^{P3}_{fork4}(false)$
15. $R^{P4}_{fork1} -> false$
16. $W^{P4}_{fork1}(false)$
17. $W^{P1}_{fork1}(true)$
18. $W^{P1}_{fork2}(true)$
19. $W^{P2}_{fork2}(true)$
20. $W^{P2}_{fork3}(true)$
21. $W^{P3}_{fork3}(true)$

22. $W^{P3}_{fork4}(true)$
23. $W^{P4}_{fork4}(true)$
24. $W^{P4}_{fork1}(true)$

4.4 Concolic Execution

Description

```
1  void bar(int x, int y) {
2      // Program state?
3      if (x > y) {
4          x = 3;
5      } else {
6          y = 3;
7      }
8      // Program state?
9      y = y * 2;
10     // Program state?
11     x = baz(x, y);
12     // Program state?
13     if (x < 0) {
14         y = y - 1;
15     }
16     // Program state?
17     assert(x + y == 0);
18 }
19 int baz(int a, int b) {
20     return a - b;
21 }
```

Questions

1. Run a concolic execution with seed inputs $x = 1$ and $y = 2$, and write down the program state while executing line numbers 2,8,10,12,16.

2. Since concolic execution is a test generation technique, our next goal is to generate a new set of inputs that leads the program down a different path than in the previous execution. For that, take the path condition from the previous execution and negate the conjunct that corresponds to the branch at line 13 in the program.

   This results in the following path condition:

   $$\neg(x_0 > y_0) \wedge \neg(x_0 - 6 < 0)$$

   Please solve this path condition for x_0 and y_0 to get test inputs for the program that take a new path.

3. Execute a new concolic execution, taking the new concrete values forx and y that you generated in the previous subtask as concrete inputs and write down the program state while executing line numbers 2,8,10,12,16. At the end of this second execution, does the assertion in line 17 fail?

4. Concolic execution can successively generate more inputs for x and y until ultimately each path through the program has been taken once. Please list two more solutions for x and y that each cover a new path through the program. For each pair of inputs, also list the final path condition when the program executes on these inputs.

Solutions

1. Solution to the first question:

At Line	Concrete Execution	Symbolic Execution	Path condition
2	$x = 1$, $y = 2$	$x = x_0$, $y = y_0$	N/A
8	$x = 1$, $y = 3$	$x = x_0$, $y = 3$	$\neg(x_0 > y_0)$
10	$x = 1$, $y = 6$	$x = x_0$, $y = 6$	$\neg(x_0 > y_0)$
12	$x = -5$, $y = 6$	$x = x_0 - 6$, $y = 6$	$\neg(x_0 > y_0)$
16	$x = -5$, $y = 5$	$x = x_0 - 6$, $y = 5$	$\neg(x_0 > y_0) \wedge (x_0 - 6 < 0)$

2. x0 = 6, y0 = 6.

3. Solution to the third question:

At Line	Concrete Execution	Symbolic Execution	Path condition
2	$x = 6$, $y = 6$	$x = x_0$, $y = y_0$	N/A
8	$x = 6$, $y = 3$	$x = x_0$, $y = 3$	$\neg(x_0 > y_0)$
10	$x = 6$, $y = 6$	$x = x_0$, $y = 6$	$\neg(x_0 > y_0)$
12	$x = 0$, $y = 6$	$x = x_0 - 6$, $y = 6$	$\neg(x_0 > y_0)$
16	$x = 0$, $y = 6$	$x = x_0 - 6$, $y = 6$	$\neg(x_0 > y_0) \wedge \neg(x_0 - 6 < 0)$

   At the end of this second execution, the assertion in line 17 fails.

4. (a) $x = 6$, $y = 5$, path condition: $(x_0 > y_0) \wedge (3 - 2y_0 < 0)$
   (b) $x = 2$, $y = 1$, path condition: $(x_0 > y_0) \wedge \neg(3 - 2y_0 < 0)$

Chapter 5

Integration Testing Plan Definition

5.1 Brief theoretical recap

Integration testing aims to verify that the components of a software system work together as expected. The primary focus is on ensuring that interfaces between components are used and provided correctly. In addition to systematically identifying test cases to cover all expected interactions between components, another critical challenge is determining the order in which components will be integrated and tested.

Coordinating the integration plan with the testing plan is essential. For example, if the integration plan specifies that components A and B should be integrated first, testing the integration of A and C before A and B can create delays and may hide integration errors between A and B, negatively impacting other testing activities. Therefore, creating a unified integration and testing plan is typically recommended.

The strategies for defining the integration and testing order depend on the software, the development team's skills, and the project schedule. Common strategies rely on the system's structure or the roles of specific components. Structural approaches include bottom-up and top-down methods. These involve creating a dependency tree of the system's architecture and using it to schedule activities. Starting with components that have no dependencies and moving upwards is a bottom-up approach, whereas starting from the top and moving down is a top-down approach.

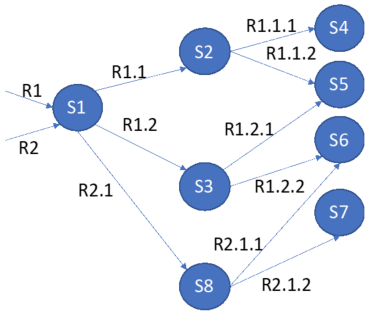
Two other approaches focus on the roles of specific components: *critical modules* and *thread-based* strategies. The former strategy centers the integration process around the most important or highest-risk component. The initial focus is on integrating this critical component with the ones around it. The latter strategy emphasizes the incremental delivery of functionalities, involving

the development, integration, and testing of parts of multiple components for each functionality. The advantage of this approach is the early release of software, albeit with limited functionality, allowing for early feedback and iterative improvements.

5.2 Microservices part 2

Description

As stated in Exercise 1.5: consider the microservice-based architecture shown in Figure 5.2. The architecture is organized into eight stateless microservices collaborating 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	Avail
S1	0.99
S2	0.9
S3	0.9
S4	0.95
S5	0.999
S6	0.99
S7	0.99
S8	0.95

Figure 5.1: Microservice-based architecture.

Questions

1. Define an integration and test plan for the architecture, motivating your choice.
2. Assume that you are asked to test whether the availability improvement theoretically obtained with the duplication of services (see the exercise proposed in Section 1.5) is actually achieved. How would you proceed to perform such a test?

Solution

1. We could use a bottom-up approach mixed with a thread one. More specifically, since we know the dependencies between services with refer-

ence to the two requests the system is able to fulfill ($R1$ and $R2$), we could focus on one of the two at a time.

Focusing on $R1$, we can see that we can focus first on the integration of $S2$ with $S4$ and $S5$. To test such integration, we would need a driver that issues request $R1.1$. This implies that, for the purpose of this integration, the part of $S5$ in charge of responding to request $R1.2.1$ is not needed and, therefore, can be postponed. After completing this step, we can focus on the integration and testing of $S3$ with $S5$ and $S6$. In this case, $S5$ must be able to respond to request $R.1.2.1$ and a driver issuing $R1.2$ would be needed. Finally, we can integrate $S1$ with $S2$ and $S3$.

We use a similar approach to test request $R2$, and in particular, the integration of $S8$ with $S6$ and $S7$ and then of $S1$ with $S8$.

2. An availability test should be performed in an environment as close as possible to the operational one. We need to execute the system for a significant time interval, ensuring that all features offered by the system are actually executed, and we need to compute MTTF and MTTR as availability can be measured as $MTTF/(MTTF + MTTR)$. Notice that in most cases, the system repair is performed by human beings. This implies that the measurement of MTTR concerns the ability of the repairing team to react quickly and repair the problem.

   Said this, in our case, in order to measure the improvement obtained through the duplicated configuration, we need, first, to execute, for a significant amount of time, the whole system without any replication. In this execution, we will make sure R1 and R2 will both be called multiple times also by users working in parallel.

   After having computed MTTF and MTTR for the system in this timeframe, we will perform the same experiment using the duplicated components. In this case, we can also purposefully take down one of the replicas, to make sure that indeed replication overall achieves the desired availability.

5.3 SmartLightingKit

Description

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 authorized users to control the lights (either locally or remotely). Local control is achieved through terminals installed in the rooms (one terminal per room). Anyone in the room can control the corresponding lights through the room terminal. Remote control is realized through a smartphone application or through a central terminal installed in the control room of the building.

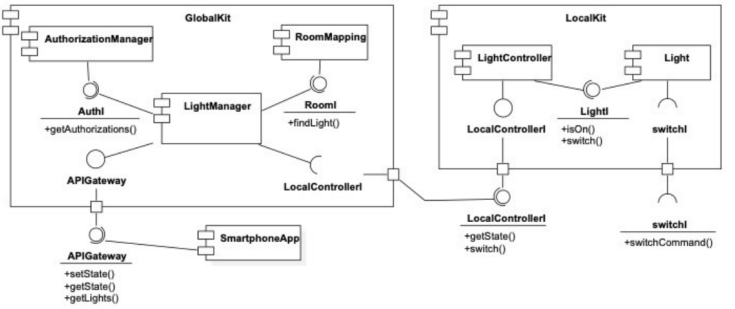


Figure 5.2: SmartLightingKit component diagram.

While controlling a room's lights, 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. Administrators can also change the authorizations assigned to regular users. These last ones can control the lights of specific rooms. When an administrator installs a new light in a room, he/she triggers a pairing process between the light and the terminal located in that room. After pairing, the system can manage the light.

The first version of SmartLightingKit focuses exclusively on a subset of the problem: switching on and off lights either from the local terminal or from remote. It does not support the definition of routines or the possibility of setting actions such as switching on/off lights when events happen, installing new lights, and modifying authorizations assigned to regular users. The architecture of this first version of the system is shown in Figure 5.2.

Questions

1. Define a suitable set of unit test cases for the `Light` and `RoomMapping` components.

2. Define an integration testing strategy.

3. In the next version of the system, the development team is focusing on the possibility of creating routines and installing new lights. The use case diagram in Figure 5.3 is defined in the requirement analysis phase for this second iteration. Define a set of system test cases for the new version

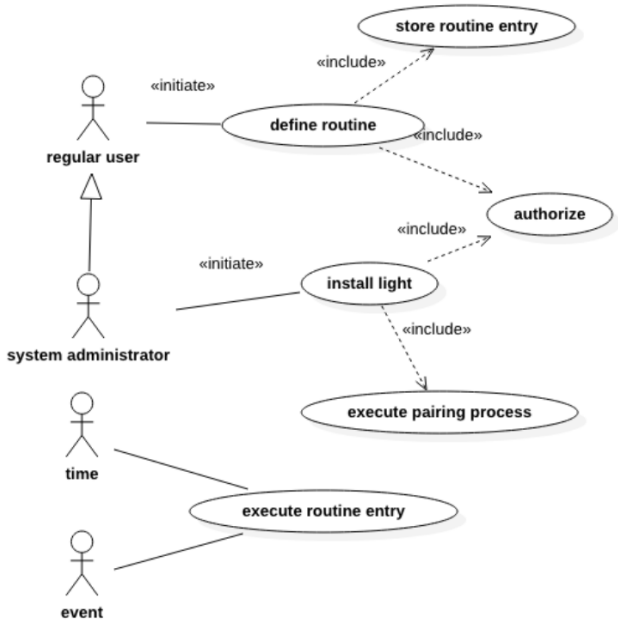


Figure 5.3: SmartLightingKit use case diagram

of SmartLightingKit focusing on the features described in the use case diagram.

Solution

1. In order to define the unit test cases, we need to understand how the component under testing behaves. Since we do not have information about the code that implements `Light` and `RoomMapping`, we will adopt a black-box approach for the selection of test cases.

   In general, a test case includes the values to be assigned to the inputs, the initial state of the software under test, and the expected output. Therefore, we will represent test cases through triples of the form:

   $$\langle \text{input values, initial state, expected output} \rangle$$

   Let us focus on `Light`. The component exposes the operations `isOn` and `switch`. The architectural diagram does not provide details about the parameters of these operations. Assuming that `Light` represents an individual light in a room and that we are going to have multiple instances of `Light`, one per physical light, we can conclude that reasonable signatures for these two operations are the following:

- `boolean isOn()`: returns true if the light is on, false otherwise.
- `void switch()`: changes the status of the light. If on, it will be turned off, and vice versa. This operation is accomplished through the operation offered by the `switchI` interface. This implies that we must implement a stub of this interface emulating the execution of the `switchCommand` operation.

The unit test cases for `isOn` can be the following:

- $\langle$NA, light is on, true$\rangle$, with NA = not available
- $\langle$NA, light is off, false$\rangle$

The unit test cases for `switch` can be:

$\langle$NA, light is on,
the stub receives the `switchCommand()` request,
a call to `isOn` will return false$\rangle$

$\langle$NA, light is off,
the stub receives the `switchCommand()` request,
a call to `isOn` will return true$\rangle$

Let us consider `RoomMapping`. The component stores the association between rooms and lights. It offers the operation `findLight`. We can interpret the operation in at least the following two different ways which lead to two different signatures:

- We could assume that, given a room, the operation returns all lights in the room. In this case, the signature could be the following: `lightList findLight(int roomId)`
- We could assume that the operation returns the room, or the handler to the corresponding local controller, where the light with a certain Id is located. In this case, the signature could be the following: `LocalControllerI findLight(int lightId)`

The unit test cases for the signature `lightList findLight(int roomId)` could be the following:

- $\langle 1, [\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,4\rangle, \langle 2,5\rangle, \langle 3,6\rangle, \langle 5,7\rangle, \langle 5,8\rangle], [1,2,3]\rangle$
- $\langle 4, [\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,4\rangle, \langle 2,5\rangle, \langle 3,6\rangle, \langle 5,7\rangle, \langle 5,8\rangle], []\rangle$
- $\langle 5, [\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,4\rangle, \langle 2,5\rangle, \langle 3,6\rangle, \langle 5,7\rangle, \langle 5,8\rangle], [7,8]\rangle$
- $\langle 3, [\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,4\rangle, \langle 2,5\rangle, \langle 3,6\rangle, \langle 5,7\rangle, \langle 5,8\rangle], [3,6]\rangle$

- $\langle 3, [], [] \rangle$

Note that $[\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,4\rangle, \langle 2,5\rangle, \langle 3,6\rangle, \langle 5,7\rangle, \langle 5,8\rangle]$ is a representation of the room-light mapping that we assume to be stored in the `RoomMapping` state. The first number in each pair represents a room number, and the second one is the light number. Both room numbers and light numbers are unique.

The second test case is verifying that the operation works well also in the case a room number is not present in the `RoomMapping` state. The first and the third test cases check that the operation behaves correctly when we consider the first and the last room in the `RoomMapping` data structure. The fourth test case is checking that the operation behaves correctly when the room number is in an intermediate position in the data structure. The fifth test case is checking the case when the `RoomMapping` state is empty. In this case, we want the operation to return an empty list independently of the room number passed as a parameter. The test cases for the operation with signature `LocalControllerI findLight(int lightId)` can be derived adopting the same line of reasoning: consider a non-empty state and check the case in which `lightId` is or does not belong to the state, consider an empty state and any `lightId`.

2. The integration testing strategy we adopt depends on the integration sequence we plan to execute. We have mentioned the bottom-up, top-down, thread-based, and critical module strategies.

   Having in this particular case three different subsystems (`SmartphoneApp`, `GlobalKit` and `LocalKit`), we could even adopt a strategy for integrating the components within a subsystem and a different one for integrating the three subsystems.

   Let us focus on each subsystem first.

   Within `LocalKit` is it quite natural to adopt a bottom-up strategy. We need a stub that implements `switchI` interface unless we can have the actual implementation available at integration time. In this last case, we will focus first on testing Light with the actual `switchI` implementation and then we will move to test the integration between `Light` and `LightController`. If the switchI implementation is not available at this stage, we will have to rely on the corresponding stub in this phase and postpone the actual integration with the switchI to a later stage, but, in any case, not after the system test. In any case, the `LocalKit` test will require a driver that uses the `LocalControllerI`.

   Focusing on the `GlobalKit` subsystem, we will need to have a stub of `LocalKit` and a driver using the `APIGateway` interface. Also in this case, we can start in a bottom-up way from `AuthorizationManager` and its integration with `LightManager`, using a stub for `RoomMapping` and then integrating it with the others.

We have no details on the internals of `SmartphoneApp`. Thus, we cannot make precise plans about its integration.

For what concerns the integration of the three subsystems, if we wish to offer our users early demos of `SmartLightingKit`, we could adopt a top-down approach. This implies that we could, as a first step, focus on the integration between `SmartphoneApp` and `GlobalKit` and then replace `LocalKit` stub with its actual implementation. Note that `LocalKit` is expected to be available in multiple instances, one per room. Therefore, different tests with an increasing number of `LocalKit` components will have to be developed.

The actual schedule of our integration and test plan depends on the organization of our team. If we assume to have independent teams focusing on the three subsystems, we could imagine to adopt the following schedule:

- Time interval t1: intra-subsystems integration and test activities occurring independently and in parallel
- Time interval t2: `SmartphoneApp` and `GlobalKit` integration and test. Further tests within `LocalKit` can take place at this stage as well. For instance, we can take the opportunity to focus on the integration between `LocalKit` and the actual `switchI` implementation.
- Time interval t3: Integration and test of `SmartphoneApp`+`GlobalKit` and one instance of `LocalKit` already integrated with the actual `switchI` implementation. When tests pass, we can run tests again with multiple instances of `LocalKit`.

3. In general, test cases are defined based on the requirement analysis documentation. In this particular case, we focus on the use case diagram provided by the exercise text and on the description of the system under analysis.

   We infer from the use case diagram that we have three main use cases. We define test cases for each of them. The description is informal, but all test cases include the input values, initial state, and expected output.

   Install light: It allows the administrator to install a new light in a certain room. The installation can take place if the user has the proper authorization as an administrator. As part of the installation procedure, the pairing process must be executed. Meaningful test cases can be the following:

   - Administrator 1 installs light 1 in room 1. The administrator has the right permissions, the room exists in the system status, light 1 is not in the system yet, and the administrator has activated the pairing button on the physical light. `SmartLightingKit` activates the pairing on its side, the pairing is successful, and the pair ⟨room 1, light 1⟩ is stored in the system.

- Administrator 1 installs light 1 in room 1. The administrator has the right permissions, the room exists in the system status, light 1 is not in the system yet, and the administrator has not activated the pairing button on the physical light. `SmartLightingKit` activates the pairing on its side, but the pairing is not successful. The system returns a pairing error to the administrator.
- Administrator 1 installs light 1 in room 1. The administrator has the right permissions, and the room does not exist in the system status. The system returns a non-existing room error to the administrator.
- Administrator 1 installs light 1 in room 1. The administrator does not have the right permissions. The system returns a permission error to the administrator.
- Administrator 1 installs light 1 in room 1. The administrator has the right permissions, the room exists in the system status, and light 1 exists already in the system. The system returns a light already existing error to the administrator.

Define routine: the regular user defines a routine through a text editor. He/she includes in the routine actions on specific rooms and/or lights and corresponding triggering events. A pair ⟨event, action⟩ is called an entry. The user then saves the routine. Possible test cases are the following:

- User 1 defines routine 1 that includes entries in rooms 1 and 2. User 1 has the right permissions on rooms 1 and 2. All defined actions and events are possible ones. `SmartLightingKit` inserts all entries in the list of entries.
- User 1 defines routine 1 that includes actions in room 1 and lights 1 and 2 in room 2. User 1 has the right permissions on room 1 and room 2. All defined actions and events are possible ones. `SmartLightingKit` inserts all entries in the list of entries.
- User 1 defines routine 1 that includes actions in rooms 1 and 2. User 1 has the right permissions in room 1 but not in room 2. `SmartLightingKit` returns to the user a permission error in room 1 and stops the procedure.
- User 1 defines routine 1 that includes actions in rooms 1 and 2. User 1 has the right permissions on rooms 1 and 2. One of the defined actions is not recognized by the system. `SmartLightingKit` returns an unknown action error to the user and stops the procedure.
- User 1 defines routine 1 that includes actions in rooms 1 and 2. User 1 has the right permissions on rooms 1 and 2. One of the defined events is not recognized by the system. `SmartLightingKit` returns an unknown event error to the user and stops the procedure.
- User 1 defines routine 1 that is empty. `SmartLightingKit` returns an empty routine error to the user and stops the procedure.

Execute routine entry: The execution of a routine entry is caused by the time passing or the occurrence of a specific event. Possible test cases are the following:

- Time 8.00 arrives. The list of routine entries contains the following pair ⟨at 8.00, switch off lights in room 1⟩. `SmartLightingKit` executes the corresponding action. Lights in room 1 get switched off.
- Time 8.00 arrives. The list of routine entries contains the following two pairs ⟨at 8.00, switch off lights in room 1⟩ and <between 8.00 and 16.00, and switch on lights 3 and 4 in room 2>. `SmartLightingKit` executes the corresponding action. Lights 3 and 4 in room 1 get switched on.
- Time 8.00 arrives. The list of routine entries does not contain any entries associated with this time. `SmartLightingKit` does not execute any action.
- Someone enters in room 3. The list of routine entries contains the following pair ⟨on entering room x, switch on lights in room x⟩. `SmartLightingKit` executes the corresponding action. Lights in room 3 get switched on.

5.4 PubCoReader

Description

PubCoReader is a software system conceived to be used in courses with multiple instructors. It allows groups of students to read (or watch – for simplicity, in the following the two terms are used as synonyms) and annotate publications (books, papers, reports, videos, ...) collaboratively. PubCoReader should support the creation of multiple groups of readers that read the same or different publications independently from the other. Group members collaborate online.

For instance, at a certain point in time, we can observe the following scenario, where three groups of students are using the system.

Group 1 is studying the book titled "Software Architectures" and is currently annotating Chapter 2 by complementing the book content with a picture from the lecture slides and some textual notes. Each group member sees the same book page and the identity of the group mate who has added an annotation to the book, as well as the exact position where the annotation has been added.

Group 2 is watching the video of the latest lecture in the course. The group members are discussing through the chat implemented as part of PubCoReader a point that is not clear to any of them and add the following annotation to the specific point under discussion "`@professor`: please help us understand this point".

Group 3 has run the "translate to audio" feature in Chapter 3 of the "Software Architectures" book and is currently listening to the generated audio. Each time a group member stops the audio and inserts an annotation, this is associated with the corresponding part of the text in the book.

Two professors are monitoring the students' activities. One of them notices the `@professor` message and temporarily joins the corresponding group to discuss the misunderstanding with them.

The development team has defined the following high-level architecture for PubCoReader.

The interfaces offered by `StudentApp` and `ProfessorApp` capture the functions of the two applications and are made available to the two types of users of PubCoReader: students and professors.

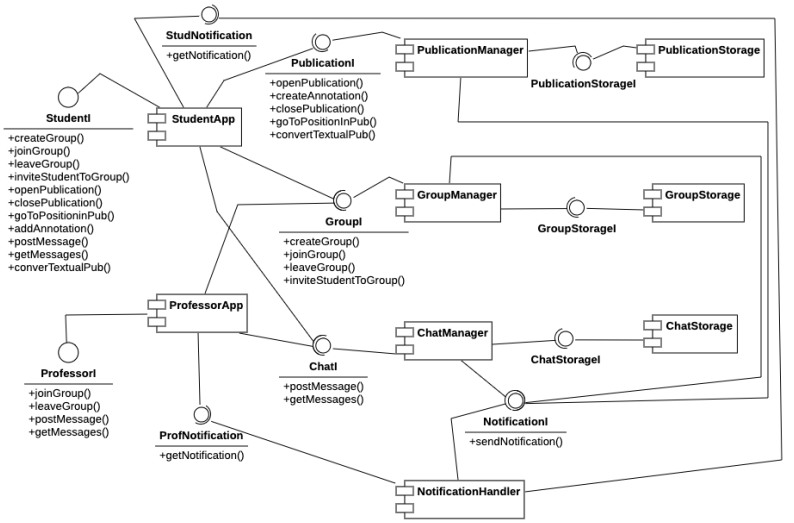


Figure 5.4: Component Diagram

Questions

1. Given this architecture and the operations defined by the components' interfaces, define an integration testing plan and explain why you think it is appropriate for the case at hand.

Solutions

1. The integration testing plan we adopt depends on the integration sequence we plan to execute. In general, the plan can be based on one or a combination of the following strategies: bottom-up, top-down, thread-based, and critical modules. In this particular case, applying a bottom-up strategy could be the simplest approach. However, we need to consider that, in this case, the usage relationships lead to a cycle since `NotificationManager` is using `ProfessorApp` and `StudentApp` that, for all the other features, are actually at the top of the usage relationship (see the usage relationships representation in Figure 5.5).

   We could proceed by excluding the implementation of the component `NotificationHandler` from the initial integration iteration, which could be replaced by a stub until the last step. More specifically, we could start with the integration and testing of the following elements:

   - `PublicationManager` and `PublicationStorage`, using a stub for `NotificationHandler` and a driver for `PublicationManager`

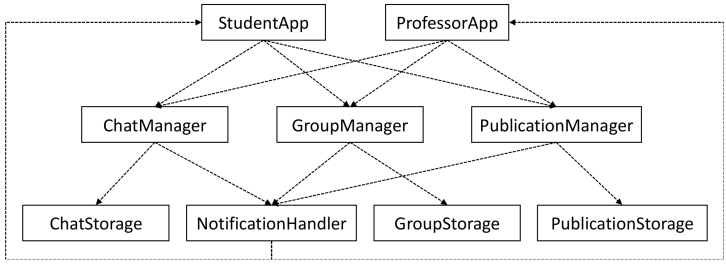


Figure 5.5: Use relationship diagram

- `GroupManager` and `GroupStorage`, using a stub for the component `NotificationHandler` and a driver for `GroupManager`
- `ChatManager` and `ChatStorage`, using a stub for the component `NotificationHandler` and a driver for `ChatManager`

These three integration steps could be performed even in parallel if we can organize the development team in three disjoint subteams.

Then, we could proceed as follows:

- Integration of `StudentApp` with `PublicationManager`, using a driver to replace the `NotificationHandler`
- Integration of `StudentApp` with `GroupManager`, using a driver to replace the `NotificationHandler`
- Integration of `StudentApp` with `ChatManager`, using a driver to replace the `NotificationHandler`

At this point, we could focus on the integration between the two components `NotificationHandler` and the `StudentApp` and, finally, we could proceed with the integration and testing of the components `ProfessorApp` and `PublicationManager`, `GroupManager`, and `ChatManager`, respectively, followed by the integration between the `NotificationHandler` and the `ProfessorApp`.