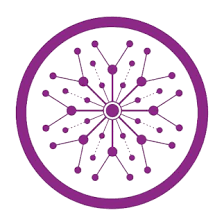
****

**Software Requirements Specification (SRS)**

**Version 1.0**

**FORMAL METHODS IN SOFTWARE ENGINEERING**

**Instructor: Sir Muneeb Ahmed**

**Hospital Appointment**

**Booking System**

**Prepared by:**

**Hifza Khalid (SU92-BSSEM-F22-202)**

**Section:BSSE-6D  
Submission Date: April 28,2025.  
Department: Software Engineering**

**Superior University Gold Campus, Lahore**

**Abstract**  
The **Hospital Appointment Booking System** is developed to streamline the process of booking, updating, viewing, and canceling appointments in hospitals. It addresses the problems of long queues and mismanagement by allowing patients to book appointments remotely and enabling hospital staff to manage schedules efficiently. The system ensures data accuracy, reduces patient waiting time, and improves overall healthcare service quality.

# **Acknowledgement**

I would like to extend my sincere thanks to [Sir Muneeb Ahmed](https://www.linkedin.com/in/munib-ahmad-08a4a4250/) for his guidance in **Z Language** for requirement gathering, and to [Sir Tayyab Khushi](https://www.linkedin.com/in/hafiz-muhammad-tayyab-khushi-0678b416b/)  for his support with the **Python** implementation of the project.

I am also grateful to my friends who participated in the Google form for requirement gathering and to the faculty at [**Ghurki Trust Teaching Hospital**](https://www.ghurkitrust.org.pk/about-us/)for their conference on **April 26, 2025**, which was crucial for understanding the current hospital system.

Lastly, I sincerely thank my family for their unwavering support, motivation, and valuable insights into **pharmacy** and **hospital systems**, which contributed to the successful completion of this project.

**Table of Contents**

[**Acknowledgement** 3](#_Toc196678877)

[**1. Introduction** 4](#_Toc196678878)

[**1.1 The Hospital Appointment Booking System** 4](#_Toc196678879)

[**1.1.2 State Space** 5](#_Toc196678880)

[**1.1.3 Invariant Explanation** 5](#_Toc196678881)

[**1.2 Operations on the Hospital Appointment Book** 6](#_Toc196678882)

[**1.2.1 Operation: AddAppointment** 6](#_Toc196678883)

[**1.2.2 Operation: FindAppointment** 7](#_Toc196678884)

[**1.2.3 Operation: UpdateAppointment** 7](#_Toc196678885)

[**1.2.4 Operation: ViewAppointments** 7](#_Toc196678886)

[**1.2.5 Operation: CancelAppointment** 8](#_Toc196678887)

[**1.2.6 Operation: Initial State** 8](#_Toc196678888)

[**1.3 Strengthening the Specification** 8](#_Toc196678889)

[**1.3.1 Success Schema** 8](#_Toc196678890)

[**1.3.2 Robust AddAppointment Operation** 9](#_Toc196678891)

[**1.3.3 Robust FindAppointment Operation** 9](#_Toc196678892)

[**1.3.6 Robust ViewAppointments Operation** 10](#_Toc196678893)

[**1.4 Final Specification Summary** 10](#_Toc196678894)

[**2. Appendices** 10](#_Toc196678895)

[**Appendix A: Basic Definitions** 10](#_Toc196678896)

[**Appendix B: Schema Symbols** 10](#_Toc196678897)

[**Appendix C: Report Status Definitions** 10](#_Toc196678898)

[**Appendix D: Example of Initial System State** 10](#_Toc196678899)

[**Appendix E: GUI Implementation Snapshot (Python + Tkinter)** 11](#_Toc196678900)

[**3. References** 11](#_Toc196678901)

[**3.1. Project Links and Additional Resources** 11](#_Toc196678902)

# **1. Introduction**

## **1.1 The Hospital Appointment Booking System**

The best way to see how the ideas of formal specification work is through a simple but realistic example. For this purpose, I have chosen a system that is frequently encountered in healthcare settings — a **Hospital Appointment Booking System**. It is a system so elementary that it is often implemented manually by a receptionist using a register or spreadsheet.

This system maintains a record of which patients have appointments and at what times, and ensures that patients are not double-booked or assigned unavailable time slots.

**1.1.1 Basic Types**

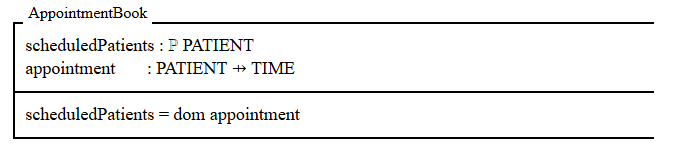
In our account of the system, we shall need to deal with the identities of patients and appointment times. At this level of abstraction, we do not need to specify the structure of these types. Hence, we introduce them as global data types:

**[PATIENT, TIME]**

This declaration allows us to refer to these sets without saying what kind of objects they contain. We only assume that elements of **PATIENT** and **TIME** can be compared and used in mappings.

## **1.1.2 State Space**

The state of the system is described using a Z schema, representing the current appointment bookings.



* **Description:**

This schema consists of:

* Above the dividing line:
* **scheduledPatients**: A set containing the patients who have appointments.
* **appointment:** A partial function mapping patients to their appointment times.
* Below the dividing line (**Predicate**):
* An invariant asserting that **scheduledPatients** is equal to the domain of the **appointment** function. In other words, every patient who is scheduled appears in the domain of **appointment.**

### **1.1.3 Invariant Explanation**

In this specification, the variable **scheduledPatients** is derivable from the **appointment** function. That is:

**scheduledPatients = dom appointment**

This relationship is an invariant of the system — it holds in every valid state and is preserved by all system operations.

Although **scheduledPatients** could be omitted (as it's derived), including it explicitly helps enhance readability and clarify intent. It also aligns with best practices in formal modeling by naming important concepts, even if they are not directly represented in the final implementation.

#### **1.1.4 Example of Appointment State**

One possible state of the system has three people in the set **scheduledPatients,** with their appointments recorded by the function **appointment:**

**scheduledPatients = { Hira, Sana, Zara }**

**appointment = {**

**Hira ↦ 10:00AM,**

**Sana ↦ 03:00PM,**

**Zara ↦ 06:00PM**

**}**

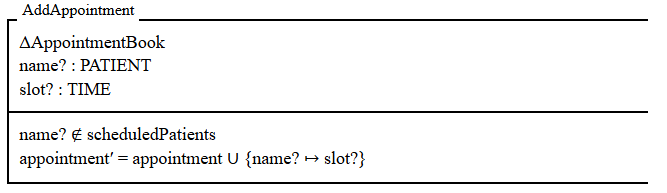
The invariant is satisfied, because **appointment** records a slot for exactly the three names in **scheduledPatients.**

Notice that in this description of the state space of the system, we have not been forced to place a limit on the number of appointments recorded in the appointment book, nor to say that the entries will be stored in a particular order. We have also avoided making a premature decision about the format of names and slots. On the other hand, we have concisely captured the information that each person can have only one appointment, because the variable **appointment** is a function, and that two people can share the same appointment slot as in our example.

## **1.2 Operations on the Hospital Appointment Book**

### **1.2.1 Operation: AddAppointment**

We now start on some operations on the system. The first is to add a new appointment:



#### **Explanation:**

The declaration **∆AppointmentBook** alerts us to the fact that the schema is describing a state change: it introduces four variables **scheduledPatients**, **appointment,** **scheduledPatients′** and **appointment′**. The first two are observations of the state before the change, and the last two are observations of the state after the change. Each pair of variables is implicitly constrained to satisfy the invariant, so it must hold both before and after the operation.

Next come the declarations of the two inputs to the operation. By convention, the names of inputs end in a question mark.

The part of the schema below the line first of all gives a pre-condition for the success of the operation: the name to be added must not already be one of those **scheduledPatients**. This is reasonable, since each person can only have one appointment.

If the pre-condition is satisfied, however, the second line says that the **appointment** function is extended to map the new name to the given slot. We expect that the set of **scheduledPatients** will be augmented with the new name:

**scheduledPatients′ = scheduledPatients ∪ {name?}**

In fact we can prove this from the specification of **AddAppointment**, using the invariants on the state before and after the operation:

**scheduledPatients′ = dom appointment′ [invariant after]**

**= dom(appointment ∪ {name? ↦ slot?}) [spec. of AddAppointment]**

**= dom appointment ∪ dom {name? ↦ slot?} [fact about 'dom']**

**= dom appointment ∪ {name?} [fact about 'dom']**

**= scheduledPatients ∪ {name?}. [invariant before]**

Stating and proving properties like this one is a good way of making sure the specification is accurate; reasoning from the specification allows us to explore the behavior of the system without going to the trouble and expense of implementing it.

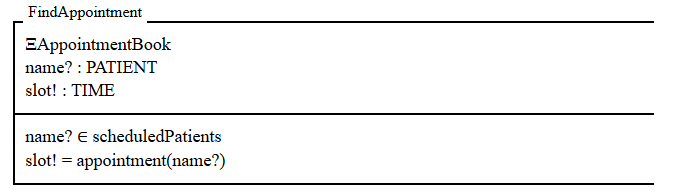
The two facts about **dom** used in this proof are examples of the laws obeyed by mathematical data types:

**dom(f ∪ g) = (dom f) ∪ (dom g)**

**dom{a ↦ b} = {a}**

### **1.2.2 Operation: FindAppointment**

Another operation might be to find the appointment of a person already in the system. Again we describe the operation with a schema:



**Explanation:**

This schema illustrates two new notations:

* The declaration **ΞAppointmentBook** indicates that this is an operation in which the state does not change: the values **scheduledPatients′** and **appointment′** of the observations after the operation are equal to their values before.

Including **ΞAppointmentBook** above the line has the same effect as including **∆AppointmentBook** above the line and the two equations:

**scheduledPatients′ = scheduledPatients**

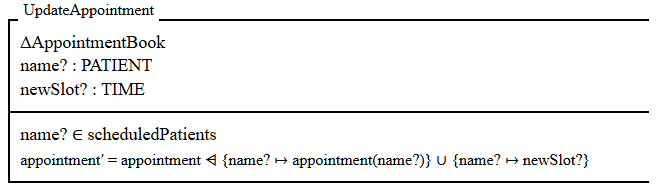
**appointment′ = appointment**

below it.

* The other notation is the use of a name ending in an exclamation mark for an output: the **FindAppointment** operation takes a name as input and yields the corresponding appointment as output. The pre-condition for success of the operation is that the name must be one of the **scheduledPatients.**

### **1.2.3 Operation: UpdateAppointment**

This operation modifies the appointment time for an existing patient. It ensures that the patient is already scheduled, and it updates the corresponding appointment slot.

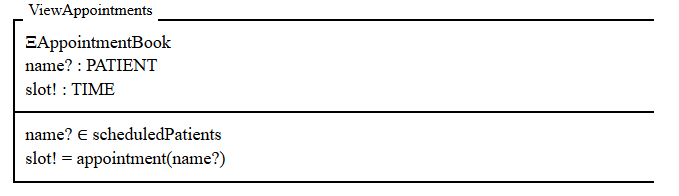
**Explanation:**

* **∆AppointmentBook**: The state will change.
* **name? ∈ scheduledPatients:** Ensures that the patient is already in the system.
* **appointment′ = appointment ⩤ {name? ↦ appointment(name?)} ∪ {name? ↦ newSlot?:** The previous appointment for **name?** is removed (using domain subtraction), and the new appointment time is added.

This operation ensures that the patient still has an appointment, but their time slot is updated.

### **1.2.4 Operation: ViewAppointments**

This operation allows us to view the appointment details for a specific patient. It retrieves the appointment slot for a given patient.

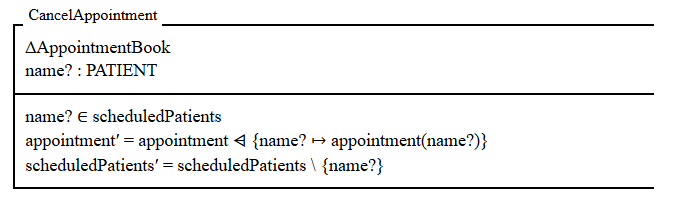
**Explanation:**

* **ΞAppointmentBook:** The state will not change.
* **name? ∈ scheduledPatients:** Ensures the patient exists in the appointment book.
* **slot! = appointment(name?):** Outputs the appointment slot for the given patient.

This operation is used to retrieve the appointment of a patient by their name.

### **1.2.5 Operation: CancelAppointment**

This operation cancels a patient's appointment, removing their entry from the system.

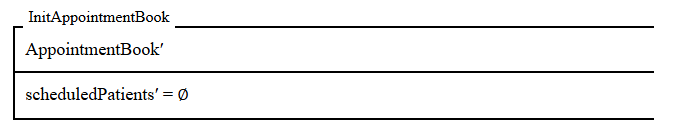
**Explanation:**

* **∆AppointmentBook:** The state will change.
* **name? ∈ scheduledPatients:** Ensures the patient is already in the appointment book.
* **appointment′ = appointment ⩤ {name? ↦ appointment(name?)}:** Removes the patient's appointment from the system (domain subtraction).
* **scheduledPatients′ = scheduledPatients \ {name?}:** Removes the patient from the list of scheduled patients.

This operation removes a patient's appointment from the system entirely, ensuring they no longer have a scheduled appointment.

## **1.2.6 Operation: Initial State**

To finish the specification, we must say what state the system is in when it is first started. This is the initial state of the system, and it also is specified by a schema:



This schema describes an appointment book in which the set **scheduledPatients** is empty: in consequence, the function **appointment** is empty too.

## **1.3 Strengthening the Specification**

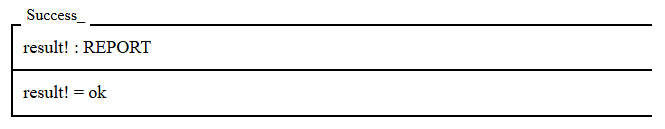
A correct implementation of our specification will faithfully record appointments and display them, so long as there are no mistakes in the input. But the specification has a serious flaw: as soon as the user tries to add an appointment for someone already known to the system, or tries to find the appointment of someone not known, it says nothing about what happens next.

We shall add an extra output **result!** to each operation on the system. When an operation is successful, this output will take the value `ok`, but it may take the other values **already\_scheduled** and **not\_scheduled** when an error is detected. The following free type definition defines **REPORT** to be a set containing exactly these three values:

**REPORT ::= ok | already\_scheduled | not\_scheduled**

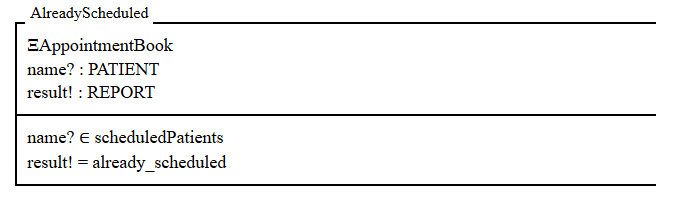
### **1.3.1 Success Schema**

We can define a schema **Success** which just specifies that the result should be `**ok**`, without saying how the state changes:



### **1.3.2 Robust AddAppointment Operation**

For the error that might be detected in the input, we define a schema which describes the conditions under which the error occurs and specifies that the appropriate report is produced. Here is a schema which specifies that the report **already\_scheduled** should be produced when the input **name?** is already a member of **scheduledPatients:**



The declaration **ΞAppointmentBook** specifies that if the error occurs, the state of the system should not change.

We can combine this description with the previous one to give a specification for a robust version of **AddAppointment:**

**RAddAppointment ≙ (AddAppointment ∧ Success) ∨ AlreadyScheduled**

This definition introduces a new schema called **RAddAppointment**, obtained by combining the three schemas on the right-hand side. The operation **RAddAppointment** must terminate whatever its input. If the input **name?** is already scheduled, the state of the system does not change, and the result **already\_scheduled** is returned; otherwise, the new appointment is added to the database as described by **AddAppointment,** and the result **ok** is returned.

### **1.3.3 Robust FindAppointment Operation**

A robust version of the **FindAppointment** operation must be able to report if the input name is not scheduled:



The robust operation either behaves as described by **FindAppointment** and reports success, or reports that the name was not scheduled:

**RFindAppointment ≙ (FindAppointment ∧ Success) ∨ NotScheduled**

**1.3.4 Robust UpdateAppointment Operation**

Similarly, we can define robust versions of the other operations. For **UpdateAppointment:**

**RUpdateAppointment ≙ (UpdateAppointment ∧ Success) ∨ NotScheduled**

**1.3.5 Robust CancelAppointment Operation**

* For **CancelAppointment:**

**RCancelAppointment ≙ (CancelAppointment ∧ Success) ∨ NotScheduled**

### **1.3.6 Robust ViewAppointments Operation**

The robust version of the **ViewAppointments** operation ensures that it reports whether a patient is scheduled and, if so, provides their appointment. If the patient is not found, it reports the result accordingly.

**RViewAppointments ≙ (ViewAppointments ∧ Success) ∨ NotScheduled**

In this schema, if the patient is scheduled, the system proceeds to retrieve the appointment as described in **ViewAppointments** and the result is ok. However, if the patient is not found in the **scheduledPatients** set, the result is **not\_scheduled**.

### **1.4 Final Specification Summary**

The formal specification of the Hospital Appointment Booking System described various aspects of the system using Z notation, including:

* **State space**: Represented by the AppointmentBook schema, which contains the set of scheduled patients and the mapping of patients to appointment times.
* **Operations**: The system's operations are formalized with schemas for adding, updating, finding, viewing, and canceling appointments, all described with clear preconditions and postconditions.
* **Error handling**: Robust versions of operations are defined to handle errors such as attempting to add an appointment for an already scheduled patient or trying to find an appointment for a non-existent patient.
* **Success reporting**: Each operation produces a result (**ok, already\_scheduled, not\_scheduled**), and the specification defines how these results should be reported for different scenarios.

The specification also illustrates how mathematical reasoning can be used to ensure the correctness of the system without needing to implement it right away, helping to identify potential issues early in the design phase.

This formal specification provides a clear and mathematically sound foundation for developing the Hospital Appointment Booking System, ensuring that all behaviors are defined and all edge cases are considered before implementation.

[Z Editor Online Tool](https://z-editor.github.io/) — Used for creating and visualizing Z schemas online.

# **2. Appendices**

### **Appendix A: Basic Definitions**

* **PATIENT**: Represents the set of all patients registered in the hospital.
* **TIME**: Represents all available appointment time slots.

### **Appendix B: Schema Symbols**

* **∆ (Delta)**: Denotes a change in the system state.
* **Ξ (Xi)**: Denotes no change in the system state.
* **↦ (Mapping arrow)**: Represents a mapping from a patient to an appointment time.

### **Appendix C: Report Status Definitions**

* **ok**: Operation successful.
* **already\_scheduled**: Attempted to schedule an already scheduled patient.
* **not\_scheduled**: Attempted operation on a patient who is not scheduled.

### **Appendix D: Example of Initial System State**

* **Scheduled Patients**: None
* **Appointments**: None
* The system starts with an empty appointment book.

## **Appendix E: GUI Implementation Snapshot (Python + Tkinter)**

The following image represents the GUI design of the **Hospital Appointment Booking System,** developed using Python and Tkinter.  
This interface supports patient appointment management functionalities including adding, finding, viewing, updating, and canceling appointments according to z language above mentioned requirements.

**These are provided for reference only and are not part of the formal Z specification.**

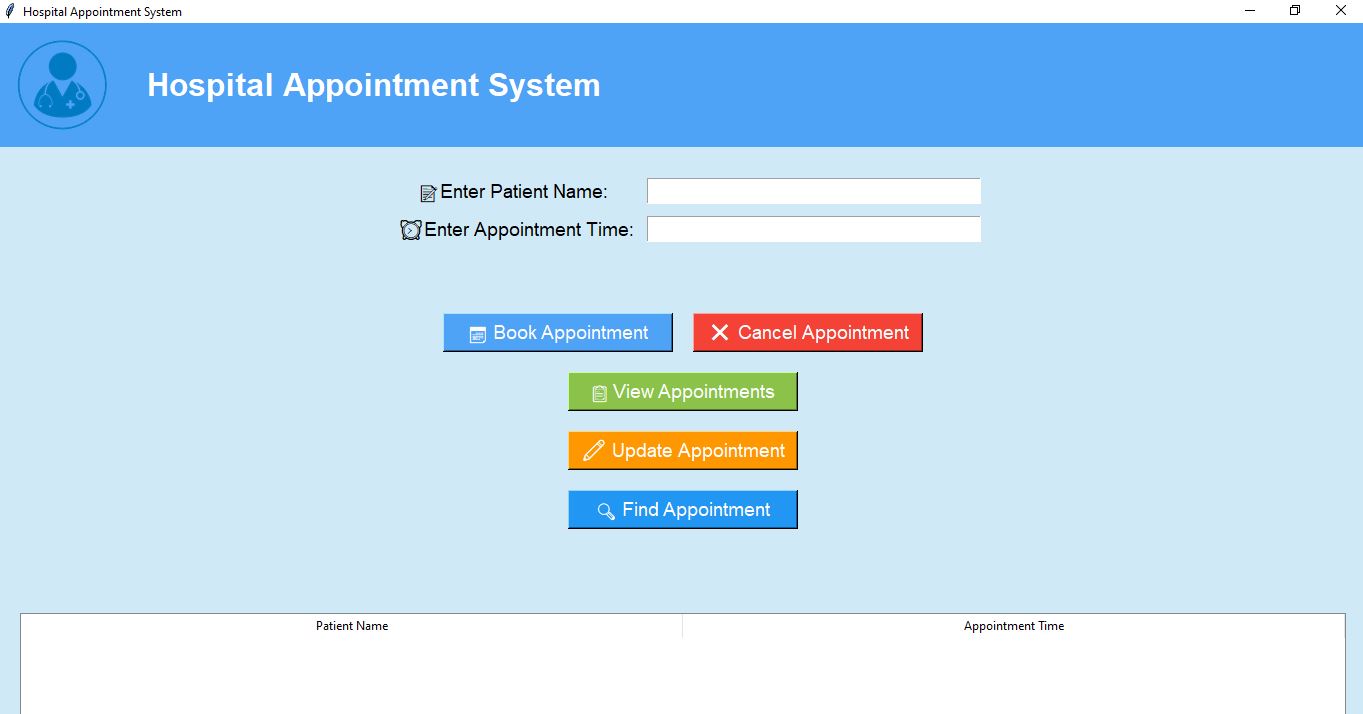


Figure 1 GUI for Hospital Appointment Booking System (Python + Tkinter)

# **3. References**

* [Python Tkinter Official Documentation](https://docs.python.org/3/library/tkinter.html)
* [Formal Methods in Software Engineering (Wikipedia)](https://en.wikipedia.org/wiki/Formal_methods)
* [Z Editor Online Tool](https://z-editor.github.io/)

## **3.1. Project Links and Additional Resources**

|  |  |  |
| --- | --- | --- |
| Resource | Description | Link |
| GitHub Repository | Source code and SRS document for the Hospital Appointment Booking System. | [GitHub Repo](https://github.com/Hifza-Khalid/FormalMethodsInSE) |
| Web Version | Front-end implementation of the system deployed on Netlify. | [**Live Website**](https://hospitalappointmentbookingsystem.netlify.app/) |
| Blog Post | Blog article explaining the design, implementation, and learning experience for this project. | [**Read Blog**](https://techinsightswithhifza.blogspot.com/2025/04/hospital-appointment-booking-system.html) |
| YouTube Demo | Video demonstration showing the features and workflow of the Hospital Appointment Booking System. | [**Watch on Youtube**](https://www.youtube.com/watch?v=22Ww32q92L0&ab_channel=Hifza%27sTechHub) |