Verified Meeting Scheduler Backend

Using Spring Boot, Z3/Coq, Runtime Verification, and Optional PRISM

---

1. Project Proposal (Teacher-ready)

Project Title

Verified Meeting Scheduler Backend using Spring Boot, Z3 (or Coq), and Runtime Verification
(with optional probabilistic analysis via PRISM)

Overview

This project develops a fully verified meeting scheduling backend using **Java Spring Boot**. Correctness of scheduling decisions is enforced using: - **Z3 SMT Solver** *or* **Coq** for static constraint validation, - **Runtime Verification (RV-Monitor/JavaMOP)** for dynamic monitoring of system behavior, - *(Optional)* **PRISM** model checking for probabilistic evaluation of scheduling success and load.

The system ensures that meeting creation, deletion, and updates follow strict safety and protocol properties. The project combines formal methods with practical backend engineering, meeting Software Engineering course requirements.

Objectives

1. Build a Spring Boot backend for meeting scheduling.
2. Use **Z3** *or* **Coq** to verify scheduling feasibility.
3. Use **Runtime Verification** to observe safety & liveness properties at runtime.
4. Demonstrate layered correctness: static + dynamic.
5. *(Optional)* Apply PRISM to analyze probabilities of scheduling success.

Motivation

Scheduling systems often suffer from inconsistent states, double bookings, and protocol violations. Combining formal verification techniques with a modern backend solves these issues and illustrates how such tools can enhance real-world software reliability.

Team Roles

Student A — Static Verification (Z3 or Coq) - Encodes scheduling constraints. - Integrates solver into Spring Boot. - Generates explanations for failures.

Student B — Runtime Verification - Specifies temporal correctness properties. - Instruments Spring Boot services with monitors. - Ensures no bad states occur at runtime.

Scope is perfect for 6 weeks

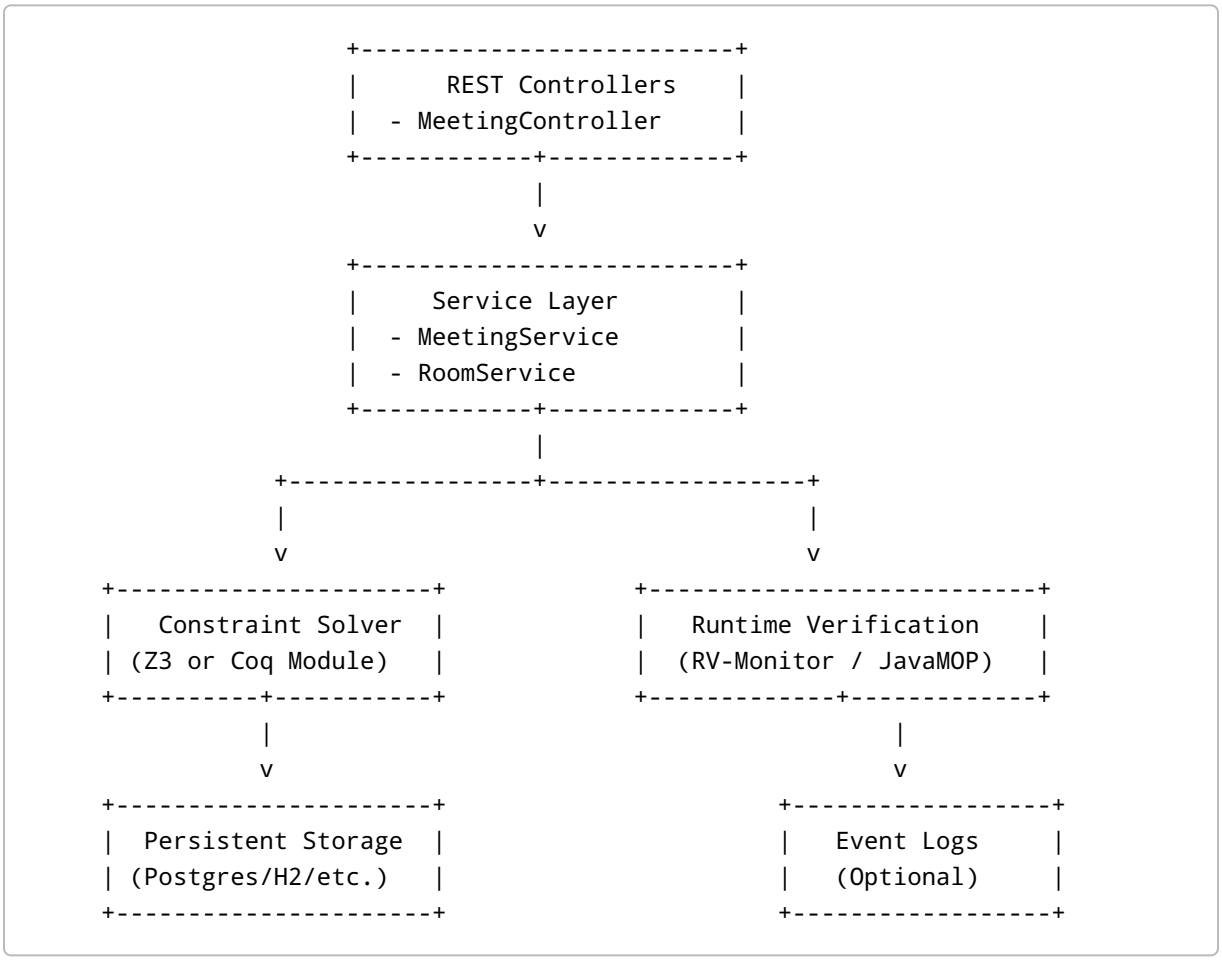
- Clear separation of work,
- Each student learns a new tool,
- Implementation remains realistic in a half-semester timeframe.

Expected Deliverables

- Spring Boot application
- Z3 or Coq constraint reasoning module
- Runtime monitors
- Test cases and evaluation
- Optional PRISM analysis
- Final report + demo

---

2. Architecture Diagram (ASCII UML)



---

3. Z3 Constraint Examples (Java Pseudo-Code)

Example: No overlapping meetings in same room

```
BoolExpr noOverlap = ctx.mkImplies(
    ctx.mkAnd(
        ctx.mkEq(room1, room2),
        ctx.mkLt(start1, end2),
        ctx.mkLt(start2, end1)
    ),
    ctx.mkFalse()
);
solver.add(noOverlap);
```

Example: All required participants must be free

```
solver.add(
   ctx.mkNot(
      ctx.mkAnd(
         ctx.mkEq(participant, participantOther),
         ctx.mkLt(startRequested, endOther),
         ctx.mkLt(startOther, endRequested)
      )
   )
);
```

Example: Checking satisfiability

```
if (solver.check() == Status.SATISFIABLE) {
    Model m = solver.getModel();
    return Success(m);
} else {
    return Failure(solver.getUnsatCore());
}
```

---

4. Coq Alternative (Replacing Z3)

Coq can be used instead of Z3, but with a different approach: - Coq is a *proof assistant* rather than an automated solver. - You **prove the correctness of your scheduling function**, rather than solving arbitrary constraints.

Example Design if Using Coq

- Define a list of meetings.
- Define a function `insert_meeting`.
- Prove the theorem:

```
Theorem insert_preserves_nonoverlap:
  ∀ m meetings, no_overlap meetings → no_overlap (insert_meeting m
meetings).
```

- Export the verified function to OCaml/Java.

Practical Notes

- **Z3 is FAR easier to integrate into Spring Boot.**
- **Coq requires more setup and produces a standalone verified algorithm**, not a general solver.
- Use Coq only if you want a more theoretical proof-heavy project.

Recommendation

✔️Use **Z3** for the main project.
✔️Use **Coq** only for a small verified component or extension.

---

5. Runtime Verification Properties (LTL-like)

Property 1 — Every create must be confirmed or rejected

```
G (create(id) → F (confirm(id) ∨ reject(id)))
```

Property 2 — No deletion of nonexistent meetings

```
G (delete(id) → previouslyCreated(id))
```

Property 3 — No overlapping meetings appear in runtime state

```
G ¬ overlaps(meetingA, meetingB)
```

Property 4 — A room cannot exceed capacity

```
G (assign(room, attendees) → attendees ≤ capacity(room))
```

Runtime Monitor Snippet (JavaMOP-like)

```
@Monitor
public class MeetingMonitor {
    @OnEvent("create")
    public void onCreate(Meeting m) { ... }

    @Property("G(create → F(confirm ∨ reject))")
    public void checkCreateFlow(...) { ... }
}
```

---

6. Optional PRISM Extension

Goal: build a **probabilistic model** of meeting scheduling.

Example PRISM Model Elements

- **States**: idle, processing, success, failure.
- **Transitions**: with probabilities based on real request frequencies.
- **Properties**:

```
P>=0.95 [ F success ]
R{"waiting_time"}min=? [ F done ]
```

What You Would Model

- Arrival rate of requests.
- Probability a meeting conflicts.
- Expected success rate at peak hours.

This is an *optional but impressive* extension.

---

7. 6-Week Work Plan

Week 1

- Define requirements, data model, REST API.

Week 2

- Implement CRUD meeting scheduler in Spring Boot.

Weeks 3–4

- Integrate Z3 (Student A)
- Implement Runtime Verification monitors (Student B)

Week 5

- Testing, refinement, demo preparation.
- (Optional) PRISM probabilistic model.

Week 6

- Final report and presentation.

---

8. Summary

This project successfully integrates: - **Spring Boot** (industry-grade backend) - **Z3 or Coq** (static formal methods) - **Runtime Verification** (dynamic correctness) - **Optionally PRISM** (probabilistic analysis)

It is perfectly scoped for 2 students and guaranteed to satisfy a demanding instructor.