Module Interface Specification for MES-ERP

Team #26, Ethical Pals
Sufyan Motala
Rachid Khneisser
Housam Alamour
Omar Muhammad
Taaha Atif

April 4, 2025

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

2 Symbols, Abbreviations and Acronyms

See SRS Documentation.

Additional symbols used in this MIS:

Symbol	Description
$\mathbb{Z}$	Set of integers
$\mathbb{R}$	Set of real numbers
$\mathbb{N}$	Set of natural numbers (positive integers)
$\mathbb{N}_0$	Set of non-negative integers (0, 1, 2, ...)
seq of T	Sequence containing elements of type T
T × U	Tuple containing elements of type T and U
:=	Assignment
(c ⇒ r)	Conditional rule (if condition c holds, result r a
Map(K → V)	Map from keys of type K to values of type V
Maybe¡T¿	Type T or null/nothing
Partial¡T¿	A record where some fields of Type T may be p
`Blob`	Represents binary data (e.g., file content)
`Timestamp`	String representing ISO 8601 date-time

Contents

3 Introduction

The following document details the Module Interface Specifications (MIS) for the McMaster Engineering Society Custom Financial Expense Reporting Platform (MES-ERP). This platform is designed to streamline financial expense management for the McMaster Engineering Society (MES), providing an efficient and user-friendly solution for submitting, approving, and tracking reimbursement requests.

The MES-ERP aims to address the unique financial management needs of the MES by integrating expense tracking, budget management, and policy compliance into a cohesive platform. The system ensures accurate and efficient handling of financial requests while maintaining compliance with organizational policies and university regulations.

Complementary documents to this MIS include the System Requirements Specification (SRS) and the Module Guide (MG), which provide additional context and design details. The complete documentation and implementation of the MES-ERP can be found at https://github.com/Housam2020/MES-ERP.

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for assignment and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by MES-ERP.

Data Type	Notation	Description
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in $(-\infty, \infty)$
natural number	$\mathbb{N}$	a number without a fractional component in $[1, \infty)$
real	$\mathbb{R}$	any number in $(-\infty, \infty)$
boolean	Bool	True or False value
string	String	A sequence of characters

The specification of MES-ERP uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type (e.g., seq of $\mathbb{R}$). Strings are sequences of characters. Tuples contain a list of values, potentially of different types (e.g., String $\times$ $\mathbb{R}$). In addition, MES-ERP uses functions, which are defined by the data types of their inputs and outputs (e.g., $f : \mathbb{R} \rightarrow$ Bool). Local functions are described by giving their type signature followed by their specification. Object types may be defined using record notation, e.g., `Type = record of field1: String, field2: ℝ end`.

5 Timeline

This section outlines the timeline for the implementation of the project. The timeline includes the development of all modules, testing, and deployment phases. Tasks are divided by modules, specifying responsibilities and key milestones.

5.1 Development Timeline

Week	Task	Details
Week 1	Initial Planning	Team meeting to finalize requirements and review the SRS. Assign responsibilities for each module.
Week 2	User Authentication and Profile Management Module	Development of secure login, roles, and basic profile updates. Begin unit testing for authentication.
Week 3	Expense Submission and Tracking Module	Implement submission forms for expenses, including receipt uploads and status tracking. Start unit testing.
Week 4	Budget and Funding Management Module	Develop logic for fetching budgets, validating funds, and updating department budgets. Integrate with the database module.
Week 5	Approval Workflow and Review Module	Implement dynamic routing rules and approval workflows. Integrate notifications for pending approvals. Conduct unit testing.
Week 6	Notifications & Compliance	Build Notifications (M??) and Policy/Compliance (M??) modules. Integrate notifications for key events.
Week 7	Continued Modules	Continue development on BH modules as needed.
Week 8	Reporting and Analytics Module	Develop functionality for generating reports (analytics views) and tracking usage statistics. Validate with sample data.

5.2 Testing and Verification

Testing will be conducted in multiple phases:

- Unit Testing: Conducted during the implementation of each module (Weeks 2–11).
- Integration Testing: Performed once modules are integrated (Week 12).
- System Testing: Comprehensive testing of the entire system to ensure functionality and performance (Week 12–13).
- User Acceptance Testing: Gather feedback from end-users during Week 13 to identify potential areas for improvement.

5.3 Responsibilities

The following responsibilities are assigned to team members:

- Module Implementation: Each team member is responsible for implementing the modules assigned to them during the initial planning phase.
- Documentation: All team members contribute to the MIS and ensure consistency with the SRS and MG.
- Testing: Shared responsibility for writing and executing test cases, with module developers performing unit tests.
- Deployment: Coordinated by the team lead, with support from all team members for configuration and setup.

6 Module Decomposition

This section provides the Module Interface Specifications (MIS) for the modules identified in the Module Guide (MG) Section **??**. The decomposition follows the HH, BH, and SD layers defined in the MG.

7 MIS of Database Module (M??)

7.1 Module

Database Interaction Layer

7.2 Uses

None (Hardware-Hiding Module).

7.3 Syntax

7.3.1 Exported Constants

- `DATABASE_URL`: String - The URL for the database connection.
- `MAX_BATCH_SIZE`: $\mathbb{N}$ - The maximum number of records to fetch in a single query. (e.g., 1000)
- `MAX_CONNECTIONS`: $\mathbb{N}$ - The maximum number of concurrent database connections allowed by the pool. (e.g., 10)
- `DEFAULT_TIMEOUT`: $\mathbb{N}$ - The default timeout in milliseconds for database queries. (e.g., 30000)

7.3.2 Exported Access Programs

Note: 'Object' type represents a structured record or JSON-like object. 'Array' represents a sequence.

Name	Input	Output	Exceptions
query	queryString: String, params: Object	results: Array of Object	DBQueryError
insert	collection: String, document: Object	documentID: String	DBInsertError
update	collection: String, filter: Object, updates: Object	modifiedCount: $\mathbb{N}_0$	DBUpdateError
delete	collection: String, filter: Object	deletedCount: $\mathbb{N}_0$	DBDeleteError
transaction	operations: Array of { type: String, collection: String, ...params }	results: Array of Object	DBTransactionError

7.4 Semantics

7.4.1 State Variables

- `dbState`: {Connected, Disconnected, Error} - Represents the current connection status to the database server.

7.4.2 Environment Variables

- `DatabaseServer`: Represents the external database system (e.g., Supabase PostgreSQL instance) holding the persisted data.

7.4.3 Assumptions

- `DatabaseServer` is accessible. A1
- Database schema (table structures, constraints) is predefined and consistent with the application's expectations. A2
- Authentication credentials (e.g., API keys) are securely managed externally (e.g., environment variables) and provided correctly during initialization. A3
- Network connectivity exists between the application server and the `DatabaseServer`. A4

7.4.4 Access Routine Semantics

`query(queryString, params)`:

- **Transition**: (`dbState` = Connected) $\implies$ Executes the database query specified by `queryString` with bound `params` against the `DatabaseServer`.
- **Output**: `out` := `results` (Array of Object representing rows matching the query).
- **Exceptions**: (`dbState` $\neq$ Connected)$\vee$(Query execution fails on `DatabaseServer`) $\implies$ `exc` := DBQueryError
- **Precondition**: `queryString` is valid SQL syntax for the target database. `params` structure matches placeholders in `queryString`. (Uses `sanitizeQuery`)
- **Postcondition**: Returned `results` are consistent with the state of the `DatabaseServer` at the time of query execution.

`insert(collection, document)`:

- **Transition**: (`dbState` = Connected)$\wedge$(`validateSchema(collection, document)`) $\implies$ Persists the `document` into the specified `collection` on the `DatabaseServer`.
- **Output**: `out` := `documentID` (String representing the unique ID assigned by the `DatabaseServer`).
- **Exceptions**: (`dbState` $\neq$ Connected)$\vee$($\neg$`validateSchema(collection, document)`)$\vee$ (Insertion fails on `DatabaseServer`) $\implies$ `exc` := DBInsertError

- **Precondition**: `document` conforms to the expected schema for the `collection`. (Uses `validateSchema`)
- **Postcondition**: `DatabaseServer` state reflects the newly inserted `document`.

`update(collection, filter, updates)`:

- **Transition**: (`dbState` = Connected) $\implies$ Applies the specified `updates` to documents in the `collection` on the `DatabaseServer` that match the `filter` criteria.
- **Output**: `out` := `modifiedCount` ($\mathbb{N}_0$ representing the number of documents updated).
- **Exceptions**: (`dbState` $\neq$ Connected)$\vee$(Update operation fails on `DatabaseServer`) $\implies$ `exc` := DBUpdateError
- **Precondition**: `filter` and `updates` are valid for the database schema.
- **Postcondition**: Matching documents on the `DatabaseServer` are updated.

`delete(collection, filter)`:

- **Transition**: (`dbState` = Connected) $\implies$ Removes documents from the `collection` on the `DatabaseServer` that match the `filter` criteria.
- **Output**: `out` := `deletedCount` ($\mathbb{N}_0$ representing the number of documents removed).
- **Exceptions**: (`dbState` $\neq$ Connected) $\vee$ (Deletion fails on `DatabaseServer`) $\implies$ `exc` := DBDeleteError
- **Precondition**: `filter` is specified to prevent accidental mass deletion.
- **Postcondition**: Matching documents are removed from the `DatabaseServer`.

`transaction(operations)`:

- **Transition**: (`dbState` = Connected) $\implies$ Executes the sequence of `operations` (inserts, updates, deletes) as an atomic transaction on the `DatabaseServer`. All operations succeed or all are rolled back.
- **Output**: `out` := `results` (Array of outputs corresponding to each operation if successful).
- **Exceptions**: (`dbState` $\neq$ Connected)$\vee$(Any operation is invalid)$\vee$(Transaction fails on `DatabaseSe`
  `exc` := DBTransactionError
- **Precondition**: All individual `operations` within the sequence are valid.
- **Postcondition**: `DatabaseServer` state reflects the successful completion of all operations, or remains unchanged if any operation failed.

7.4.5 Local Functions

- `validateSchema(collection: String, document: Object) → Bool`: Checks if the structure and data types of `document` match the expected schema for the given `collection`. (Implementation details hidden).
- `sanitizeQuery(query: String) → String`: Escapes potentially harmful characters in `query` to prevent SQL injection attacks. (Implementation details hidden).

8 MIS of User Authentication & Profile Management Module (M??)

8.1 Module

User Authentication & Profile Management

8.2 Uses

- Database Interaction Layer (M??)

8.3 Syntax

8.3.1 Exported Constants

- `SESSION_TIMEOUT`: $\mathbb{N}$ - The duration (in seconds) before an inactive session expires. (e.g., 3600)
- `MAX_LOGIN_ATTEMPTS`: $\mathbb{N}$ - Maximum number of failed login attempts before account lockout. (e.g., 5)
- `LOCKOUT_DURATION`: $\mathbb{N}$ - Duration (in seconds) of account lockout after exceeding `MAX_LOGIN_ATTEMPTS` (e.g., 600)

8.3.2 Exported Types

Type `Credentials` = record of email: String, password: String end
Type `ProfileData` = record of userID: String, email: String, fullName: String, phoneNum: String, /* etc. */ end
Type `Permissions` = record of allowedActions: seq of String end
Type `Timestamp` = String representing ISO 8601 date-time

8.3.3 Exported Access Programs

Name	Input	Output	Exceptions
authenticate	credentials: Credentials	sessionToken: String	AuthenticationFailed
getProfile	userID: String	profileData: ProfileData	UserNotFound
updateProfile	userID: String, updates: Partial¡ProfileData¿	confirmation: Bool	InvalidProfileData
validateSession	sessionToken: String	(isValid: Bool, userID: Maybe¡String¿)	None
getUserPermissions	userID: String	permissions: Permissions	UserNotFound
registerUser	credentials: Credentials	userID: String	RegistrationFailed
signOut	sessionToken: String	confirmation: Bool	None

8.4 Semantics

8.4.1 State Variables

- `sessions`: Map (String → record of userID: String, expiry: Timestamp end) - A mapping of active session tokens to user information and expiration times.
- `loginAttempts`: Map (String → record of count: $\mathbb{N}_0$, lockoutExpiry: Timestamp end) - A record of failed login attempts per user (keyed by email or IP).

8.4.2 Environment Variables

- `AuthProvider`: Represents the external authentication service (Supabase Auth) which manages credentials and issues tokens.

8.4.3 Assumptions

- The `AuthProvider` is available and functioning correctly. A5
- User emails are unique identifiers for authentication. A6
- Input data for profile updates is validated by the caller or within `updateProfile`. A7
- The Database module (M??) is available for retrieving/storing non-credential profile data. A8

8.4.4 Access Routine Semantics

`authenticate(credentials)`:

- **Transition**: Verifies `credentials` against `AuthProvider`. If valid and user not locked out, generates a new session token, updates `sessions` map, resets `loginAttempts` for the user. If invalid, increments `loginAttempts`; if count exceeds `MAX_LOGIN_ATTEMPTS`, sets lockout expiry.
- **Output**: (Authentication succeeds) $\implies$ `out` := `sessionToken` (String)

- **Exceptions**: (Authentication fails ∨ User locked out) $\implies$ `exc` := AuthenticationFailed

`getProfile(userID)`:

- **Transition**: Calls M??::query to retrieve profile data for `userID` from the `'users'` table.
- **Output**: (User found in DB) $\implies$ `out` := `profileData` (ProfileData)
- **Exceptions**: (User not found in DB ∨ DB query fails) $\implies$ `exc` := UserNotFound

`updateProfile(userID, updates)`:

- **Transition**: Validates `updates` (A7). Calls M??::update to apply changes to the `'users'` table for the given `userID`.
- **Output**: (Update succeeds) $\implies$ `out` := True
- **Exceptions**: (Invalid `updates` ∨ DB update fails) $\implies$ `exc` := InvalidProfileData

`validateSession(sessionToken)`:

- **Transition**: Checks if `sessionToken` exists in `sessions` map and if its expiry time has not passed.
- **Output**: (Session is valid) $\implies$ `out` := (True, `userID`). (Session invalid or expired) $\implies$ `out` := (False, null).
- **Exceptions**: None (returns validity status).

`getUserPermissions(userID)`:

- **Transition**: Calls M??::query to retrieve all roles associated with `userID` from `'user_roles'`, then queries `'role_permissions'` and `'permissions'` tables to aggregate all permission names.
- **Output**: (User found and permissions retrieved) $\implies$ `out` := `permissions` (Permissions)
- **Exceptions**: (User not found ∨ DB query fails) $\implies$ `exc` := UserNotFound

`registerUser(credentials)`:

- **Transition**: Calls `AuthProvider` to create a new user with given `credentials`. If successful, calls M??::insert to create a corresponding record in the `'users'` table and assigns a default role via M??::insert into `'user_roles'`.
- **Output**: (Registration succeeds) $\implies$ `out` := `userID` (String)

- **Exceptions**: (Registration fails at AuthProvider $\vee$ DB insert fails) $\implies$ `exc` := RegistrationFailed

`signOut(sessionToken)`:

- **Transition**: Removes the entry corresponding to `sessionToken` from the `sessions` map. May optionally call `AuthProvider` to invalidate the token externally if applicable.
- **Output**: (Sign out successful or token not found) $\implies$ `out` := True
- **Exceptions**: None (fails silently or returns True).

8.4.5 Local Functions

None.

9 MIS of Expense Submission & Tracking Module (M??)

9.1 Module

Expense Submission & Tracking

9.2 Uses

- Database Interaction Layer (M??)
- Data Validation Module (M??)
- Policy & Compliance Management Module (M??)

9.3 Syntax

9.3.1 Exported Constants

- `MAX_RECEIPT_SIZE`: $\mathbb{N}$ - The maximum file size (in bytes) for uploaded receipts. (e.g., 5242880 for 5MB)
- `ALLOWED_FILE_TYPES`: seq of String - List of accepted file formats for receipts [”pdf”, ”jpg”, ”png”, ”jpeg”].
- `EXPENSE_CATEGORIES`: seq of String - Predefined expense categories [”conference”, ”travel”, ”supplies”, ”materials”, . . .]. (Note: May be dynamically loaded from DB instead)

9.3.2 Exported Types

Type `ExpenseDetails` = record of group_id: String, amount_requested_cad: $\mathbb{R}$, budget_line: String, /* etc. */ end
Type `Attachment` = record of fileName: String, fileData: `Blob`, /* etc. */ end
Type `ExpenseStatus` = record of status: String, timestamp: Timestamp, /* etc. */ end
Type `ExpenseSummary` = record of requestID: String, submitterName: String, amount: $\mathbb{R}$, status: String end

9.3.3 Exported Access Programs

Name	Input	Output	Exceptions
submitExpense	expenseDetails: ExpenseDetails, attachments: seq of Attachment	requestID: String	InvalidExpense
uploadAttachment	requestID: String, file: Attachment	fileID: String	FileUploadFailed
getExpenseStatus	requestID: String	status: ExpenseStatus	RequestNotFound
updateExpenseDetails	requestID: String, updates: Partial¡ExpenseDetails¿	confirmation: Bool	InvalidUpdate
searchExpenses	filters: Object	expenseList: seq of ExpenseSummary	InvalidSearch

9.4 Semantics

9.4.1 State Variables

- None (This module is largely stateless; request data is persisted via the DB module).

9.4.2 Environment Variables

- `FileStorageProvider`: Represents the external service (e.g., Supabase Storage) used for persisting uploaded files.

9.4.3 Assumptions

- All monetary values provided in `amount_requested_cad` are assumed correct after potential currency conversion if applicable. A9
- `FileStorageProvider` is available and has sufficient capacity. A10
- Users have necessary permissions (e.g., `create_requests`) to submit expenses, verified by the calling context (e.g., middleware or UI). A11

9.4.4 Access Routine Semantics

`submitExpense(expenseDetails, attachments)`:

- **Transition**: Calls M??::validate(`expenseDetails`). If valid, generates a unique `requestID`. Persists `expenseDetails` (with `requestID`, `userID` from context, `status`='Pending') via M??::insert into `payment_requests`. For each attachment, calls `uploadAttachment`. Calls M??::logAction(type='SubmitRequest', ...).
- **Output**: (Submission successful) $\implies$ `out` := `requestID` (String)
- **Exceptions**: (Validation fails $\vee$ DB insert fails $\vee$ Any attachment upload fails) $\implies$ `exc` := InvalidExpense
- **Precondition**: `expenseDetails` contains required fields (e.g., amount, group_id, budget_line). User context provides `userID`.
- **Postcondition**: Expense request record created in DB with 'Pending' status. Associated attachments stored via `FileStorageProvider`. Audit log updated.

`uploadAttachment(requestID, file)`:

- **Transition**: Calls M??::validateFile(`file`, `MAX_RECEIPT_SIZE`, `ALLOWED_FILE_TYPES`). If valid, generates a unique `fileID`. Stores `file.fileData` via `FileStorageProvider`. Calls M??::insert or update to link `fileID` and metadata (`file.fileName`) to the request identified by `requestID` (e.g., in an `attachments` table or JSONB field). Optionally calls internal OCR function (`_extractAmountFromReceipt`) on `file.fileData`.
- **Output**: (Upload successful) $\implies$ `out` := `fileID` (String)
- **Exceptions**: (Validation fails $\vee$ File storage fails $\vee$ DB update fails) $\implies$ `exc` := FileUploadFailed
- **Precondition**: `requestID` refers to an existing request. `file` contains valid data.
- **Postcondition**: File is stored externally. DB record links file to the request. Potential update to request's amount based on OCR.

`getExpenseStatus(requestID)`:

- **Transition**: Calls M??::query to retrieve the status and relevant tracking info for the request identified by `requestID` from `payment_requests`.
- **Output**: (Request found) $\implies$ `out` := `status` (ExpenseStatus)
- **Exceptions**: (Request not found $\vee$ DB query fails) $\implies$ `exc` := RequestNotFound
- **Postcondition**: Status information is retrieved.

`updateExpenseDetails(requestID, updates)`:

- **Transition**: Fetches current request state via M??. Checks if request `status` allows editing (e.g., 'Pending', 'Rejected'). Calls M??::validate(`updates`). If valid and editable, calls M??::update to apply changes to `payment_requests` for `requestID`. Calls M??::logAction(type='UpdateRequest', ...).
- **Output**: (Update successful) $\implies$ `out` := True
- **Exceptions**: (Request not found $\vee$ Request not editable $\vee$ Validation fails $\vee$ DB update fails) $\implies$ `exc` := InvalidUpdate
- **Precondition**: `requestID` exists. Request is in an editable state. User context provides permissions.
- **Postcondition**: Expense details updated in DB. Audit log updated.

`searchExpenses(filters)`:

- **Transition**: Constructs a query based on `filters` (e.g., status, group_id, date range). Calls M??::query on `payment_requests` (potentially joining with `users`, `groups`) to retrieve matching requests.
- **Output**: (Query successful) $\implies$ `out` := `expenseList` (seq of ExpenseSummary)
- **Exceptions**: (Invalid `filters` $\vee$ DB query fails) $\implies$ `exc` := InvalidSearch
- **Postcondition**: List of matching expense summaries is returned.

9.4.5 Local Functions

- `_extractAmountFromReceipt(fileData: Blob)` $\rightarrow$ `Maybe<`$\mathbb{R}$`>`: Internal helper that attempts OCR on the receipt data and returns an extracted amount or null. (Uses Tesseract.js library).

10 MIS of Approval Workflow and Review Module (M??)

10.1 Module

Approval Workflow and Review

10.2 Uses

- Database Interaction Layer (M??)
- Budget and Funding Management Module (M??)

- Notifications and Communication Module (M??)
- Policy & Compliance Management Module (M??)

10.3 Syntax

10.3.1 Exported Constants

None.

10.3.2 Exported Types

Type `RequestStatus` = String (e.g., "Pending", "Approved", "Rejected")
Type `UserDetails` = record of userID: String, roles: seq of String, groups: seq of String end

10.3.3 Exported Access Programs

Name	Input	Output	Exceptions
getPendingRequests	approverDetails: UserDetails	pendingList: seq of ExpenseSummary	AccessDenied
addNote	requestID: String, note: String, userID: String	confirmation: Bool	RequestNotFound
updateStatus	requestID: String, newStatus: RequestStatus, approverID: String, comment: String (optional)	confirmation: Bool	InvalidStatus

10.4 Semantics

10.4.1 State Variables

- None (Workflow state stored in DB via M??).

10.4.2 Environment Variables

None.

10.4.3 Assumptions

- Approver's identity and permissions (`approverDetails` or `approverID`) are validated by the calling context. A12
- The Database (M??), Budget (M??), Notification (M??), and Compliance (M??) modules are available and functioning correctly. A13
- Approval rules (e.g., which roles can approve which requests, thresholds for multi-level approval) are defined and accessible (potentially configuration or within this module's logic). A14

10.4.4 Access Routine Semantics

`getPendingRequests(approverDetails)`:

- **Transition**: Queries M?? for requests with status 'Pending'. Filters the list based on `approverDetails`' roles/groups and defined approval rules (A14).
- **Output**: (Requests found matching criteria) $\implies$ `out` := `pendingList` (seq of ExpenseSummary)
- **Exceptions**: (Approver lacks permission to view any requests) $\implies$ `exc` := AccessDenied

`addNote(requestID, note, userID)`:

- **Transition**: Calls M??::update to add the `note` (associated with `userID` and timestamp) to the specified `requestID`'s record (e.g., in a comments field/table). Calls M??::logAction(type='AddNote', ...).
- **Output**: (Update successful) $\implies$ `out` := True
- **Exceptions**: (Request not found $\vee$ DB update fails) $\implies$ `exc` := RequestNotFound

`updateStatus(requestID, newStatus, approverID, comment)`:

- **Transition**: Fetches request via M??. Verifies `approverID` has permission to change status based on rules (A14) and request details (e.g., amount, group) using `_canApprove`. If `newStatus` is 'Approved', may call M??::validateFunds. If validation passes (or status is 'Rejected'), calls M??::update to set the new status, record `approverID`, timestamp, and `comment`. Calls M??::logAction(type='UpdateStatus', ...). Calls M??::sendNotification(rec newStatus).
- **Output**: (Status update successful) $\implies$ `out` := True
- **Exceptions**: (Request not found $\vee$ Approver lacks permission $\vee$ Budget validation fails for 'Approved' $\vee$ Invalid `newStatus` transition $\vee$ DB update fails) $\implies$ `exc` := InvalidStatus

10.4.5 Local Functions

- `_canApprove(approverDetails: UserDetails, request: ExpenseDetails) → Bool`: Internal function checking if the approver has the necessary role/group permissions based on defined rules (A14) and request properties.

11 MIS of Budget and Funding Management Module (M??)

11.1 Module

Budget and Funding Management

11.2 Uses

- Database Interaction Layer (M??)
- Policy & Compliance Management Module (M??)

11.3 Syntax

11.3.1 Exported Constants

None.

11.3.2 Exported Types

Type `BudgetLine` = record of id: Maybe¡ℕ¿, group_id: String, line_label: String, amount: ℝ, line_type: String, order_index: ℕ end
Type `GroupWithLines` = record of id: String, name: String, total_budget: ℝ, lines: seq of BudgetLine end
Type `BudgetDetails` = record of groupID: String, totalAllocated: ℝ, currentSpent: ℝ, lines: seq of BudgetLine end

11.3.3 Exported Access Programs

Name	Input	Output	Exceptions
getBudget	groupID: String	budgetDetails: BudgetDetails	GroupNotFound
validateFunds	requestAmount: ℝ, groupID: String	isSufficient: Bool	GroupNotFound
getOperatingBudget	userID: String	budgetData: seq of GroupWithLines	AccessDenied
saveOperatingBudget	budgetData: seq of GroupWithLines, userID: String	confirmation: Bool	SaveFailed, AccessDenied

11.4 Semantics

11.4.1 State Variables

- None (Budget data stored in DB via M??)

11.4.2 Environment Variables

None.

11.4.3 Assumptions

- Budget structures (lines, amounts) stored in the database are accurate. A15
- Permissions for viewing/modifying budgets (`userID`) are checked by the caller. A16

11.4.4 Access Routine Semantics

`getBudget(groupID)`:

- **Transition**: Calls M??::query to retrieve group details (`total_budget`) and associated budget lines (`operating_budget_lines`) for the given `groupID`. Calculates `currentSpent` based on associated expense lines.
- **Output**: (Group found) $\implies$ `out` := `budgetDetails` (BudgetDetails)
- **Exceptions**: (Group not found $\vee$ DB query fails) $\implies$ `exc` := GroupNotFound

`validateFunds(requestAmount, groupID)`:

- **Transition**: Calls `getBudget(groupID)`. Compares `requestAmount` against (`totalAllocated - currentSpent`).
- **Output**: (`totalAllocated - currentSpent` ¿= `requestAmount`) $\implies$ `out` := True. Else $\implies$ `out` := False.
- **Exceptions**: (Group not found) $\implies$ `exc` := GroupNotFound

`getOperatingBudget(userID)`:

- **Transition**: Checks if `userID` has permission (A16). Calls M??::query to retrieve all groups and all operating budget lines, ordering them appropriately.
- **Output**: `out` := `budgetData` (structured representation of groups and their lines).
- **Exceptions**: (User lacks permission $\vee$ DB query fails) $\implies$ `exc` := AccessDenied

`saveOperatingBudget(budgetData, userID)`:

- **Transition**: Checks if `userID` has permission (A16). Iterates through `budgetData`. For each group, updates its total via M??::update. For each line, determines if it's new, updated, or deleted, and calls appropriate M??::insert, M??::update, or M??::delete operations within a transaction (M??::transaction). Calls M??::logAction(type='SaveBudget', userID=userID, ...).
- **Output**: (Save successful) $\implies$ `out` := True
- **Exceptions**: (User lacks permission) $\implies$ `exc` := AccessDenied. (DB transaction fails) $\implies$ `exc` := SaveFailed

11.4.5 Local Functions

None.

12 MIS of Notifications & Communication Module (M??)

. . . *[Requires similar review/updates as above: Check Uses, Syntax, State/Env Vars, Assumptions, Access Semantics]* . . .

13 MIS of Reporting and Analytics Module (M??)

. . . *[Requires similar review/updates]* . . .

14 MIS of Graphical User Interface (GUI) Module (M??)

. . . *[Requires similar review/updates - Note: State vars here might represent UI state like active view/filters, Env var is the Browser Window]* . . .

15 MIS of Policy & Compliance Management Module (M??)

. . . *[Requires similar review/updates - Focus on logging actions and potentially validating against rules]* . . .

16 MIS of Administrator and Configuration Panel Module (M??)

. . . *[Requires similar review/updates - Focus on operations like addRole, updateGroup, etc., using other modules]* . . .

17 MIS of Data Validation Module (M??)

. . . *[Requires similar review/updates - Focus on validation functions for different data types/structures]* . . .

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

Appendix

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. **What went well while writing this deliverable?**
   Rachid: Collaborating as a team was smooth, and we were able to divide the modules effectively, which streamlined the writing process.
   Sufyan: Our TA meeting going over the modules and discussing it helped us understand the requirements better.
   Housam: Identifying each module's secrets and responsibilities early on helped maintain clarity and reduced redundancy in our design process.
   Taaha: Understanding a double checking our modules over with the TA helped tremendously, ensuring we had the correct approach
   Omar: The TA meeting went over a lot of our confusion that we had, in addition the extra alloted time allowed us to flesh out this deliverable a bit more.

2. **What pain points did you experience during this deliverable, and how did you resolve them?**
   Rachid: One challenge was ensuring consistency across modules. Regular team reviews and communication helped resolve any inconsistencies.
   Sufyan: Coordinating between the MG and MIS was a bit challenging, since we broke up who does each PDF.
   Housam: Aligning anticipated changes with module-level details was challenging but resolved through a thorough review of the SRS and MG.
   Taaha: Some of the questions were somewhat hard to understand what to do such as the ones requiring to create diagrams.
   Omar: A pain point I had was ensuring consistency between documents and ensuring all ideas align with eachother.

3. **Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g., your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?**

GROUP: Many decisions, like integrating notifications, came directly from stakeholder feedback. Others, like modular decomposition, were based on best practices and team experience.

4. **While creating the design doc, what parts of your other documents (e.g., requirements, hazard analysis, etc.), if any, needed to be changed, and why?**
   GROUP: The requirements document was updated to better align with the final design, specifically in the Reporting module to include export formats.

5. **What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)**
   Rachid: With unlimited resources, we could enhance system scalability and user interface design, making it more robust and user-friendly.
   Sufyan: We could also improve the integration with external systems such as banking APIs but we are limited by Open Banking not being available.
   Housam: With unlimited resources, we could incorporate advanced machine learning algorithms for OCR-based receipt processing to enhance accuracy and reduce manual intervention.
   Taaha: Provided unlimited resources, we could integrate better support with online banking organizations to allow for easier transactions.
   Omar: We could increase the use cases of the application, we could allow users more options based off what the MES requires or we could refine our solution to be more user friendly. We could have additionaly made a mobile app as well as a web app.

6. **Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)**
   Rachid: We considered alternative approaches for managing the workflow logic, such as using external libraries, but chose an in-house solution for simplicity and better control over implementation.
   Sufyan: We also considered using a third-party notification service, but opted for an in-house system to maintain data privacy and security.
   Housam: We debated between centralized compliance validation versus distributed validation across modules. While centralized validation offered simplicity, we chose the distributed approach to align better with modular decomposition and scalability goals.
   Taaha: We considered using external libraries to manage numerous functionality such as workflow process and notifications.
   Omar: A design solution I considered was making a mobile app instead of a web app, this in my opinion would have made the notifcaion process easier and more friendly to users. The tradeoff being the MES did not want this initially so we would not be making an appealing app to our main stakeholder. We decided to select a web app

since that would appeal to all stakeholders and we are still able to notify users via email.