Software Requirements Specification (SRS)

SWIFT MT7xx Message Automation System

Table of Contents

Software Requirements Specification (SRS)

SWIFT MT7xx Message Automation System

1. Introduction

1.1 Purpose

This Software Requirements Specification (SRS) document outlines the comprehensive requirements for the SWIFT MT7xx Message Automation System. The system aims to digitize and automate the processing, validation, and construction of SWIFT Category 7 (MT7xx) messages according to the 2019 standards. This document serves as the definitive reference for stakeholders, developers, and testers involved in the development and implementation of the system.

1.2 Document Conventions

This document follows standard SRS conventions with requirements categorized as: - **Mandatory (M)**: Essential for system functionality - **Optional (O)**: Desirable but not critical - **Conditional (C)**: Required only under specific conditions

1.3 Intended Audience

This SRS is intended for: - Project managers overseeing the development process - Software developers implementing the system - Quality assurance testers validating system functionality - Business analysts and stakeholders evaluating system capabilities - End users who will interact with the system

1.4 Project Scope

The SWIFT MT7xx Message Automation System will provide a comprehensive solution for handling Documentary Credits and Guarantees messages as defined in the SWIFT Category 7 standards for November 2019. The system will enable users to validate existing messages against SWIFT standards, construct new messages with proper formatting and field validation, and visualize message interdependencies. The system will incorporate an intuitive user interface for business users while maintaining strict adherence to SWIFT message standards.

1.5 References

1. SWIFT Standards MT November 2019 - Category 7 Documentary Credits and Guarantees (Advance Information)
2. SWIFT MT7xx Message Automation System Blueprint

1.6 System Overview

The SWIFT MT7xx Message Automation System will consist of several integrated components: 1. **Message Digitization Engine**: For parsing and structuring SWIFT MT7xx standards 2. **Validation Engine**: For verifying message compliance with SWIFT standards 3. **Message Composer**: For constructing valid MT7xx messages 4. **Field Dependency Graph**: For visualizing and enforcing field and message interdependencies 5. **User Interface**: For intuitive interaction with the system 6. **Database**: For storing message templates, validation rules, and business logic

The system will be accessible through a web-based interface and will also provide API endpoints for integration with other systems.

2. Functional Specifications (FS)

2.1 System Features

2.1.1 Message Validation

The system shall provide comprehensive validation of SWIFT MT7xx messages against the 2019 Category 7 standards. This includes:

The validation engine will parse incoming MT7xx messages by field identifiers and validate them against the SWIFT standards. It will check for the presence of mandatory fields, correct field formatting, proper sequencing, and adherence to network validated rules. The system will generate detailed validation reports highlighting any discrepancies or errors found in the message structure, content, or format.

For example, when validating an MT700 (Issue of a Documentary Credit) message, the system will verify that mandatory fields such as :27: (Sequence of Total), :40A: (Form of Documentary Credit), and :31D: (Date and Place of Expiry) are present and correctly formatted. It will also check conditional dependencies, such as ensuring that if field :40A: contains "IRREVOCABLE", then field :42C: is properly included as required by the standards.

The validation process will be comprehensive, covering all aspects of the message structure including: - Field presence validation (mandatory, optional, conditional) - Field format validation (character types, length restrictions) - Field content validation (allowed values, numeric ranges) - Cross-field validation (interdependencies between fields) - Sequence validation (correct ordering of fields)

2.1.2 Message Construction

The system shall enable users to construct valid MT7xx messages through an intuitive interface. This includes:

The message composer will provide a structured form-based interface for users to input field values for specific MT7xx message types. As users input data, the system will dynamically validate entries against field-specific rules and provide immediate feedback on any formatting or content issues. The system will automatically enforce mandatory field requirements and suggest appropriate values for optional fields based on common usage patterns.

The message construction process will guide users through the complex interdependencies between fields, ensuring that conditional fields are properly included or excluded based on the values of related fields. For example, when constructing an

MT707 (Amendment to a Documentary Credit), the system will ensure that the reference to the original documentary credit is properly included and that amendment details are formatted according to SWIFT standards.

Upon completion, the system will generate a properly formatted MT7xx message ready for transmission through SWIFT networks. The constructed message will be available for export in various formats including plain text, structured JSON, and ISO20022 equivalent (where applicable).

2.1.3 Field Dependency Management

The system shall visualize and enforce field dependencies within and across MT7xx messages. This includes:

The field dependency graph will provide a visual representation of how fields within a message are interconnected and how different message types relate to each other in a transaction flow. For example, it will show how an MT707 (Amendment) relates to an original MT700 (Issue of a Documentary Credit) and potentially to subsequent messages like an MT754 (Advice of Payment/Acceptance/Negotiation).

The system will maintain a comprehensive database of field dependencies derived from the SWIFT standards, including: - Intra-message dependencies (fields within the same message) - Inter-message dependencies (fields across different message types) - Conditional requirements based on field values - Business practice recommendations beyond strict SWIFT requirements

This dependency management will ensure that users understand the implications of changes to one field on other fields or messages, reducing errors and improving the quality of SWIFT message processing.

2.1.4 Message Type Support

The system shall support all MT7xx message types as defined in the 2019 Category 7 standards, including but not limited to:

- MT700/MT701: Issue of a Documentary Credit
- MT705: Pre-Advice of a Documentary Credit
- MT707/MT708: Amendment to a Documentary Credit
- MT710/MT711: Advice of a Third Bank's or a Non-Bank's Documentary Credit
- MT720/MT721: Transfer of a Documentary Credit
- MT730: Acknowledgement
- MT732: Advice of Discharge
- MT734: Advice of Refusal
- MT740: Authorization to Reimburse

- MT742: Reimbursement Claim
- MT747: Amendment to an Authorization to Reimburse
- MT750: Advice of Discrepancy
- MT752: Authorization to Pay, Accept or Negotiate
- MT754: Advice of Payment/Acceptance/Negotiation
- MT756: Advice of Reimbursement or Payment
- MT760: Guarantee
- MT767: Amendment to a Guarantee
- MT768: Acknowledgement of a Guarantee Message
- MT769: Advice of Reduction or Release
- MT775: Amendment of a Documentary Credit
- MT785: Notification of Charges
- MT786: Request for Payment of Charges
- MT787: Advice of Charges
- MT790-799: Queries and Free Format Messages

Each message type will have dedicated validation rules, construction templates, and field specifications based on the SWIFT standards.

2.1.5 User Management

The system shall provide user management capabilities to control access and permissions. This includes:

The user management module will allow administrators to create and manage user accounts with different permission levels. Users will be assigned roles that determine their access to system features, such as: - Viewer: Can only view and validate messages - Editor: Can construct and edit messages - Administrator: Has full access to all system features including user management

The system will maintain audit logs of all user actions, including message validation, construction, and modifications to system settings. This will ensure accountability and provide a trail for troubleshooting and compliance purposes.

2.1.6 Reporting and Analytics

The system shall provide reporting and analytics capabilities to track message processing activities. This includes:

The reporting module will generate various reports on message validation and construction activities, including: - Validation error reports highlighting common issues - Usage statistics by message type and user - Trend analysis of message volumes and error rates - Compliance reports for audit purposes

These reports will be available in various formats (PDF, Excel, CSV) and can be scheduled for automatic generation and distribution to relevant stakeholders.

2.2 User Classes and Characteristics

2.2.1 Business Users

Business users are primarily concerned with the construction and validation of SWIFT messages as part of their daily operations. They typically have domain knowledge of trade finance and documentary credits but may not be technical experts in SWIFT message formats. These users require an intuitive interface that guides them through the message construction process and provides clear validation feedback.

2.2.2 Technical Users

Technical users are responsible for integrating the system with other applications and maintaining the system's technical components. They require access to APIs, configuration settings, and technical documentation. These users typically have a deeper understanding of SWIFT message formats and system integration requirements.

2.2.3 Administrators

Administrators are responsible for managing user accounts, system settings, and monitoring system performance. They require access to all system features and administrative functions. Administrators typically have both business and technical knowledge and serve as the bridge between business and technical users.

2.3 Operating Environment

The system shall operate in a web-based environment accessible through standard web browsers. It shall be compatible with the following:

- Web Browsers: Chrome (latest 2 versions), Firefox (latest 2 versions), Edge (latest 2 versions), Safari (latest 2 versions)
- Operating Systems: Windows 10/11, macOS (latest 2 versions), major Linux distributions
- Mobile Devices: Responsive design for tablets and smartphones (iOS and Android)

The system shall be designed to handle concurrent users with minimal performance degradation and shall be available 24/7 with scheduled maintenance windows.

2.4 Design and Implementation Constraints

The system design and implementation shall be constrained by the following factors:

- SWIFT Standards Compliance: The system must strictly adhere to the SWIFT MT7xx message standards as defined in the 2019 Category 7 documentation.
- Security Requirements: The system must implement industry-standard security measures to protect sensitive financial data.
- Performance Requirements: The system must provide real-time validation and feedback for message construction.
- Integration Requirements: The system must provide APIs for integration with existing banking and financial systems.
- Regulatory Compliance: The system must comply with relevant financial regulations and data protection laws.

2.5 User Documentation

The system shall provide comprehensive user documentation, including:

- User Manuals: Detailed instructions for using all system features
- Quick Start Guides: Simplified instructions for common tasks
- Online Help: Context-sensitive help accessible from within the application
- API Documentation: Technical documentation for system integration
- Administrator Guides: Instructions for system configuration and management

2.6 Assumptions and Dependencies

The development and operation of the system are based on the following assumptions and dependencies:

- The SWIFT MT7xx message standards will remain stable during the development period.
- Users will have basic knowledge of SWIFT messages and documentary credits.
- The system will have access to up-to-date SWIFT standards documentation.
- Integration with existing systems will be possible through standard APIs.
- Adequate hardware and network resources will be available for system operation.

3. Technical Specifications (TS)

3.1 System Architecture

3.1.1 Overall Architecture

The SWIFT MT7xx Message Automation System will be implemented using a modern, scalable, multi-tier architecture consisting of the following layers:

The presentation layer will be implemented as a responsive web application using React.js for the frontend. This layer will handle user interactions, form rendering, and client-side validations. The application will communicate with the backend services through RESTful APIs and will implement responsive design principles to ensure usability across different devices and screen sizes.

The business logic layer will be implemented using Node.js for the backend services. This layer will handle the core business logic, including message validation, construction, and dependency management. It will be organized into microservices to allow for independent scaling and maintenance of different system components.

The data access layer will manage interactions with the database and external systems. It will implement data access patterns that ensure efficient and secure data operations while maintaining data integrity and consistency.

The persistence layer will be implemented using SQL Server (hosted on DESKTOP-I3D98TM with Windows authentication) for structured data storage. This will include message templates, validation rules, user data, and audit logs. The database schema will be designed to optimize query performance for common operations while ensuring data integrity through appropriate constraints and relationships.

3.1.2 Component Diagram

The system will consist of the following major components:

1. **Web Application**: React.js-based frontend for user interactions
2. **API Gateway**: Entry point for all client requests, handling authentication and routing
3. **Message Validation Service**: Microservice for validating SWIFT messages
4. **Message Construction Service**: Microservice for building SWIFT messages
5. **Field Dependency Service**: Microservice for managing field interdependencies
6. **User Management Service**: Microservice for user authentication and authorization
7. **Reporting Service**: Microservice for generating reports and analytics
8. **Database**: SQL Server database for persistent storage

9. **Caching Layer**: Redis for performance optimization
10. **Logging and Monitoring**: ELK stack for system monitoring and troubleshooting

These components will interact through well-defined interfaces, allowing for independent development, testing, and deployment.

3.1.3 Deployment Architecture

The system will be deployed using a containerized approach with Docker and Kubernetes for orchestration. This will enable:

- Scalability: Ability to scale individual components based on load
- Resilience: Automatic recovery from component failures
- Portability: Consistent deployment across different environments
- Isolation: Clear separation between components

The deployment will support both on-premises and cloud-based installations, with appropriate configurations for each environment.

3.2 Database Design

3.2.1 Entity-Relationship Diagram

The database will consist of the following primary entities and their relationships:

1. **MessageTypes**: Stores information about different MT7xx message types
2. MessageTypeID (PK)
3. MessageTypeCode (e.g., MT700, MT707)
4. Description
5. Version
6. IsActive
7. **Fields**: Stores information about all possible fields in MT7xx messages
8. FieldID (PK)
9. FieldCode (e.g., :20:, :27:)
10. Name
11. Description
12. Format (e.g., 35x, 4!c)
13. ValidationRegex
14. IsActive
15. **MessageTypeFields**: Maps fields to message types with attributes

16. MessageTypeFieldID (PK)
17. MessageTypeID (FK)
18. FieldID (FK)
19. Sequence
20. IsMandatory
21. IsConditional
22. ConditionExpression

23. MaxOccurrences

24. **FieldDependencies**: Stores dependencies between fields

25. DependencyID (PK)
26. SourceFieldID (FK)
27. TargetFieldID (FK)
28. DependencyType (e.g., Requires, Excludes)
29. ConditionExpression

30. MessageTypeID (FK, optional)

31. **ValidationRules**: Stores validation rules for fields

32. RuleID (PK)
33. FieldID (FK)
34. MessageTypeID (FK, optional)
35. RuleType (e.g., Format, Content, Dependency)
36. RuleExpression
37. ErrorMessage

38. Severity

39. **Users**: Stores user information

40. UserID (PK)
41. Username
42. PasswordHash
43. Email
44. FirstName
45. LastName
46. IsActive

47. LastLoginDate

48. **Roles**: Stores role information

49. RoleID (PK)
50. RoleName
51. Description

52. IsActive

53. **UserRoles**: Maps users to roles

54. UserRoleID (PK)
55. UserID (FK)

56. RoleID (FK)

57. **Permissions**: Stores permission information

58. PermissionID (PK)
59. PermissionName
60. Description

61. IsActive

62. **RolePermissions**: Maps roles to permissions

    - RolePermissionID (PK)
    - RoleID (FK)
    - PermissionID (FK)

63. **AuditLogs**: Stores audit information

    - LogID (PK)
    - UserID (FK)
    - ActionType
    - EntityType
    - EntityID
    - OldValue
    - NewValue
    - Timestamp
    - IPAddress

64. **Messages**: Stores constructed or validated messages

    - MessageID (PK)
    - MessageTypeID (FK)
    - ReferenceNumber
    - Content

    - Status
    - CreatedBy (FK to Users)
    - CreatedDate
    - LastModifiedBy (FK to Users)
    - LastModifiedDate

65. **MessageValidationResults**: Stores validation results
    - ResultID (PK)
    - MessageID (FK)
    - IsValid
    - ValidationDate
    - ValidatedBy (FK to Users)

66. **ValidationErrors**: Stores validation errors
    - ErrorID (PK)
    - ResultID (FK)
    - FieldID (FK)
    - ErrorType
    - ErrorMessage
    - Severity

3.2.2 Database Schema

The database schema will be implemented in SQL Server with appropriate indexes, constraints, and relationships to ensure data integrity and performance. The schema will include:

- Primary and foreign key constraints
- Unique constraints for business keys
- Check constraints for data validation
- Indexes for performance optimization
- Stored procedures for complex operations
- Views for simplified data access
- Triggers for audit logging

3.2.3 Data Migration

The system will include tools for importing SWIFT standards from the official documentation into the database. This will involve:

- Parsing the PDF documentation to extract field specifications
- Mapping fields to message types with appropriate attributes

- Defining validation rules based on the standards
- Setting up field dependencies based on the documentation

This migration process will be automated to the extent possible but may require manual verification and adjustment.

3.3 API Specifications

3.3.1 RESTful API Endpoints

The system will expose the following RESTful API endpoints:

1. **Authentication API**
2. POST /api/auth/login: Authenticate user and return JWT token
3. POST /api/auth/logout: Invalidate user session
4. POST /api/auth/refresh: Refresh JWT token
5. **User Management API**
6. GET /api/users: Get list of users (admin only)
7. GET /api/users/{id}: Get user details
8. POST /api/users: Create new user
9. PUT /api/users/{id}: Update user
10. DELETE /api/users/{id}: Delete user
11. GET /api/users/{id}/roles: Get user roles
12. PUT /api/users/{id}/roles: Update user roles
13. **Message Validation API**
14. POST /api/validate: Validate SWIFT message
15. GET /api/validate/history: Get validation history
16. GET /api/validate/history/{id}: Get validation details
17. **Message Construction API**
18. GET /api/messages/types: Get list of message types
19. GET /api/messages/types/{code}: Get message type details
20. GET /api/messages/types/{code}/fields: Get fields for message type
21. POST /api/messages/construct: Construct SWIFT message
22. GET /api/messages/history: Get construction history
23. GET /api/messages/history/{id}: Get constructed message details

24. **Field Dependency API**

25. GET /api/fields: Get list of fields
26. GET /api/fields/{code}: Get field details
27. GET /api/fields/{code}/dependencies: Get field dependencies

28. GET /api/messages/types/{code}/dependencies: Get message type dependencies

29. **Reporting API**

30. GET /api/reports/validation: Get validation statistics
31. GET /api/reports/usage: Get system usage statistics
32. GET /api/reports/audit: Get audit logs

3.3.2 API Authentication and Security

All API endpoints will be secured using JWT (JSON Web Tokens) for authentication. The authentication flow will be:

1. User logs in with username and password
2. Server validates credentials and returns JWT token
3. Client includes JWT token in Authorization header for subsequent requests
4. Server validates token for each request
5. Token expires after configurable time period
6. Client can refresh token using refresh endpoint

API endpoints will be protected based on user roles and permissions, with appropriate authorization checks for each operation.

3.3.3 API Documentation

The API will be documented using OpenAPI (Swagger) specifications, providing:

- Endpoint descriptions
- Request and response schemas
- Authentication requirements
- Example requests and responses
- Error codes and descriptions

The documentation will be accessible through a Swagger UI interface for interactive exploration and testing.

3.4 Integration Specifications

3.4.1 External System Integration

The system will provide integration capabilities with the following external systems:

1. **Banking Core Systems**: For retrieving customer and account information
2. **SWIFT Alliance Access**: For sending and receiving SWIFT messages
3. **Document Management Systems**: For storing related documents
4. **Workflow Systems**: For integrating with business processes

Integration will be implemented through: - RESTful APIs for synchronous operations - Message queues for asynchronous operations - File-based integration for batch processing

3.4.2 Integration Patterns

The system will implement the following integration patterns:

1. **Request-Response**: For synchronous operations requiring immediate response
2. **Publish-Subscribe**: For event-driven integration
3. **File Transfer**: For batch processing of messages
4. **Database Integration**: For direct database access where appropriate

3.4.3 Error Handling and Retry Mechanisms

Integration points will implement robust error handling and retry mechanisms:

- Transient errors will be retried with exponential backoff
- Persistent errors will be logged and alerted
- Circuit breaker pattern will be implemented to prevent cascading failures
- Dead letter queues will capture failed messages for manual processing

3.5 Security Specifications

3.5.1 Authentication and Authorization

The system will implement a role-based access control (RBAC) model with:

- Multi-factor authentication for sensitive operations
- Fine-grained permissions for system functions
- Role-based access to features and data
- IP-based access restrictions (optional)
- Session timeout and automatic logout

3.5.2 Data Protection

Sensitive data will be protected through:

- Encryption of data at rest (database encryption)
- Encryption of data in transit (TLS/SSL)
- Masking of sensitive data in logs and reports
- Secure handling of credentials and secrets

3.5.3 Audit and Compliance

The system will maintain comprehensive audit logs for:

- User authentication events
- System configuration changes
- Message validation and construction activities
- Data access and modifications

Audit logs will be tamper-proof and will include: - User identification - Timestamp - Action performed - Affected data - IP address and device information

3.6 Performance Specifications

3.6.1 Response Time

The system shall meet the following response time requirements:

- Page load time: < 2 seconds
- Form submission: < 1 second
- Message validation: < 3 seconds for complex messages
- Message construction: < 2 seconds
- Report generation: < 5 seconds for standard reports

3.6.2 Throughput

The system shall support the following throughput requirements:

- Concurrent users: 100 typical, 500 peak
- Message validations: 1000 per hour
- Message constructions: 500 per hour
- API requests: 10,000 per hour

3.6.3 Scalability

The system shall be designed for horizontal scalability:

- Stateless application servers for easy scaling
- Database read replicas for query scaling
- Caching layer for performance optimization
- Load balancing for request distribution

3.6.4 Availability

The system shall meet the following availability requirements:

- 99.9% uptime during business hours
- Scheduled maintenance windows outside business hours
- Failover capabilities for critical components
- Disaster recovery with RPO < 1 hour and RTO < 4 hours

3.7 Monitoring and Logging

3.7.1 Application Monitoring

The system shall implement comprehensive monitoring:

- Real-time performance metrics
- Resource utilization (CPU, memory, disk, network)
- Error rates and exception tracking
- User activity and system usage

3.7.2 Alerting

The system shall provide alerting mechanisms:

- Threshold-based alerts for performance issues
- Error-based alerts for system failures
- Security alerts for suspicious activities
- Capacity planning alerts for resource constraints

3.7.3 Logging

The system shall implement structured logging:

- Application logs with appropriate severity levels
- Access logs for security monitoring
- Performance logs for troubleshooting

- Audit logs for compliance

Logs will be centralized, searchable, and retained according to compliance requirements.

4. UI/UX Design Specifications

4.1 User Interface Overview

4.1.1 Design Philosophy

The UI/UX design for the SWIFT MT7xx Message Automation System follows a user-centered approach that prioritizes clarity, efficiency, and ease of use. The interface is designed to accommodate both novice users who need guidance through the complex process of SWIFT message handling and experienced users who require efficient workflows for high-volume operations.

The design philosophy emphasizes:

The system employs a clean, modern aesthetic with a professional color scheme appropriate for financial applications. The interface uses a consistent visual language throughout, with standardized components, typography, and iconography to create a cohesive user experience. Visual hierarchy is carefully implemented to guide users' attention to the most important elements on each screen, reducing cognitive load and improving task completion rates.

Accessibility is a core consideration, with the interface designed to comply with WCAG 2.1 AA standards. This includes proper color contrast, keyboard navigation support, screen reader compatibility, and responsive design for various devices and screen sizes.

4.1.2 Responsive Design

The system implements a responsive design approach that adapts to different screen sizes and devices. The layout adjusts dynamically to provide optimal viewing and interaction experiences across:

- Desktop computers (1920×1080 and higher)
- Laptops (1366×768 and higher)
- Tablets (portrait and landscape orientations)
- Mobile devices (limited functionality)

The responsive design uses a fluid grid system, flexible images, and CSS media queries to ensure that the interface remains usable and visually appealing across all supported devices.

4.1.3 Navigation Structure

The system employs a hierarchical navigation structure with the following main components:

1. **Top Navigation Bar**:
2. Logo and system name
3. Main navigation menu
4. Search functionality
5. User profile and settings
6. Notifications

7. Help and support

8. **Side Navigation Menu**:

9. Dashboard
10. Message Validation
11. Message Construction
12. Message Templates
13. Field Dependencies
14. Reports and Analytics

15. Administration (for authorized users)

16. **Breadcrumb Navigation**:

17. Shows current location within the application hierarchy

18. Provides quick navigation to parent sections

19. **Context-Sensitive Actions**:

20. Action buttons relevant to the current screen
21. Quick access to common operations

The navigation structure is designed to minimize the number of clicks required to access frequently used features while maintaining a clear organizational hierarchy.

4.2 Key Screens and Workflows

4.2.1 Dashboard

The dashboard serves as the entry point to the system, providing an overview of recent activities and quick access to common tasks.

The dashboard features a clean, card-based layout with the following components:

- **Activity Summary**: Displays statistics on recent message validations and constructions
- **Quick Actions**: Provides one-click access to common tasks such as "Validate Message," "Create New Message," and "View Templates"
- **Recent Messages**: Shows recently validated or constructed messages with status indicators
- **Validation Error Trends**: Displays a chart of common validation errors to help identify recurring issues
- **System Notifications**: Shows important system announcements and updates
- **Favorites**: Allows users to pin frequently used message types or templates for quick access

The dashboard is customizable, allowing users to rearrange cards and hide/show components based on their preferences and workflow needs.

4.2.2 Message Validation Screen

The message validation screen provides a dedicated interface for validating SWIFT MT7xx messages against the standards.

The screen is divided into two main sections:

1. **Input Section**:
2. Large text area for pasting or typing the SWIFT message
3. File upload option for importing messages from files
4. Message type selector (auto-detected from input when possible)
5. Validation options (e.g., strict mode, warning level)

6. "Validate" button prominently displayed

7. **Results Section** (appears after validation):

8. Overall validation status (Valid/Invalid) with color coding
9. Summary of errors and warnings

10. Detailed list of validation issues with:
    - Field reference
    - Error description
    - Severity level
    - Suggested correction
11. Interactive message view with highlighted error locations
12. Option to export validation results
13. Option to switch to message construction to fix errors

The validation process provides real-time feedback as users type or paste messages, with syntax highlighting and inline error indicators to help identify issues before formal validation.

4.2.3 Message Construction Screen

The message construction screen provides a guided interface for building valid SWIFT MT7xx messages from scratch.

The screen features a multi-step workflow:

1. **Message Type Selection**:
2. Grid or list view of available message types
3. Search and filter options
4. Brief description of each message type

5. Selection of commonly used types

6. **Field Input Form**:

7. Dynamic form generated based on the selected message type
8. Fields grouped by logical sections
9. Mandatory fields clearly marked
10. Format hints and examples for each field
11. Real-time validation with visual feedback
12. Conditional fields that appear/disappear based on other field values

13. Auto-completion suggestions for common values

14. **Preview and Validation**:

15. Split-screen view showing the constructed message in SWIFT format
16. Validation summary
17. Option to edit specific fields
18. Option to save as template

19. Export options (text, JSON, etc.)

The construction interface employs progressive disclosure to manage complexity, showing only relevant fields based on previous selections and hiding advanced options until needed.

4.2.4 Field Dependency Visualizer

The field dependency visualizer provides an interactive graphical representation of field relationships within and across message types.

The visualizer features:

- **Interactive Graph**: Nodes representing fields and edges representing dependencies
- **Zoom and Pan Controls**: For navigating complex dependency networks
- **Filtering Options**: To focus on specific message types or fields
- **Dependency Types**: Color-coded edges for different types of dependencies (requires, excludes, etc.)
- **Details Panel**: Shows detailed information about selected fields or dependencies
- **Search Functionality**: To quickly locate specific fields
- **Export Options**: To save visualizations for documentation or reference

The visualizer helps users understand the complex interdependencies in SWIFT messages, improving their ability to construct valid messages and troubleshoot validation issues.

4.2.5 Template Management

The template management screen allows users to create, edit, and manage message templates for frequent use cases.

The screen includes:

- **Template Library**: Grid or list view of saved templates
- **Categorization**: Grouping templates by message type, purpose, or custom categories
- **Search and Filter**: To quickly find specific templates
- **Template Editor**: For creating and modifying templates
- **Variable Fields**: Support for defining placeholder fields that can be filled in when using the template
- **Sharing Options**: For team collaboration on templates
- **Version History**: To track changes to templates over time

Templates streamline the message construction process for recurring scenarios, reducing errors and improving efficiency.

4.2.6 Administration Console

The administration console provides authorized users with tools for managing system settings, users, and permissions.

The console includes:

- **User Management**: Interface for creating, editing, and deactivating user accounts
- **Role Management**: Tools for defining and assigning user roles
- **Permission Settings**: Fine-grained control over feature access
- **System Configuration**: Settings for system behavior and defaults
- **Audit Logs**: Review of system activity for security and compliance
- **Import/Export Tools**: For bulk operations on system data
- **Maintenance Tools**: For database operations and system updates

The administration console employs a clear, tabular layout for data management with appropriate safeguards for critical operations.

4.3 Interaction Design

4.3.1 Input Methods

The system supports multiple input methods to accommodate different user preferences and scenarios:

- **Direct Text Entry**: For typing messages or field values
- **File Upload**: For importing messages from external sources
- **Copy-Paste**: For transferring content from other applications
- **Form-Based Input**: Structured forms for guided data entry
- **Template-Based**: Pre-filled forms based on saved templates
- **Drag-and-Drop**: For file uploads and reordering items
- **Keyboard Shortcuts**: For power users to improve efficiency

Input methods are designed to be intuitive and forgiving, with appropriate validation and error recovery mechanisms.

4.3.2 Feedback Mechanisms

The system provides clear feedback to users through various channels:

- **Visual Feedback**: Color coding, icons, and animations to indicate status

- **Textual Feedback**: Clear error messages and success confirmations
- **Inline Validation**: Real-time feedback as users enter data
- **Toast Notifications**: Temporary messages for non-critical information
- **Modal Dialogs**: For important confirmations or errors requiring attention
- **Progress Indicators**: For long-running operations
- **Status Updates**: For background processes

Feedback is designed to be timely, relevant, and actionable, helping users understand system status and recover from errors.

4.3.3 Error Handling

The system implements a comprehensive approach to error handling in the user interface:

- **Prevention**: Guiding users to avoid errors through clear instructions and constraints
- **Detection**: Identifying errors as early as possible through validation
- **Correction**: Providing specific guidance on how to fix errors
- **Recovery**: Maintaining user progress and context when errors occur

Error messages are written in plain language, avoiding technical jargon, and clearly explain: - What went wrong - Why it happened - How to fix it

The system distinguishes between different severity levels (errors, warnings, suggestions) with appropriate visual treatment for each.

4.4 Visual Design

4.4.1 Color Palette

The system employs a professional color palette with the following components:

- **Primary Colors**: Deep blue (#1A365D) and complementary teal (#00A3B4) for primary actions and branding
- **Secondary Colors**: Amber (#FFC107) and coral (#FF6B6B) for notifications and warnings
- **Neutral Colors**: Various shades of gray for backgrounds, text, and UI elements
- **Semantic Colors**: Green (#28A745) for success, red (#DC3545) for errors, yellow (#FFC107) for warnings
- **Accent Colors**: Used sparingly for emphasis and visual interest

The color palette is designed for high contrast and accessibility, with careful attention to color combinations that work for users with color vision deficiencies.

4.4.2 Typography

The system uses a clear, readable typography system:

- **Primary Font**: Open Sans for general UI text
- **Secondary Font**: Roboto Mono for code and message content
- **Font Sizes**: Ranging from 12px for small text to 24px for main headings
- **Font Weights**: Regular (400) for body text, Semi-bold (600) for emphasis, Bold (700) for headings
- **Line Heights**: 1.5 for body text, 1.2 for headings
- **Text Colors**: Dark gray (#333333) for primary text, medium gray (#666666) for secondary text

The typography system is implemented with a responsive approach, adjusting sizes and spacing for different screen sizes.

4.4.3 Iconography

The system uses a consistent icon set to enhance visual communication:

- **Functional Icons**: For actions and navigation (e.g., save, delete, search)
- **Status Icons**: To indicate state (e.g., valid, invalid, warning)
- **Message Type Icons**: Visual identifiers for different message types
- **Field Type Icons**: Visual cues for field formats and requirements

Icons are implemented as SVGs for crisp rendering at all sizes and are accompanied by text labels where appropriate for clarity.

4.4.4 Layout and Grid System

The system employs a 12-column grid system for consistent layout across screens:

- **Margins**: Consistent outer margins (24px on desktop, 16px on tablet, 8px on mobile)
- **Gutters**: 16px spacing between columns
- **Containers**: Maximum width constraints for different screen sizes
- **Breakpoints**: Defined at 576px, 768px, 992px, and 1200px for responsive design
- **Card Components**: For grouping related content with consistent padding and styling
- **Vertical Rhythm**: Consistent spacing multiples (8px base unit) for harmonious vertical spacing

The layout system ensures visual consistency while allowing flexibility for different content types and screen sizes.

4.5 Accessibility Considerations

4.5.1 WCAG Compliance

The system is designed to comply with WCAG 2.1 AA standards, including:

- **Perceivable**: Content is presentable to users in ways they can perceive
- Text alternatives for non-text content
- Captions and alternatives for multimedia
- Content that can be presented in different ways
- Content that is distinguishable (color, contrast, etc.)
- **Operable**: User interface components are operable
- Keyboard accessibility for all functionality
- Sufficient time to read and use content
- No content that could cause seizures
- Navigable content with multiple ways to find pages
- **Understandable**: Information and operation are understandable
- Readable and understandable text
- Predictable operation and appearance
- Input assistance to help users avoid and correct mistakes
- **Robust**: Content is robust enough to be interpreted by various user agents
- Compatible with current and future user tools

4.5.2 Keyboard Navigation

The system supports comprehensive keyboard navigation:

- **Tab Order**: Logical and predictable tab sequence
- **Focus Indicators**: Clear visual indication of focused elements
- **Keyboard Shortcuts**: For common actions
- **Skip Links**: To bypass repetitive navigation
- **Modal Trapping**: Focus management for modal dialogs

4.5.3 Screen Reader Support

The system implements proper semantic markup and ARIA attributes:

- **Semantic HTML**: Using appropriate HTML elements for their intended purpose

- **ARIA Landmarks**: To identify regions of the page
- **ARIA Labels**: For elements without visible text
- **ARIA Live Regions**: For dynamic content updates
- **Alternative Text**: For images and icons

4.5.4 Color and Contrast

The system ensures sufficient color contrast and does not rely solely on color for conveying information:

- **Text Contrast**: Minimum 4.5:1 ratio for normal text, 3:1 for large text
- **UI Component Contrast**: Minimum 3:1 ratio for interactive elements
- **Secondary Indicators**: Icons, patterns, or text labels in addition to color
- **High Contrast Mode**: Support for operating system high contrast settings

4.6 Usability Testing

4.6.1 Testing Methodology

The UI/UX design will be validated through a comprehensive usability testing program:

- **Heuristic Evaluation**: Expert review against established usability principles
- **Cognitive Walkthrough**: Step-by-step analysis of user tasks
- **User Testing**: Observation of representative users performing realistic tasks
- **A/B Testing**: Comparison of alternative designs for key interfaces
- **Accessibility Audit**: Formal evaluation of accessibility compliance

4.6.2 Key Metrics

Usability will be measured using the following metrics:

- **Task Success Rate**: Percentage of users who complete tasks successfully
- **Time on Task**: Average time to complete specific tasks
- **Error Rate**: Frequency of user errors during task completion
- **Satisfaction Rating**: User-reported satisfaction with the interface
- **System Usability Scale (SUS)**: Standardized measure of perceived usability
- **Net Promoter Score (NPS)**: Likelihood of users to recommend the system

4.6.3 Iterative Improvement

The UI/UX design will evolve based on testing results and user feedback:

- **Prioritized Issues**: Ranking of identified usability problems by severity
- **Design Iterations**: Revisions to address identified issues

- **Validation Testing**: Confirmation that changes resolve the original problems
- **Continuous Improvement**: Ongoing monitoring and refinement of the user experience

The iterative approach ensures that the final design meets user needs and expectations while supporting efficient and error-free task completion.

5. Business Rules for Message Fields

5.1 Business Rules Engine Overview

5.1.1 Purpose and Scope

The Business Rules Engine is a critical component of the SWIFT MT7xx Message Automation System that enforces the complex business logic governing SWIFT message fields. This engine ensures that messages not only conform to the syntactic requirements of the SWIFT standards but also adhere to semantic rules and business practices specific to documentary credits and guarantees.

The Business Rules Engine serves as the central repository for all validation and business logic, providing a consistent and maintainable approach to rule enforcement. By separating business rules from application code, the system enables business analysts and domain experts to manage rules without requiring developer intervention, improving agility and reducing maintenance costs.

The engine encompasses rules for field validation, inter-field dependencies, cross-message validation, and business practice recommendations. It supports both deterministic rules (with clear yes/no outcomes) and probabilistic rules (with confidence scores for potential issues), allowing for different levels of validation stringency based on user preferences and use cases.

5.1.2 Rule Categories

The Business Rules Engine implements several categories of rules:

1. **Field Format Rules**: Ensure that field content adheres to the specified format (e.g., character types, length restrictions)
2. **Field Content Rules**: Validate the semantic correctness of field values (e.g., valid currency codes, date formats)
3. **Mandatory Field Rules**: Enforce the presence of required fields based on message type and context

4. **Conditional Field Rules**: Manage the complex dependencies between fields within a message
5. **Cross-Field Validation Rules**: Ensure consistency between related fields (e.g., amounts, dates, references)
6. **Cross-Message Rules**: Validate relationships between different message types in a transaction flow
7. **Business Practice Rules**: Implement recommended practices beyond strict SWIFT requirements

Each rule category serves a specific purpose in ensuring message validity and compliance with both technical standards and business expectations.

5.1.3 Rule Implementation Approach

The Business Rules Engine employs a hybrid approach to rule implementation:

1. **Declarative Rules**: Simple validation rules expressed in a domain-specific language or configuration
2. **Procedural Rules**: Complex validation logic implemented as code functions
3. **Decision Tables**: Matrix-based rules for scenarios with multiple conditions and outcomes
4. **Rule Flows**: Sequential or branching rule execution for complex validation scenarios

This hybrid approach balances flexibility, performance, and maintainability, allowing for the most appropriate implementation method based on rule complexity and execution requirements.

5.2 Database-Driven Rule Management

5.2.1 Rule Storage and Organization

Business rules are stored in the database using a structured schema that enables efficient rule management and execution:

1. **Rules Table**: Central repository of all business rules
2. RuleID (PK)
3. RuleName
4. Description
5. RuleType (Format, Content, Dependency, etc.)
6. Severity (Error, Warning, Info)
7. IsActive
8. CreatedBy

9. CreatedDate
10. LastModifiedBy
11. LastModifiedDate
12. **RuleConditions Table**: Stores conditions for rule activation
13. ConditionID (PK)
14. RuleID (FK)
15. FieldID (FK)
16. OperatorType (Equals, NotEquals, Contains, etc.)
17. ComparisonValue
18. LogicalOperator (AND, OR)
19. SequenceNumber
20. **RuleActions Table**: Defines actions to take when rule conditions are met
21. ActionID (PK)
22. RuleID (FK)
23. ActionType (Validate, Require, Exclude, etc.)
24. TargetFieldID (FK)
25. Parameters (JSON)
26. ErrorMessage
27. SequenceNumber
28. **RuleScopes Table**: Defines the applicability scope of rules
29. ScopeID (PK)
30. RuleID (FK)
31. MessageTypeID (FK)
32. ApplicabilityExpression

This database structure allows for flexible rule definition, efficient rule execution, and comprehensive rule management capabilities.

5.2.2 Rule Versioning and History

The system maintains a complete history of rule changes to support audit requirements and enable rollback if needed:

1. **RuleVersions Table**: Stores historical versions of rules
2. VersionID (PK)
3. RuleID (FK)

4. VersionNumber
5. RuleDefinition (JSON)
6. EffectiveFrom
7. EffectiveTo
8. ChangedBy
9. ChangeReason
10. **RuleChangeLog Table**: Records all modifications to rules
11. LogID (PK)
12. RuleID (FK)
13. ChangeType (Create, Update, Deactivate)
14. ChangedBy
15. ChangeDate
16. OldValue
17. NewValue
18. ChangeReason

This versioning approach ensures traceability and compliance with regulatory requirements while supporting business continuity through rule evolution.

5.2.3 Rule Administration Interface

The system provides a dedicated interface for rule management:

1. **Rule Explorer**: Browse and search existing rules
2. **Rule Editor**: Create and modify rules through a visual interface
3. **Rule Tester**: Validate rules against sample data
4. **Rule Analyzer**: Identify rule conflicts and dependencies
5. **Rule Deployment**: Control the activation and deactivation of rules
6. **Rule Import/Export**: Exchange rules between environments

The administration interface empowers business analysts to manage rules without developer intervention, improving agility and reducing maintenance costs.

5.3 Field-Specific Business Rules

5.3.1 Common Field Rules

Certain business rules apply across multiple message types and fields:

1. **Reference Number (Field :20:)**
2. Must be unique within the sender's system

3. Cannot contain spaces or special characters except slashes and hyphens
4. Maximum length of 16 characters
5. Should follow the institution's reference numbering convention

6. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('ReferenceNumberFormat', 'Validates format of reference numbers', 'Format', 'Error');

   INSERT INTO RuleConditions (RuleID, FieldID, OperatorType, ComparisonValue) VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode = ':20:'), 'Matches', '^[A-Za-z0-9/-]{1,16}$'); ```

7. **Date Fields (Fields :31D:, :42C:, etc.)**

8. Must follow the format YYMMDD
9. Must represent valid calendar dates
10. Future dates must be within reasonable business timeframes
11. Expiry dates must be after issue dates

12. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('DateFormat', 'Validates format of date fields', 'Format', 'Error');

    INSERT INTO RuleConditions (RuleID, FieldID, OperatorType, ComparisonValue) VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode LIKE '%Date%'), 'Matches', '^[0-9]{6}$');

    INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('DateValidity', 'Ensures dates are valid calendar dates', 'Content', 'Error');

    -- Additional implementation for date validity check would be in procedural code ```

13. **Amount Fields (Fields :32B:, :39A:, etc.)**

14. Currency code must be valid ISO currency code
15. Amount must follow the format of digits with decimal comma
16. Negative amounts are not allowed
17. Tolerance percentages must be between 0 and 100

18. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('CurrencyCodeValidity', 'Validates currency codes in amount fields', 'Content', 'Error');

INSERT INTO RuleConditions (RuleID, FieldID, OperatorType, ComparisonValue) VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode LIKE '%Amount%'), 'InList', 'USD,EUR,GBP,JPY,CHF,AUD,CAD,CNY');

-- Additional rules for amount format and range would be similarly implemented
```

19. **Free Text Fields (Fields :45A:, :46A:, etc.)**

20. Cannot contain certain special characters
21. Line length restrictions apply
22. Total field length restrictions apply
23. Certain keywords may trigger warnings based on business context

24. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('FreeTextCharacters', 'Validates characters in free text fields', 'Format', 'Error');

    INSERT INTO RuleConditions (RuleID, FieldID, OperatorType, ComparisonValue) VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode IN (':45A:', ':46A:')), 'DoesNotContain', '|~^{}[]`\'); ```

### 5.3.2 MT700 Specific Rules

The MT700 (Issue of a Documentary Credit) message type has specific business rules:

1. **Form of Documentary Credit (Field :40A:)**
2. Must contain one of the allowed values: "IRREVOCABLE", "IRREVOCABLE TRANSFERABLE"
3. If "IRREVOCABLE TRANSFERABLE", then field :42C: becomes mandatory

4. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('MT700_40A_AllowedValues', 'Validates allowed values for form of documentary credit', 'Content', 'Error');

   INSERT INTO RuleConditions (RuleID, FieldID, OperatorType, ComparisonValue) VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode = ':40A:'), 'InList', 'IRREVOCABLE,IRREVOCABLE TRANSFERABLE');

   INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('MT700_TransferableRequires42C', 'Requires field 42C when credit is transferable', 'Dependency', 'Error');
```

INSERT INTO RuleConditions (RuleID, FieldID, OperatorType, ComparisonValue) VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode = ':40A:'), 'Equals', 'IRREVOCABLE TRANSFERABLE');

INSERT INTO RuleActions (RuleID, ActionType, TargetFieldID, ErrorMessage) VALUES (@@IDENTITY, 'Require', (SELECT FieldID FROM Fields WHERE FieldCode = ':42C:'), 'Field :42C: is required when credit is transferable'); ```

5. **Available With (Field :41A:, :41D:)**

6. Either :41A: or :41D: must be present, but not both
7. If :41A: contains "BY ACCEPTANCE", then field :42C: becomes mandatory

8. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('MT700_41A_41D_Exclusivity', 'Ensures only one of 41A or 41D is present', 'Dependency', 'Error');

   INSERT INTO RuleConditions (RuleID, FieldID, OperatorType, ComparisonValue) VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode = ':41A:'), 'Exists', 'TRUE');

   INSERT INTO RuleActions (RuleID, ActionType, TargetFieldID, ErrorMessage) VALUES (@@IDENTITY, 'Exclude', (SELECT FieldID FROM Fields WHERE FieldCode = ':41D:'), 'Fields :41A: and :41D: cannot both be present');

   -- Additional rule for BY ACCEPTANCE requirement would be similarly implemented ```

9. **Shipment Period (Fields :44C:, :44D:)**

10. If present, shipment date or period must be before expiry date
11. Latest shipment date must be after earliest shipment date if both are specified

12. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('MT700_ShipmentBeforeExpiry', 'Ensures shipment is before expiry', 'CrossField', 'Error');

    -- This complex rule would likely be implemented as a procedural rule with custom code ```

5.3.3 MT707 Specific Rules

The MT707 (Amendment to a Documentary Credit) message type has specific business rules:

1. **Amendment Number (Field :26E:)**

2. Must be sequential relative to previous amendments
3. Must be numeric

4. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('MT707_26E_Numeric', 'Ensures amendment number is numeric', 'Format', 'Error');

   INSERT INTO RuleConditions (RuleID, FieldID, OperatorType, ComparisonValue) VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode = ':26E:'), 'Matches', '^[0-9]+$');

   -- Sequential validation would require procedural code with database lookup ```

5. **Amendment Details (Field :77A:)**

6. Must contain specific amendment instructions
7. Cannot contradict other fields in the message
8. Certain keywords trigger validation against specific fields

9. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('MT707_77A_Keywords', 'Identifies keywords in amendment details', 'Content', 'Warning');

   INSERT INTO RuleConditions (RuleID, FieldID, OperatorType, ComparisonValue) VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode = ':77A:'), 'Contains', 'AMOUNT');

   INSERT INTO RuleActions (RuleID, ActionType, Parameters, ErrorMessage) VALUES (@@IDENTITY, 'Validate', '{"checkField": ":32B:"}', 'Amendment mentions AMOUNT but field :32B: is not present'); ```

10. **Reference to Original DC (Field :21:)**

11. Must reference an existing documentary credit
12. Referenced DC must not be expired

13. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('MT707_21_ExistingReference', 'Validates reference to existing DC', 'CrossMessage', 'Error');

    -- This would require procedural code with database lookup to validate against existing messages ```

5.3.4 Cross-Message Rules

Certain business rules span multiple message types:

1. **Amendment Consistency**
2. MT707 amendments must be consistent with the original MT700
3. Cumulative amendments must not contradict each other

4. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('AmendmentConsistency', 'Ensures amendments are consistent with original DC', 'CrossMessage', 'Error');

   -- This complex rule would be implemented as a procedural rule with custom code ```

5. **Message Sequence**

6. MT754 (Advice of Payment) must reference a valid MT700 or MT707
7. MT799 (Free Format) related to a DC must reference an existing DC

8. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('MessageSequence', 'Validates proper sequencing of related messages', 'CrossMessage', 'Error');

   -- This would require procedural code with database lookup ```

9. **Transaction Completeness**

10. All required messages in a transaction flow must be present
11. Messages must be in the correct chronological order

12. Database implementation: ```sql INSERT INTO Rules (RuleName, Description, RuleType, Severity) VALUES ('TransactionCompleteness', 'Ensures all required messages in a flow are present', 'CrossMessage', 'Warning');

    -- This would require procedural code with transaction analysis logic ```

5.4 Rule Execution Framework

5.4.1 Rule Execution Process

The Business Rules Engine follows a structured process for rule execution:

1. **Rule Selection**: Identify applicable rules based on message type and context
2. **Rule Sorting**: Order rules by priority and dependencies
3. **Condition Evaluation**: Evaluate rule conditions against message data

4. **Action Execution**: Perform actions for rules with satisfied conditions
5. **Result Aggregation**: Collect and categorize validation results
6. **Result Reporting**: Format and return validation results to the caller

This process ensures efficient and consistent rule execution while supporting complex validation scenarios.

5.4.2 Performance Optimization

The rule execution framework implements several optimizations:

1. **Rule Indexing**: Rules are indexed by message type and field for quick lookup
2. **Condition Caching**: Frequently used condition results are cached
3. **Lazy Evaluation**: Rules are evaluated only when needed
4. **Parallel Execution**: Independent rules are executed in parallel
5. **Early Termination**: Rule execution stops when critical errors are found
6. **Result Caching**: Validation results are cached for repeated validations

These optimizations ensure that rule execution remains performant even with large rule sets and complex messages.

5.4.3 Extensibility Mechanisms

The rule execution framework provides several mechanisms for extending the rule system:

1. **Custom Rule Types**: Define new rule types for specialized validation
2. **External Rule Sources**: Integrate rules from external systems or standards
3. **Rule Plugins**: Add custom rule implementations through a plugin architecture
4. **Rule Templates**: Create reusable rule patterns for common validation scenarios
5. **Rule Inheritance**: Define hierarchical relationships between rules

These extensibility mechanisms ensure that the rule system can adapt to evolving business requirements and integration needs.

5.5 Business Rule Examples

5.5.1 Simple Validation Rule

A simple validation rule for checking the format of a reference number:

```
 -- Rule definition
INSERT INTO Rules (RuleName, Description, RuleType, Severity)
VALUES ('ReferenceNumberFormat', 'Validates format of reference
numbers', 'Format', 'Error');
```

```
-- Rule condition
INSERT INTO RuleConditions (RuleID, FieldID, OperatorType,
ComparisonValue)
VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode
= ':20:'), 'Matches', '^[A-Za-z0-9/\-]{1,16}$');

-- Rule action
INSERT INTO RuleActions (RuleID, ActionType, ErrorMessage)
VALUES (@@IDENTITY, 'Validate', 'Reference number must be 1-16
alphanumeric characters, slashes, or hyphens');
```

This rule ensures that reference numbers follow the required format, providing clear error messages when violations are detected.

5.5.2 Conditional Field Rule

A conditional rule that requires field :42C: when field :40A: contains "IRREVOCABLE TRANSFERABLE":

```
 -- Rule definition
INSERT INTO Rules (RuleName, Description, RuleType, Severity)
VALUES ('MT700_TransferableRequires42C', 'Requires field 42C
when credit is transferable', 'Dependency', 'Error');

-- Rule condition
INSERT INTO RuleConditions (RuleID, FieldID, OperatorType,
ComparisonValue)
VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode
= ':40A:'), 'Equals', 'IRREVOCABLE TRANSFERABLE');

-- Rule action
INSERT INTO RuleActions (RuleID, ActionType, TargetFieldID,
ErrorMessage)
VALUES (@@IDENTITY, 'Require', (SELECT FieldID FROM Fields
WHERE FieldCode = ':42C:'),
'Field :42C: is required when credit is transferable');
```

This rule enforces the dependency between the form of documentary credit and the draft at field, ensuring that all required fields are present based on context.

5.5.3 Cross-Field Validation Rule

A cross-field validation rule that ensures the expiry date is after the issue date:

```
 -- Rule definition
INSERT INTO Rules (RuleName, Description, RuleType, Severity)
```

```
VALUES ('ExpiryAfterIssue', 'Ensures expiry date is after issue
date', 'CrossField', 'Error');

-- Rule conditions
INSERT INTO RuleConditions (RuleID, FieldID, OperatorType,
ComparisonValue, LogicalOperator, SequenceNumber)
VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode
= ':31D:'), 'Exists', 'TRUE', 'AND', 1);

INSERT INTO RuleConditions (RuleID, FieldID, OperatorType,
ComparisonValue, LogicalOperator, SequenceNumber)
VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode
= ':31C:'), 'Exists', 'TRUE', 'AND', 2);

-- Rule action (simplified - actual implementation would use
procedural code)
INSERT INTO RuleActions (RuleID, ActionType, Parameters,
ErrorMessage)
VALUES (@@IDENTITY, 'ValidateDates', '{"dateField1": ":31D:",
"dateField2": ":31C:", "comparison": "greaterThan"}', 'Expiry
date must be after issue date');
```

This rule ensures logical consistency between date fields, preventing invalid temporal relationships in the message.

5.5.4 Complex Business Rule

A complex business rule for validating amendment consistency:

```
 -- Rule definition
INSERT INTO Rules (RuleName, Description, RuleType, Severity)
VALUES ('AmendmentConsistency', 'Ensures amendments are
consistent with original DC', 'CrossMessage', 'Error');

-- Rule conditions
INSERT INTO RuleConditions (RuleID, FieldID, OperatorType,
ComparisonValue)
VALUES (@@IDENTITY, (SELECT FieldID FROM Fields WHERE FieldCode
= ':21:'), 'Exists', 'TRUE');

-- Rule action (simplified - actual implementation would use
procedural code)
INSERT INTO RuleActions (RuleID, ActionType, Parameters,
ErrorMessage)
VALUES (@@IDENTITY, 'ValidateAmendment', '{"checkOriginal":
true, "checkPreviousAmendments": true}', 'Amendment is
inconsistent with original documentary credit or previous
amendments');
```

This rule would be implemented primarily through procedural code that performs complex validation logic across multiple messages, ensuring the integrity of the amendment process.

5.6 Rule Management Lifecycle

5.6.1 Rule Development Process

The development of business rules follows a structured process:

1. **Requirements Gathering**: Identify business requirements and validation needs
2. **Rule Analysis**: Analyze requirements and translate them into specific rules
3. **Rule Design**: Design rule structure, conditions, and actions
4. **Rule Implementation**: Implement rules in the database or code
5. **Rule Testing**: Test rules against sample data and edge cases
6. **Rule Documentation**: Document rule purpose, behavior, and implementation
7. **Rule Deployment**: Deploy rules to the production environment

This process ensures that rules accurately reflect business requirements and are properly implemented and tested.

5.6.2 Rule Governance

The management of business rules is governed by the following principles:

1. **Ownership**: Each rule has a designated business owner responsible for its accuracy
2. **Change Control**: Changes to rules follow a formal approval process
3. **Version Control**: Rule versions are tracked and managed
4. **Impact Analysis**: Changes are evaluated for their impact on existing processes
5. **Compliance Review**: Rules are reviewed for compliance with standards and regulations
6. **Performance Monitoring**: Rule execution performance is monitored and optimized

These governance principles ensure that the rule system remains accurate, compliant, and performant over time.

5.6.3 Rule Maintenance

The ongoing maintenance of business rules includes:

1. **Regular Review**: Periodic review of rules for continued relevance
2. **Standards Updates**: Updates based on changes to SWIFT standards
3. **Performance Tuning**: Optimization of rule execution performance

4. **Bug Fixing**: Correction of identified issues in rule behavior
5. **Enhancement**: Addition of new rules based on evolving requirements
6. **Deprecation**: Controlled retirement of obsolete rules

This maintenance approach ensures that the rule system evolves with changing business needs and technical capabilities while maintaining stability and reliability.

6. Conclusion and Implementation Recommendations

6.1 Summary of Requirements

The SWIFT MT7xx Message Automation System as specified in this SRS document provides a comprehensive solution for handling SWIFT Category 7 Documentary Credits and Guarantees messages according to the 2019 standards. The system encompasses several key components:

1. **Message Validation Engine**: For verifying compliance with SWIFT standards
2. **Message Construction Interface**: For building valid MT7xx messages
3. **Field Dependency Management**: For visualizing and enforcing field relationships
4. **Business Rules Engine**: For implementing complex validation logic
5. **User Interface**: For intuitive interaction with the system

The system is designed to meet the needs of various user classes, including business users who work with SWIFT messages daily, technical users responsible for system integration, and administrators who manage the system. The requirements specified in this document ensure that the system will be usable, reliable, secure, and maintainable.

6.2 Implementation Approach

6.2.1 Phased Implementation

A phased implementation approach is recommended for this system:

Phase 1: Core Functionality - Message validation for most common MT7xx types (MT700, MT707) - Basic message construction interface - Essential business rules - User management and authentication

Phase 2: Extended Functionality - Support for all MT7xx message types - Advanced message construction with templates - Comprehensive business rules - Reporting and analytics

Phase 3: Integration and Enhancement - External system integration - Advanced visualization features - Performance optimization - Mobile support

This phased approach allows for early delivery of core value while managing implementation complexity.

6.2.2 Development Methodology

An Agile development methodology is recommended for this project:

- **Sprint Planning**: 2-week sprints with defined deliverables
- **User Stories**: Requirements broken down into actionable user stories
- **Continuous Integration**: Automated build and test processes
- **Regular Demos**: Stakeholder reviews at the end of each sprint
- **Iterative Refinement**: Continuous improvement based on feedback

This approach ensures that the development remains aligned with business needs and can adapt to changing requirements.

6.3 Testing Strategy

6.3.1 Testing Levels

A comprehensive testing strategy should include:

- **Unit Testing**: Testing individual components in isolation
- **Integration Testing**: Testing interactions between components
- **System Testing**: Testing the complete system against requirements
- **Performance Testing**: Verifying system performance under load
- **Security Testing**: Identifying and addressing security vulnerabilities
- **User Acceptance Testing**: Validation by end users

6.3.2 Test Data

Test data should include:

- **Sample Messages**: Representative examples of all MT7xx message types
- **Edge Cases**: Messages that test boundary conditions and unusual scenarios
- **Invalid Messages**: Messages with various types of errors to test validation
- **Real-World Data**: Anonymized production data for realistic testing

6.4 Deployment Considerations

6.4.1 Environment Requirements

The system should be deployed in environments that meet the following requirements:

- **Hardware**: Servers with sufficient CPU, memory, and storage

- **Software**: Required operating systems, databases, and middleware
- **Network**: Adequate bandwidth and security measures
- **Monitoring**: Tools for system health and performance monitoring
- **Backup**: Regular backup procedures and disaster recovery capabilities

6.4.2 Training and Support

A comprehensive training and support plan should include:

- **User Training**: Training sessions for different user roles
- **Administrator Training**: Specialized training for system administrators
- **Documentation**: Comprehensive user and technical documentation
- **Help Desk**: Support resources for user assistance
- **Knowledge Base**: Repository of common issues and solutions

6.5 Future Enhancements

Potential future enhancements to consider:

1. **ISO20022 Integration**: Support for mapping between MT7xx and ISO20022 formats
2. **Machine Learning**: Intelligent suggestions for message construction
3. **Blockchain Integration**: Support for documentary credits on blockchain platforms
4. **Mobile Applications**: Dedicated mobile apps for on-the-go access
5. **API Marketplace**: Expanded API offerings for third-party integration
6. **Advanced Analytics**: Business intelligence features for message trends and patterns

These enhancements can be evaluated and prioritized based on business value and technical feasibility after the core system is implemented.