Maude Strategies-Based SoSs Workflow Modeling

Charaf Eddine Dridi[1,2][a], Nabil Hameurlain[1][b] and Faiza Belala[2][c]

[1] *LIUPPA Laboratory, University of Pau, Pau, France*

[2] *LIRE Laboratory, Constantine 2 University – Abdelhamid Mehri, Constantine, Algeria*

Keywords: SoS, Missions, Management Strategies, Workflow, Maude Language, Formal Modeling and Verification.

Abstract: Systems of Systems (SoSs) are large-scale, complex, and occasionally critical software systems. They emerge from the integration of autonomous, heterogeneous, and evolving Constituent Systems (CSs) that collaborate to fulfill operational missions and provide functionalities that exceed those of individual systems. The modeling, simulation, and analysis of SoSs are challenging due to complexities such as temporal constraints on missions and the emergence of both desired and unwanted behaviors within CSs. In this paper, we propose a formal-based solution to specify and validate the functional behavior of SoSs. We employ the Maude Strategy language, a rewriting logic-based language to define the operational semantics of these systems. This includes implementing a set of strategies for managing mission execution, which aim to enhance the SoS workflow by avoiding undesirable behaviors and promoting desirable ones. Our approach offers an executable solution for these strategies and validates the SoS behavior using model-checking techniques provided by Maude. To demonstrate its applicability, the approach is illustrated with a case study of a French Emergency SoS.

1 INTRODUCTION

Systems-of-Systems (SoSs) have experienced significant attention from the computer science community due to their evolutionary progress and ability to integrate multiple Constituent Systems (CSs) from diverse domains, including healthcare, military defense, energy grids, and emergency management. An SoS consists of distributed and complex CSs that collaborate within a network structure, leveraging their diverse physical and functional characteristics to achieve a global capability that surpasses what each CS could achieve individually (Maier, 1998).

1.1 Context and Problematic

SoSs can take four different types (Maier, 1998): Directed (with a centrally managed purpose and central ownership of all CSs), Virtual (lack of central purpose and central alignment), Collaborative (with voluntary interactions of CSs to achieve a goal), and Acknowledged (independent ownership of the CSs). Subsequently, the definition of SoSs is introduced based on the key consideration that SoSs are more than simply a set of connected CSs sharing data and offering missions; it also defines the logical structure and behavior of qualitative/quantitative features involved in SoSs, whose CSs can have their operational/managerial independence and whose emergent behavior is aligned with missions' execution.

In time-aware missioned SoSs, two critical aspects based on quantitative priorities can govern the success of mission achievement: temporal constraints and self-management of workflows (Dridi et al., 2022)(Dridi et al., 2023)(Halima et al., 2021)(Maamar et al., 2016):

- The first aspect addresses the complexities of time constraints and explores how various time-related properties (e.g., mission duration, deadlines, etc.) affect the overall execution of each mission. Especially in critical and emergency applications, missions of various CSs are required to be executed in real-time and accomplished within specific amounts of time. This property influences the starting, execution, and completion time of missions, where a violated constraint can result in a disaster.
- The second aspect self-manages the Workflows (WFs) of missions. This requires analyzing, planning and executing the series of missions (or func-

[a] https://orcid.org/0000-0001-5724-8187
[b] https://orcid.org/0000-0003-3311-4146
[c] https://orcid.org/0000-0002-4563-4061

Dridi, C. E., Hameurlain, N. and Belala, F.
Maude Strategies-Based SoSs Workflow Modeling.
DOI: 10.5220/0013481800003928
In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 316-327
ISBN: 978-989-758-742-9; ISSN: 2184-4895

tional chains of missions) required to reach a specific mission. The goal of workflow management is to ensure that the global mission of an SoS is completed correctly, consistently, and efficiently. Understanding the temporal constraints of missions and their dependencies is essential for achieving functional missions, maintaining situational awareness, and ensuring system success.

The main questions we intent to address in this paper are: **I)** How to thoroughly express and ensure properties regarding SoSs requirements and characteristics, while continuously evolving during runtime. **II)** How to define behaviors related to quantitative aspects such as time of SoSs and their management. **III)** How to ensure a strategic management of the execution of the different missions in SoSs' WFs. **IV)** How to ensure the autonomous executability of the desired behaviors and verify their correctness. Therefore, the main goal of our formal approach is to provide a formal design and specification of the SoSs' self-managed behaviors and express quantitative properties over these behaviors that can be formally verified.

1.2 Contribution

Formal methods present the appropriate mechanisms to address behavioral issues and the complexity inherent in SoS architectures. They are rigorous and rely on formal logic and operational semantics. In this paper, we propose a formal approach for modeling and analysis of time-aware SoSs and their WFs self-management (Figure 1). More precisely, we aim to facilitate the selection and execution of the best Functional Chain (FC), which represents the best path of sequenced missions leading to the final mission. To this end, we employ a set of rules and strategies that evaluate the entire WF: (1) the rules use conditional equations to prioritize missions based on time constraints during their execution. These rules classify missions into two categories, "Primary" or "Alternative," allowing the system to focus on selecting only the primary choices. (2) The strategies manage the execution of prioritized missions composing the optimal FC. They guide and govern the selection of rules to avoid the execution of unwanted behaviors (i.e., states characterized by violations, conflicts, or unwanted actions) representing the alternative missions and enforce rules leading to the desired behavior, described by primary missions.

The approach is based on the formal specification language Maude Strategy and its tools (Ölveczky, 2004), including a model-checker for formal verification (Ölveczky, 2004). The choice of this language can be argued by several reasons:

- The strong expressivity of the language to model SoSs in terms of structure (i.e., the WF structure and its FCs) and its components such as CSs, missions, and their temporal attributes (e.g., time, delays, deadlines, etc.).
- The executability of its rewriting logic semantics allows the implementation of conditional rules, such as prioritizing primary missions over the alternative ones in terms of time.
- The language and semantics support strategies, which allow for mechanisms to guide and specify which rule should be executed and under what conditions.
- Maude provides a bounded model-checking of logical properties represented in terms of combined patterns/invariants expressing specific SoS application requirements, which is relevant to study the managed SoS environment temporal evolution.

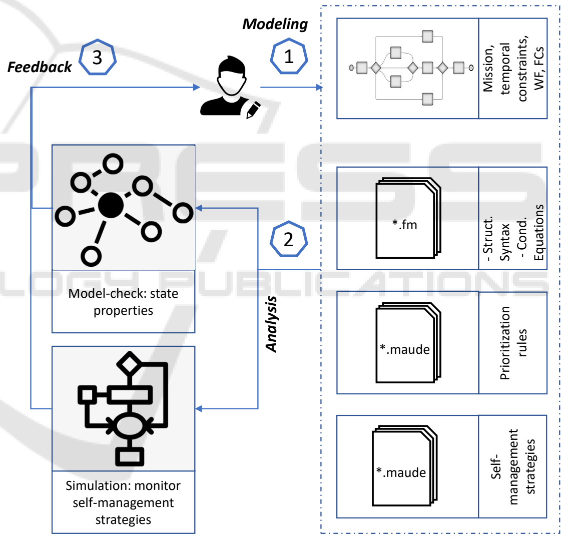
Modeling
1
Feedback
3
Mission, temporal constraints, WF, FCs
Model-check: state properties
2
*.fm
- Struct. Syntax - Cond. Equations
Analysis
*.maude
Prioritization rules
Simulation: monitor self-management strategies
*.maude
Self-management strategies

Figure 1: Proposed approach for self-managed SoSs.

Formally, the Maude-based specifications (Ölveczky, 2004) (Rubio et al., 2021) (Martí-Oliet and Meseguer, 1996) provide a powerful semantic framework for modeling and analyzing the behavior of workflows, making them ideal for validating and simulating the quantitative aspects of WF design. The approach defines operational semantics, conditional equations, rules, and management strategies to capture the dynamic behavior of the SoS and to provide an executable specification. Therefore The approach supports two main aspects of analysis: simulation/monitoring and formal verification. Using the Maude rewriting engine and built-in model-checker, the system's state evolution can be monitored to ensure the correct execution of self-management strategies.

1.3 Paper Structure

The remainder of the paper is structured as follows: Section 2 introduces our case study, FESoS, and outlines various components of an SoS. Section 3 provides an overview of rewriting theory and the Maude strategy language. Section 4 explores the static and dynamic aspects of the workflow. Section 5 focuses on implementing self-management strategies to promote desired behaviors and mitigate unwanted ones. Section 6 discusses the practical application and validation of our approach. Section 7 reviews related works, and Section 8 provides a summary and conclusion of the paper.

2 MOTIVATING EXAMPLE

The case study explores the FESoS (Petitdemange et al., 2018) which aims to protect people and property. The FESoS consists of several interconnected CSs, and Missions. The MonitoringSoS, SAMU, Hospital CS, Civil Security, SDIS35, Search and Rescue Teams (SRT), and Fire and Rescue Services (FRS) are all part of the FESoS. In a possible situation, a major fire breaks out in a populated area, posing a significant threat to the safety of people and property. The FESoS is triggered to respond to this emergency. The MonitoringSoS deploys UnmannedAerialVehicles (UAVs) for Aerial Surveillance. The Wireless Sensor Net CS (WSNCS) conducts Environmental Monitoring of different environmental factors e.g. to assess air quality, temperature, etc. The CODIS35 controller oversees operations, analyzes data and coordinates resources. The SAMU is tasked with two missions: Patient Evacuation and Patient Transportation. TheHospitalCS activates its Emergency Reception and Triage to receive and assess the incoming patients. The Medical Treatment provides necessary medical interventions. Civil Security takes charge of Emergency Response Coordination managing communication and coordination between all involved entities. SDIS35 and SDIS56, the fire-fighting and emergency response organizations, implement their respective missions to contain and extinguish the fire. SRT conduct Rapid Assessment and Search Operations to locate and rescue individuals trapped or stranded due to the fire. FRS engages in Fire Suppression and Control, ensuring the fire does not spread further. They also handle Hazardous Material Handling and Containment and Structural Assessment and Collapse Rescue if needed.

The effective coordination of response efforts within the FESoS serves as a motivating example for the necessity of self-control mechanisms and strategic prioritization rules based on time. The dynamic nature of emergency scenarios, such as civilian evacuation, fire suppression, and medical response, requires the use of real-time data to sequence and prioritize missions effectively and ensure a coordinated response (see Figure 2). In this context, the adoption of strategies and rules focused on temporal prioritization becomes critical. These strategies ensure that time-sensitive missions are executed in the appropriate sequence while avoiding unwanted behaviors such as delays, premature starts, or extended durations. By leveraging temporal interdependencies, the system can dynamically adjust workflows to address evolving conditions and meet strict time constraints. Integrating temporal prioritization strategies into the logical architecture model not only enhances the understanding of FESoS behavior but also facilitates the design of an effective emergency response system. Such a system must dynamically adapt to time-critical requirements, ensuring the smooth execution of missions while avoiding unwanted behaviors, and optimizing the overall impact of the FESoS.

3 OVERVIEW ON MAUDE LANGUAGE

The utilization of Maude as a declarative language becomes relevant. Maude's expressive nature in equational and rewriting logic aligns with our need for a powerful language to support programming, formal specification execution, and formal analysis and verification (Ölveczky, 2004). By leveraging Maude's concurrent rewriting capabilities and equational structural axioms, we can logically deduce and reason about the behavior of the system under consideration.

In the context of our paper on temporal mission priorities and states, the modules in Maude, along with their rewriting theory, denoted as a tuple $R = (\Sigma, E \cup A, R)$, play a crucial role. These modules provide a formal and structured representation of the system's behavior and transitions, incorporating temporal aspects.

The equational theory part of the modules, represented by $(\Sigma, E \cup A)$, encompasses the signature $\Sigma = (S, C, <, F, M)$, which includes:

- **Sorts and Subsorts (S):** Representing the types and hierarchies of temporal entities in the system.
- **Class Names (C):** Defining the classifications of

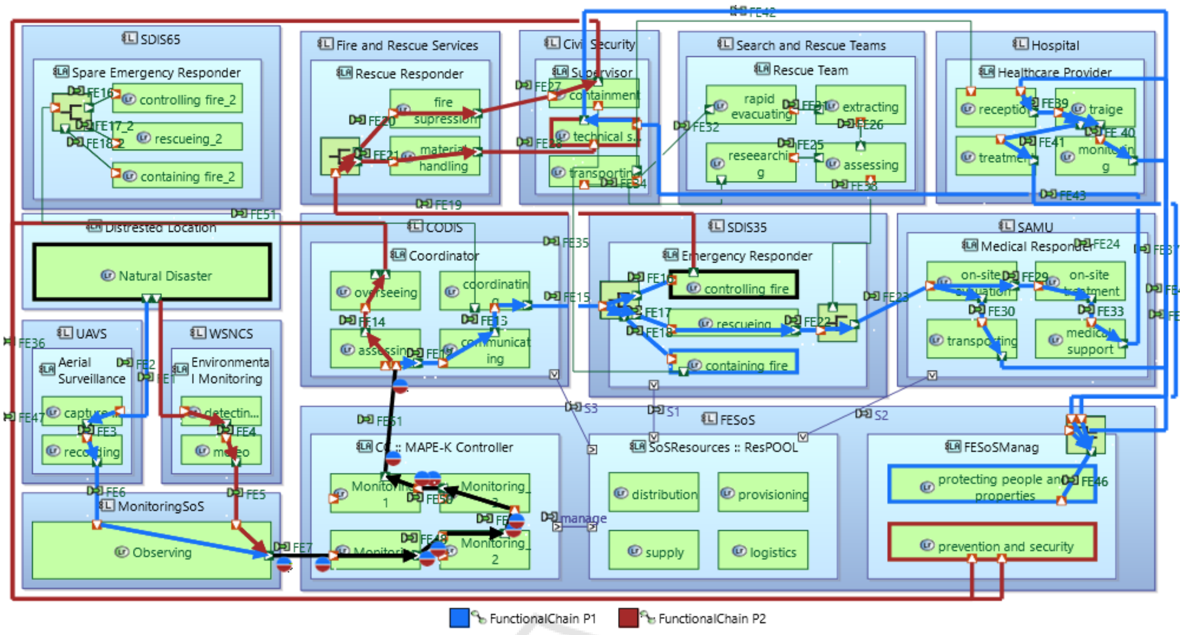

Figure 2: Overview of the Logical Architecture model of FESoS.

missions within the workflow.

- **Subclass Relation (<):** Structuring hierarchies among class names.
- **Functions and Messages (F, M):** Enabling interactions and operations on missions.

The set E denotes equations and membership tests, some of which can be conditional, while A represents equational axioms associated with specific operators in the signature Σ. The tuple R consists of both conditional and nonconditional rewrite rules, capturing the dynamic behavior of missions over time.

The specification of the proposed formal approach is based on the operational semantics of different elements of SoSs' workflows using Maude. In its basic form, the latter is equipped with two types of modules: functional modules and system modules. By using the extension of the Strategy Language for Maude, we add the strategies' module to formalize the semantics of the self-management approach. The three types of modules are summarized below:

- **Functional Module:** Specifies the static part of a specified SoS as a theory in membership equational logic as a pair $(\Sigma, E \cup A)$, where:
  - Σ specifies the type structure (sorts, subsorts, operators, etc.).
  - E is the collection of possibly conditional equations.
  - A is a collection of equational attributes for the operators (e.g., associative, commutative).
- **System Module:** Specifies the dynamics of the implemented SoS as a rewrite theory represented as a triple $(\Sigma, E \cup A, R)$, where:
  - $(\Sigma, E \cup A)$ is the module's equational theory part.
  - R is a collection of possibly conditional rewrite rules that describe the system's dynamic behavior.
- **Strategy Module:** Defines a set of strategies that control and guide the application of rewriting rules in Maude as $(\Sigma, E \cup A, S(R, SM))$, where:
  - $(\Sigma, E \cup A)$ represents the equational theory of the system.
  - SM is a semantics describing the behavior of a system, constructed from:
    - R: A set of rewriting rules.
    - S: A strategy module guiding the rewriting and application of these rules.

The different modules can be implemented using an object-oriented specification, encompassing objects, messages, classes, and inheritance. Concurrent systems are modeled as a multiset of juxtaposed objects and messages, where interactions between objects are governed by rewrite rules.

An object is represented as:

$$< O : C \,|\, a_1 : v_1, ..., a_n : v_n >,$$

where O is the object name, an instance of class C, a_i are attribute identifiers, and v_i are their corresponding values, for $i = 1...n$.

Class declarations follow the syntax:

$$\text{class } < C \,|\, a_1 : s_1, ..., a_n : s_n >,$$

where C is the class name, and s_i are sorts for attribute a_i. Subclasses can also be declared, making use of inheritance. Messages are declared using the keyword `msg`.

4 FORMAL MODELING SOS' WF

In this section, we focus on the initial step of establishing formal definitions and semantics for our approach. This involves specifying and defining the attributes and properties of missions. WFs describe the sequence and organization of interconnected missions aimed at completing specific goals, while functional chains represent a more specific path of missions that ensure the goal is accomplished. The functional chains present the desired behaviors, which refer to the intended actions and mission executions that align with the system's objectives, facilitating effective workflows of missions and achievement of goals. Conversely, unwanted behaviors represent actions or outcomes that deviate from planned intentions and potentially lead to inefficiencies or conflicts within the SoS' workflows. In this section, we present the operational semantics of both static entities and dynamic behavior describing the workflow specified using the Maude rewriting language.

As illustrated in Figure 3, the proposed self-management strategies are encoded within Maude's functional and system modules. This integration results in the definition of five complementary modules that collectively encompass the operational semantics, summarized below:

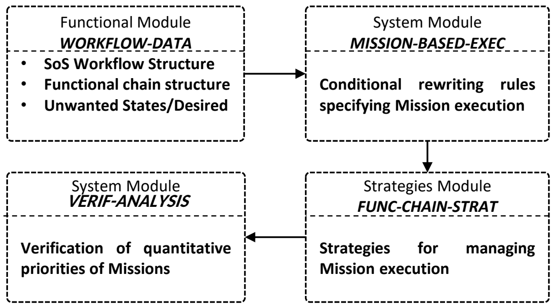
Functional Module
WORKFLOW-DATA
• SoS Workflow Structure
• Functional chain structure
• Unwanted States/Desired
System Module
MISSION-BASED-EXEC
Conditional rewriting rules specifying Mission execution
System Module
VERIF-ANALYSIS
Verification of quantitative priorities of Missions
Strategies Module
FUNC-CHAIN-STRAT
Strategies for managing Mission execution

Figure 3: Overview of the Maude-based specification modules.

- **Functional Module WORKFLOW-DATA:** Its role is to define the structural syntax of the workflow of Directed SoSs by implementing the functional chain elements (i.e., different executing missions) in the form of sorts and operations. This structure allows for the construction of a complete structured process that begins with individual missions and finishes with one or more global missions, resulting in a structured missions workflow enabled by the systematic organization of temporal dependencies.

- **The System Module MISSION-BASED-EXEC:** This module includes the WORKFLOW-DATA module to implement a set of rules and conditions to enable the execution of missions based on temporal attributes and conditions. It encapsulates the semantics provided by a suite of rewriting rules, facilitating the execution of a suitable functional chain by selecting the optimal choices based on timing criteria.

- **The FUNC-CHAIN-STRAT Module:** This module integrates the TIME-BASED-EXEC module to formulate strategies that direct and plan SoS execution. It focuses on enforcing specific rewriting rules to self-prioritize behaviors, ensuring missions align with objectives. It effectively avoids unwanted behaviors and selects optimal mission paths based on criteria such as arrival time and duration.

- **System VERIF-ANALYSIS Module:** This module describes a set of LTL properties introduced to verify the desired/unwanted behaviors related to time constraints of the SoS. These properties are analyzed using the Maude model-checker tool, and more precisely, search commands are used to verify a property on a simulation from an initial state of a SoS to a final state, applying the set of conditional rewriting rules.

This approach supports a modular integration, in which Maude-based modules can be integrated with others, allowing easy extension or independent editing of their specifications. More specifically, by leveraging conditional rewrite logic, the system maintains the operational flexibility and resilience essential to managing the complexities of these systems.

4.1 Mission Specification

The specification of each mission in the SoS' WF includes the necessary information to describe the mission's structure, checking predicates, temporal constraints, and their violation signals, all specified in an object-oriented class. The latter is characterized by its timeline and operational parameters. This class is defined by several attributes that describe the mission's state and progress, alongside its temporal constraints and resource requirements. This class is defined by the following Maude declaration:

```
class Mission | localClock : Time, duration
: Time, arrivalTime : Time, quitTime : Time,
delay : Time, deadline : Time, missionState:
```

```
MissionState, resType : ResType,  RAUT Time,
sg : Sig, rs : Rs .
```

The localClock attribute serves as a real-time clock to track the mission's progress and timeline. The duration describes the total operational timeframe, while arrivalTime and quitTime describe the starting and ending, framing the mission's execution state. Any deviations from this timeline are captured by the delay attribute. The deadline describes a strict time limit for mission completion. The missionState reflects the current state of the mission instance.

After its Idle state and if its TCs are respected, the mission moves to waitConsResp and waits for an answer from the controller after sending an allocation message. At this moment, a resource may be allocated to this mission instance, and it moves to the state executing, then to succeed after finishing the execution. In case of unavailability of resources (or absence), the Mission will receive a reject message from the controller and return to the Idle state. If the Mission encounters any unexpected problem, it can ask the controller to extend the availability time of the resource by sending the message isAskRenewing to ask for a renewal of the allocation of resources. And then, the controller can reply by sending a renewAskOk message, If the renewal request is accepted, or by sending isFailed message if the request is denied and the mission reaches the Idle state, and at the end, the mission instance will send an exit message to the controller and it will move to Idle state. The following Maude code specifies possible states that a mission can be in throughout its lifecycle.

```
sort MissionState .
ops idle failed waitConsResp  executing
succeeded rnwAsk : -> MissionState [ctor] .
ops isConsReqSent isConsReqAccepted
    isConsReqRejected : M R -> Bool .
```

4.2 WF Model: Static Aspects

The concept of WF is defined in an SoS as a comprehensive model that structures a set of FCs of missions to accomplish specific goals. The WF model includes FCs which are specific sequences or combinations of missions designed to execute a part of the overall WF. The WORKFLOW-DATA module in Maude specifies structured and orchestrated WFs and their associated FCs. This module focuses on:

- Linking various FCs and integrating them into the overarching WF structure.
- Classifying these chains into two distinct categories: primary and alternative. This classification enables the definition of the operational behavior and managing missions' priorities within the WF.

For this classification, the module defines a set of sorts such as Mission, SecOrPri, FuncChain, and Workflow which form the structure of these elements. Each sort serves a distinct purpose: Mission encapsulates individual missions or operations, while SecOrPri represents a secondary or primary categorization of these missions. The FuncChain sort particularly acts as a generic structure under which both missions and their classifications (secondary or primary) are nested.

```
op DelayViolated DeadlineViolated
    isWCETViolated isArrivalTimeViolated:
    Time Time -> Bool .
op isTCViolated : Mission -> Bool .
ops AlternativeM PrimaryM :
    -> SecOrPri [ctor] .
op __ : FuncChain FuncChain
    -> FuncChain [ctor assoc comm] .
op _|_ : FuncChain FuncChain
    -> Workflow [ctor comm prec] .
```

The classification of Missions into primary (PrimaryM) and alternative (AlternativeM) allows for the prioritization of certain FCs over others in WF, this feature is essential in scenarios where time-sensitive missions are involved.

- PrimaryM: represents a subset of missions that are given priority during the current execution to achieve the final global Mission.
- AlternativeM: represents a subset of missions that are not given priority for the current execution, these missions could still exist in the system and could be relevant in future executions.

Subsequently, the module employs conditional equations to analyze the complexities of time. These equations enable real-time responses to changes in WF. The module defines different conditions which work on monitoring the system's compliance with its operational parameters. These conditions act as triggers for rewriting rules the two modules TIME-BASED-EXEC, allowing the SoS to adapt its behavior in response to environmental changes.

4.3 WF Model: Dynamic Aspects

The `MISSION-BASED-EXEC` module in Maude leverages the conditional equations of `WORKFLOW-DATA` to dynamically prioritize and manage missions based on temporal constraints. It supports:

- Prioritization of missions by urgency and impact.

- Management of sequential, simultaneous, and concurrent executions.
- Handling unexpected operational changes while maintaining time constraints.
- Ensuring missions are completed within designated time frames.

The module categorizes missions into `PrimaryM` and `AlternativeM` and dynamically adjusts their priorities to construct functional chains (FCs). Below are the key prioritization rules, with enhanced explanations:

Sequential Mission Ordering. This rule ensures that missions $M1$ and $M2$ are executed in a strict sequential order when one depends on the completion of the other. The rule rearranges missions to maintain the correct order, ensuring logical dependencies are respected.

```
rl [sequentialMissionOrdering] : M1 M2
AlternativeM | PrimaryM => AlternativeM |
PrimaryM M1 M2 .
```

Prioritize Earliest Start. This rule prioritizes missions based on their start times, ensuring that missions scheduled to begin earlier are given precedence. By comparing the arrival times of $M2$ and $M3$, the mission with the earlier start time is promoted to `PrimaryM`.

```
crl [prioritizeEarliestStart] : M2 M3
AlternativeM | PrimaryM M1 =>
if (getArrivalTime(M2) < getArrivalTime(M3))
then AlternativeM M3 | PrimaryM M2 M1
else AlternativeM M2 | PrimaryM M3 M1 fi .
```

Prioritize Earliest Completion. This rule focuses on completing missions as quickly as possible by prioritizing those with earlier finish times. This method minimizes delays in the overall workflow and ensures that shorter missions do not unnecessarily hold up progress. The rule compares the completion times (`getQuitTime`) of $M2$ and $M3$, prioritizing the mission that can finish sooner.

```
crl [prioritizeEarliestCompletion] : M2 M3
AlternativeM | PrimaryM M1 =>
if (getQuitTime(M2) <= getQuitTime(M3))
then AlternativeM M3 | PrimaryM M2 M1
else AlternativeM M2 | PrimaryM M3 M1 fi .
```

Prioritize Shortest Mission. This rule gives priority to missions with the shortest duration, ensuring that smaller missions are completed first. This can accelerate the overall workflow by quickly resolving missions that would otherwise accumulate and create inefficiencies. The rule evaluates the durations (`getDuration`) of $M2$ and $M3$ and promotes the shorter mission to `PrimaryM`.

```
crl [prioritizeShortestMission] : M2 M3
AlternativeM | PrimaryM M1 =>
if (getDuration(M2) <= getDuration(M3))
then AlternativeM M3 | PrimaryM M2 M1
else AlternativeM M2 | PrimaryM M3 M1 fi .
```

Minimize Mission Delays. This rule targets the reduction of delays in mission execution, which is crucial in maintaining efficiency and meeting deadlines. By comparing the delay metrics (`getDelay`) of $M2$ and $M3$, the rule prioritizes the mission with the least delay, ensuring smoother progression through the functional chain.

```
crl [minimizeMissionDelays] : M2 M3
AlternativeM | PrimaryM M1 =>
if (getDelay(M2) <= getDelay(M3))
then AlternativeM M3 | PrimaryM M2 M1
else AlternativeM M2 | PrimaryM M3 M1 fi .
```

On the other hand, the rules defined in the previous modules aim to ensure desired behavior and outcomes during mission execution. However, practical execution may encounter challenges such as the complexity of inter-CS interactions, unpredictable environmental conditions, or human error. These issues can significantly impact the effectiveness of rule implementation, especially in dynamic and critical emergency scenarios.

The rest of this section addresses rules that describe unwanted behaviors and their operational significance within workflows. These rules enable the system to overcome runtime challenges, avoid environmental issues, and ensure that missions align with the strategic goals of the SoS. The rules focus on addressing timing violations:

Deadline Violation. This rule checks if a mission exceeds its deadline. If the current time (`Clock`) is greater than the mission's deadline (`Deadline`) and the mission state is not already `failed`, the mission's state transitions to `failed`. This is essential to identify and address missions that cannot meet their time frame.

```
crl [deadlineViolation] :
< Mid : Mission | deadline : Deadline,
localClock : Clock, missionState : State>
=> < Mid : Mission | missionState : failed >
if isDeadlineViolated(Clock, Deadline) and
State =/= failed .
```

Delay Violation. This rule activates when a mission's delay (`Delay`) exceeds its expected duration (`ExpectedDuration`). If the delay surpasses this threshold and the mission state is not already `failed`, the state is updated to `failed`. This ensures timely identification of delayed missions and promotes timely execution.

```
crl [delayViolation] :
< Mid : Mission | delay : Delay,
expectedDuration : ExpectedDuration,
missionState : State > =>
< Mid : Mission | missionState : failed >
if isDelayViolated(Delay, ExpectedDuration)
and State =/= failed .
```

Arrival Time Violation. This rule ensures that missions do not start too early. If the current time (`Clock`) is less than the mission's expected arrival time (`ExpectedArrivalTime`) and the mission state is not `failed`, the state transitions to `failed`. This prevents premature mission execution and ensures proper scheduling.

```
crl [arrivalTimeViolation] : < Mid : Mission |
arrivalTime : ExpectedArrivalTime, localClock
: Clock, missionState : State, ... > => < Mid
: Mission | missionState : failed, ... > if
isArrivalTimeViolated(Clock,
ExpectedArrivalTime) and State =/= failed .
```

Quit Time Violation. This rule checks if a mission exceeds its quit time (`QuitTime`). If the current time (`Clock`) surpasses the quit time and the mission state is not `failed`, the state transitions to `failed`. This ensures missions adhere to their time limits and prevents cascading delays.

```
crl [quitTimeViolation] :
< Mid : Mission | quitTime : QuitTime,
localClock: Clock, missionState : State,
... > => < Mid : Mission | missionState :
failed, ... > if isQuitTimeViolated(Clock,
QuitTime) and State =/= failed .
```

These rules collectively address critical timing violations, ensuring missions do not start too early, delay excessively, miss deadlines, or exceed allocated time. They play a crucial role in managing workflows in time-sensitive environments by strategically identifying and mitigating potential issues.

5 STRATEGIES FOR SELF-MANAGEMENT IN SoSs

As seen in the previous section, the execution of desired behavior rules related to time does not guarantee that their execution is respected, since escaping from a violated, non-deterministic or unwanted behavior states without applying the rules describing the unwanted behavior is possible. This suggests that the unwanted behavior rules must be applied before any rules of desired behaviors, for which strategies will be helpful in this case. In this section, we propose a set of self-management strategies proposed govern both the desired behaviors specified using rewriting rules and avoid any unpredictable, unwanted ones. These strategies are key to the SoS's ability to balance functional execution constraints, they help at:

- Determining the execution of missions provided by CSs within the SoS. The aim is to ensure timely mission completion.
- Selecting and executing the optimal path to achieve the SoS's final goal, considering factors such as urgency.
- Prioritizing missions with urgent or time-sensitive requirements. It schedules missions according to their time-criticality, addressing scenarios where timing is a crucial factor.

The Strategy module forms the foundation for two derived classes: MainStrat and ComponentStrat. The ComponentStrat provides additional support to govern and respond to dynamic prioritization of missions, while the MainStrat represents the primary strategies that are essential for selecting a functional chain to execute. The main strategies within the MainStrat class i.e. mission-based functional chain strategies, determine the execution path of the workflow, guiding it toward completing its goals. These strategies are robust and incorporate the component rules to promote advantageous missions and ensure that the workflow remains resilient in the face of changing internal and external conditions.

5.1 Component Strategies

In this section, we define a set of component strategies based on time constraints in the module FUNC-CHAIN-STRAT. The latter includes MISSION-BASED-EXEC to provide four component strategies as follows:

orderBasedExecStr: this strategy switches between concurrent and sequential mission execution. It starts with executing concurrent execution, governing parallel mission, and then goes to sequential ordering if concurrent execution is not feasible.

```
sd orderBasedExecStr :=
    SequentialMissionOrdering
    or-else ConcurrentMissionExecution .
```

timeCriticalmissionstr: this strategy focuses on time-sensitive missions. It firstly prioritizes missions based on their start times and then completion times, ensuring that the most urgent missions are addressed promptly.

```
sd timeCriticalMissionStr :=
    PrioritizeEarliestStart
    or-else PrioritizeEarliestCompletion .
```

quickCompletionStr: this strategy emphasizes quick mission completion. It aims to prioritize missions with shorter durations and minimize delays in mission execution.

```
sd quickCompletionStr :=
    PrioritizeShortestMission
    or-else MinimizeMissionDelays .
```

violatedConstraintsStr: this strategy handles missions that risk violating time constraints. It dynamically applies rules to manage and rectify such situations, preventing potential disruptions in mission execution.

```
sd violatedConstraintsStr :=
    DeadlineViolation |
    DelayViolation |
    ArrivalTimeViolation |
    QuitTimeViolation .
```

In these strategies, the operators ***or-else***, |, and ***;*** play pivotal roles in strategizing mission execution within a SoS:

- The ***or-else*** operator is crucial for providing fallback options, ensuring that if one strategy is not applicable, an alternative can be immediately employed. This is exemplified in the timeCriticalMisStr strategy, where the system first attempts to apply prioritizeEarliestStart and, failing that, defaults to prioritizeEarliestCompletion.

- The | operator introduces a layer of non-determinism, allowing the system to choose any applicable strategy without a fixed order, as seen in the violatedConstraintsStr strategy, where the system can select any one of the violation handling rules like deadlineViolation or delayViolation, etc.

- The ***;*** operator, on the other hand, ensures a controlled, sequential execution of strategies.

In the next section, we show how these strategies will be coordinated and executed in a specific sequence using one Main Strategy.

the FUNC-CHAIN-STRAT module aligns with the dynamic and often unpredictable nature of SoS environments, ensuring efficient and effective system operations.

5.2 Main Strategies

The `FUNC-CHAIN-STRAT` module also defines one main Main Strategy `missionBasedFCStrategy` which incorporate the previous strategies and their rules. In this strategy, the component strategies like `orderBasedExecStr`, `timeCriticalMissionStr`, and `quickCompletionStr` are executed in a specific sequence using the flexibility of or-else, the non-determinism of |, and the controlled execution of ;. This sequential execution guarantees a structured and orderly approach to execute missions. Collectively, these strategies enable the system to adaptively manage various temporal scenarios in an SoS, prioritizing mission execution utilization while avoiding conflicts and undesirable states. The latter is reached by leveraging the flexibility of not(violatedConstraintsStr) which aligns with the dynamic and the unpredictable nature of time related violated constraints in an SoS environments.

```
strat missionBasedFCStrat .
sd missionBasedFCStrat := (match AlternativeM
| G) ? idle : (orderBasedExecStr;
timeCriticalMissionStr; quickCompletionStr;
not(violatedConstraintsStr); selectedFuncCh) .
```

The strategy missionBasedFCStrategy is recursive strategies that repeat these steps forever or until the final mission is reached. They are restrictive and avoid executing conflicts or any unwanted states by discarding all missions where violatedConstraints and/or missionBasedFCStrategy are possible with $not(\alpha) \equiv \alpha ? fail : idle$. i.e. Executing violatedConstraints and/or missionBasedFCStrategy still requires visiting the conflicts and unwanted states defined in the module WORKFLOW-DATA and specified in the two execution modules as conditional rules, but this execution path is discarded as if the state were never visited.

More specifically, these strategies encapsulate the operational semantics defined in WORKFLOW-DATA, including various ops and ceq conditions that determine the states of missions. For instance, conditions like DelayViolated, DeadlineViolated, isWCETViolated, and isArrivalTimeViolated identify risky states based on time constraints. These, along with the unscheduledMissionExecution rules in TIME-BASED-EXEC, help identify and manage undesirable states.

By synchronizing these strategies and rules, the module effectively prevents management conflicts and contradictory decisions. It ensures that each mission is executed while maintaining the overall balance within the SoS.

6 EXECUTION AND ANALYSIS

In this section, we analyze the FESoS case study through multiple execution scenarios. As depicted in Figure 2, two functional chains are presented, demonstrating their potential to achieve global mission objectives. For simplicity and ease of reference, the missions within these functional chains are designated as M1, M2, M3, ..., Mn. The Maude Strategy Language is employed to execute these mission specifications. Using the Maude's command `srew` which facilitates the exploration of different state transitions within the strategy, the missionBasedFCStrat strategy guides the system from its initial state to the critical global mission. It integrates various component strategies from the MISSION-BASED-EXE module (e.g. SequentialMissionOrdering and ConcurrentMissionExecution), as well as prioritization strategies (e.g. PrioritizeEarliestStart and PrioritizeEarliestCompletion) to categorize FESoS missions into two distinct groups: PrimaryM and AlternativeM:

```
Maude> srew initState using
    missionBasedFCStrat.
Solution 1
rewrites: 124
result missionBasedFCStrat: AlternativeM M4
    M5 M11 M13...M29 | PrimaryM M0 M1 M2 M3
    M6 M7 M8.....M30 M33 M34
No more solutions.
rewrites: 124
```

PrimaryM Missions: These missions constitute the main functional chain essential for achieving the global mission M34. Prioritized based on urgency, start and completion times, and their critical role in the global objective, missions such as M0 through M30, M33, and M34 are executed with precedence to ensure the success of the overall mission.

AlternativeM Missions: These missions, including M4, M5, M11, and M13 through M29, provide supportive roles. They supply necessary resources and assistance to facilitate the seamless execution of the primary functional chain.

On the other hand, the `search` command in Maude is a versatile tool for exploring (following a breadth-first strategy) the reachable state space in different ways of systems defined by rewrite rules. This command is particularly powerful for systems where multiple outcomes or states can result from a given initial condition, and it's essential for analyzing systems with complex behaviors or verifying properties across potentially vast numbers of states:

```
search [ n, m ] in <ModId> : <Term-1>
<SearchArrow> <Term-2> such that <Condition>.
```

In the context of the FESoS case study, the search command could be used to simulate different scenarios of emergency responses, ensuring that the strategic objectives are met and that the system behaves as expected in various potential emergencies. Given the diverse potentials of FESoS to accomplish global mission objectives, the search command enables the exploration of all possible functional chain rules and helps identify terms that align with a specified target:

```
Maude> search initState =>* AlternativeM |
PrimaryM M0 M1 M2 M3 M4 M5 M6 M7... M33 M34 .
Solution 1 (state 74)
states: 75 rewrites: 123
empty substitution
no more solutions.
states 80
```

In this case, the search command was employed to verify whether the initial state of the FESoS system could transition into an expected configuration, specifically a PrimaryM mission chain that represents the final goal. The search confirmed a valid path to the desired state, involving 123 rewrite steps across 75 possible states, with no variable substitutions required. However, the path revealed conflicts with the module's rules, as it navigated through states exhibiting unwanted behaviors. This was further validated using the show path command, which traced the transitions, showing early adherence to rules like sequentialMissionOrdering but later involving violations such as arrivalTimeViolation. For instance:

```
Maude> show path 40 .
states 0, WF: PrimaryM | AlternativeM M0 M1 M2
M3 M4 M5 M6 M7 M8..... M33 M34
===[ rl ...[sequentialMissionOrdering] . ] ===>
state 1, WF:  PrimaryM M0 | AlternativeM M1 M2
M3 M4 M5 M6 M7 M8..... M33 M34
===[ rl ...[sequentialMissionOrdering] . ] ===>
state 2, WF:  PrimaryM M0 M1 | AlternativeM M2
M3 M4 M5 M6 M7 M8..... M33 M34
===[ rl ...[sequentialMissionOrdering] . ] ===>
state 3, WF:  PrimaryM M0 M1 M2 | AlternativeM
M3 M4 M5 M6 M7 M8..... M33 M34
===[ rl ...[sequentialMissionOrdering] . ] ===>
state 4, WF: PrimaryM M0 M1 M2 M3 | AlternativeM
M4 M5 M6 M7 M8..... M33 M34
===[ rl ...[prioritizeEarliestStart] . ] ===>
state 5, WF: PrimaryM M0 M1 M2 M3 M6 |
AlternativeM M4 M5 M6 M7 M8..... M33 M34 ....
===[ rl ...[arrivalTimeViolation] . ] ===>
state 40, WF: PrimaryM M0 M1 M2 M3 M6 M7 M8..M30
| AlternativeM M4 M5 M6 M7 M8..... M33 M34
```

The provided result shows a workflow sequence of states and transitions (denoted as WF). This workflow is structured around managing a set of missions (M0, M1, M2, etc.) using two categories: PrimaryM and AlternativeM:

- Initial State (state 0): All missions (M0 through M34) are under AlternativeM. This represents a starting point where all missions are initially in an alternative or secondary queue, awaiting prioritization or scheduling.
- Transitions Using [sequentialMissionOrdering]: The sequence of transitions from state 1 to state

4 appears to follow a rule labeled [sequentialMissionOrdering]. This suggests a process where missions are moved one by one from AlternativeM to PrimaryM. For example, in state 1, M0 is moved to PrimaryM, and in state 2, M1 is also moved, and so on.

- Transition Using [prioritizeEarliestStart]: At state 4, the transition rule changes to [prioritizeEarliestStart], indicating a shift in the strategy for ordering missions. This means that instead of simply moving missions sequentially, the next mission to be moved to PrimaryM is selected based on the strategy of earliest start time. This is evident as M6 is moved to PrimaryM in state 5, skipping M4 and M5.
- Final State (state 40): By state 40, a set of missions (M0, M1, M2, M3, M6, M7, M8, ..., M30) are in PrimaryM, and the remaining ones (M4, M5, M31, M32, M33, M34) are in AlternativeM. The transition to this state is marked by [arrivalTimeViolation], suggesting that this transition might be due to a violation of expected arrival times of missions, impacting the scheduling or prioritization.

7 RELATED WORK

While the field of SoS research employs formal languages and approaches such as ArchSoS, mKAOS, BRS etc. they still neglect time-aware missioned SoSs and their self-managed workflows. This section provides an overview of these works.

In (Seghiri et al., 2022), the authors proposed ArchSoS, a formal ADL combining BRS and Maude language to address hierarchical structures and dynamic reconfigurations in SoSs. ArchSoS facilitates understanding and design analysis through graphical and formal syntax. The approach integrates rewrite theories to provide operational semantics, enabling modeling, simulation, and qualitative analysis of SoS behaviors using Maude's rewriting engine and LTL model-checking. While effective in modeling and verifying cooperation and mission consistency. In the same context, the paper (Stary and Wachholder, 2016) have explored the interoperability and emergent behavior in SoSs using bigraphs. This approach addresses the complexity of highly interactive distributed systems, such as e-learning environments, where dynamic adaptation to user needs and operational conditions is critical. Bigraphs provide a dual representation of systems, capturing both physical (space) and logical (link) relationships, making them particularly suited for managing the interactions of federated, independent CSs within an SoS.

In (Silva et al., 2020), the authors introduced a methodology to formally verify mission-related properties in SoS architectural models, emphasizing emergent behaviors and mission accomplishment. Using mKAOS and DynBLTL, they defined and verified properties at three levels, derived automatically from mKAOS models, with statistical model checking via PlasmaLab. On the other hand, the paper (Gassara and Jmaiel, 2017) has presented BiGMTE, a tool for modeling and simulating SoS architectures using BRS transformation techniques. The tool integrates the Big Red graphical editor for creating bigraphs and the GMTE tool for graph matching and transformation. The methodology involves five steps: creating BRS, encoding them into graphs, graph matching, transformation, and visualizing the results. In the same context, the paper (Gassara et al., 2017) presents B3MS, a multi-scale modeling methodology for SoS based on BRS. It ensures "correct by design" architectures through a refinement process that starts with a designer-defined coarse-grained scale and automatically refines it with lower-scale details while respecting system constraints.

The authors (Oquendo, 2016a) have introduced an approach to enable the description of evolutionary architectures that sustain emergent behaviours and support dynamic reconfigurations for ongoing SoS missions. The work defines SosADL from a behavioural viewpoint, facilitating the specification of independent CSs, mediators among them, coalitions of mediated CSs, and the architectural conditions that necessitate the emergence of specific SoS behaviours. Moreover, the same authors have proposed (Oquendo, 2016b) an approach to support automated verification and establish correctness properties of SoS architectures, thereby ensuring that the structured and analytical approach to SoS architecture is not only descriptive but also verifiable. In another research work (Morrison and Kirby, 2004), they have explored the functionalities of the ArchWare ADL, emphasizing its role in active architectures. The paper highlights the challenges of a cohesive system wherein model specification and enactment coalesce as integral facets of a singular, dynamic execution state.

While these approaches provide robust frameworks for representing dynamic behaviors and structural changes, they fail to address critical quantitative aspects, such as the management of temporal constraints which are essential for time-sensitive environments. Moreover, They lack strategies for managing and executing CSs in an SoS, preventing the effective handling of Workflows of missions. The absence of these strategies makes it difficult to verify the correctness of quantitative properties or simulate both desired and unwanted behaviors and this is what limits their applicability and effectiveness in real scenarios.

8 CONCLUSION

In this paper, we proposed a formal solution for designing, implementing, and verifying Time-Aware SoS workflows, focusing on managing mission behaviors constrained by temporal conditions. We introduced a set of rewriting rules to specify and prioritize primary and alternative missions based on quantitative properties and constraints. These rules are governed by strategies that enforce desired behaviors and avoid unwanted ones, ensuring the efficient execution of workflows. The approach was validated through a case study of the French Emergency SoS, using the Maude Strategy language for formal verification.

For future work, we aim to extend the formalization and verification of self-* properties (e.g., self-adaptation, self-awareness) to support a broader range of predictable and unpredictable workflows. Additionally, we plan to enhance resource prediction and mission performance evaluation, providing SoSs with improved tools for planning and optimizing architectures to achieve better performance, cost efficiency, and resource optimization.

REFERENCES

Dridi, C. E., Hameurlain, N., and Belala, F. (2022). A maude-based rewriting approach to model and control system-of-systems' resources allocation. In *Advances in Model and Data Engineering in the Digitalization Era - MEDI-DETECT 2022 Workshop Proceedings*, volume 1751 of *Communications in Computer and Information Science*, pages 207–221. Springer.

Dridi, C. E., Hameurlain, N., and Belala, F. (2023). A maude-based formal approach to control and analyze time-resource aware missioned systems-of-systems. In *31th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE-2023)*. IEEE.

Gassara, A., Rodriguez, I. B. Jmaiel, M., and Drira, K. (2017). A bigraphical multi-scale modeling methodology for system of systems. *Computers & Electrical Engineering*, 58:113–125.

Gassara, A. Bouassida, I. and Jmaiel, M. (2017). A tool for modeling sos architectures using bigraphs. In *Proceedings of the Symposium on Applied Computing*, pages 1787–1792.

Halima, R. B., Klai, K., Sellami, M., and Maamar, Z. (2021). Formal modeling and verification of property-based resource consumption cycles. In *2021 IEEE International Conference on Services Computing (SCC)*, pages 370–375. IEEE.

Maamar, Z., Faci, N., Sakr, S., Boukhebouze, M., and Barnawi, A. (2016). Network-based social coordination of business processes. *Information Systems*, 58:56–74.

Maier, M. W. (1998). Architecting principles for systems-of-systems. *Systems Engineering: The Journal of the International Council on Systems Engineering*, 1(4):267–284.

Martí-Oliet, N. and Meseguer, J. (1996). Rewriting logic as a logical and semantic framework. *Electronic Notes in Theoretical Computer Science*, 4:190–225.

Morrison, R. and Kirby, G. Balasubramaniam, D. e. a. (2004). Support for evolving software architectures in the archware adl. *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 69–78.

Oquendo, F. (2016a). Formally describing the architectural behavior of software-intensive systems-of-systems with sosadl. In *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 13–22. IEEE.

Oquendo, F. (2016b). Formally describing the software architecture of systems-of-systems with sosadl. In *2016 11th system of systems engineering conference (SoSE)*, pages 1–6. IEEE.

Petitdemange, F., Borne, I., and Buisson, J. (2018). Modeling system of systems configurations. In *2018 13th Annual Conference on System of Systems Engineering (SoSE)*, pages 392–399. IEEE.

Rubio, R., Martí-Oliet, N., Pita, I., and Verdejo, A. (2021). Strategies, model checking and branching-time properties in maude. *Journal of Logical and Algebraic Methods in Programming*, 123:100700.

Seghiri, A., Hameurlain, N., and Belala, F. (2022). A formal language for modelling and verifying systems-of-systems software architectures. *International Journal of Systems and Service-Oriented Engineering (IJSSOE)*, 12(1):1–17.

Silva, E., Batista, T., and Oquendo, F. (2020). On the verification of mission-related properties in software-intensive systems-of-systems architectural design. *Science of Computer Programming*, 192:102425.

Stary, C. and Wachholder, D. (2016). System-of-systems support—a bigraph approach to interoperability and emergent behavior. *Data & Knowledge Engineering*, 105:155–172.

Ölveczky, P. C. (2004). Real-time maude 2.3 manual. Technical report, Research Report. http://urn.nb.no/URN:NBN:no-35645.