Design, Formalization, and Verification of Decision Making for Intelligent Systems*

Mohammad Hejase[†], Andreas Katis[‡], and Anastasia Mavridou[§]
Intelligent Systems Division, NASA Ames Research Center, Moffet Field, CA, 94035.

The development of autonomous systems requires a rigorous process that can guarantee a system's reliability in critical applications. At its core, an autonomous system bases its behavior on a well-defined decision making system. In this paper, we present a methodological basis for the design, formalization and formal verification of Decision Making systems for autonomous agents. The approach is generally applicable to operational objectives that can be functionally decomposed and subsequently represented as Hierarchical Finite State Machines. As a case study, we present the application of this method to implement a Decision Making model in Simulink. Furthermore, we present how we use NASA's FRET tool to write requirements in structured natural language and generate formal specifications that can be automatically digested by NASA's CoCoSim tool. Finally, we present how, by leveraging CoCoSim, we perform formal verification against the Simulink model and present analysis results.

I. Introduction

In recent years, the demand for and investment in autonomous systems has been growing rapidly across various domains, including space exploration, aerospace operations, and autonomous vehicles in urban settings. The imminent deployment of such systems, coupled with the critical nature of their decision making capabilities, necessitates the development of trustworthy and certifiable systems. Trusted autonomy refers to the ability of autonomous systems to make decisions reliably and in a manner that can be understood, validated, and trusted by human operators or stakeholders. This concept is of paramount importance, considering that these vehicles are utilized in environments where the well-being of human lives, valuable assets, and critical missions is at stake. To address this challenge, this paper proposes a design methodology for decision making systems that promotes trusted autonomy through a predictable, formalizable, and verifiable implementation. By pursuing rigorous yet simple design methodologies that prioritize trust and verification, the path to successful deployment of autonomous systems in diverse and complex environments can be paved.

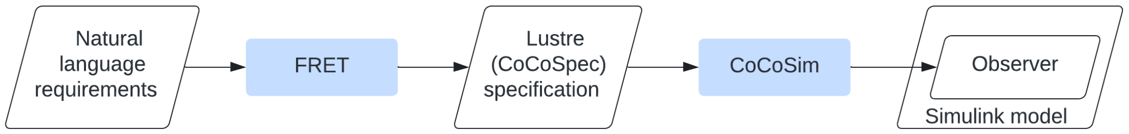
In this paper a design methodology is proposed for Decision Making systems in Section II. The proposed methodology begins with procedures for the systematic functional decomposition of the system overall goal until it is represented by a set of smaller and interconnected simpler sub-goals with clearly defined primitives (goals, tasks, procedures, measurements, resources, and constraints). A Hierarchical Finite State Machine (H-FSM) implementation for Decision Making (DM) is constructed as a result of the functional decomposition. Requirement types are then derived that give complete coverage to the intended DM behavior and implementation. Section III of the paper formalizes the DM requirement types using the FRET tool. Section IV presents the CoCoSim tool for the verification of a DM implementation. A workflow overview of the FRET-CoCoSim toolchain is shown in Figure 1. Section V presents a case study to illustrate the proposed design procedure, requirement elicitation and formalization, and verification of the implemented DM system.

*GOVERNMENT RIGHTS NOTICE. This work was authored by employees of KBR Wyle Services, LLC under Contract No. 80ARC020D0010 with the National Aeronautics and Space Administration. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, or allow others to do so, for United States Government purposes. All other rights are reserved by the copyright owner.

[†]Intelligent Systems Division, NASA Ames Research Center.

[‡]Intelligent Systems Division, KBR Inc. / NASA Ames Research Center.

[§]Intelligent Systems Division, KBR Inc. / NASA Ames Research Center.



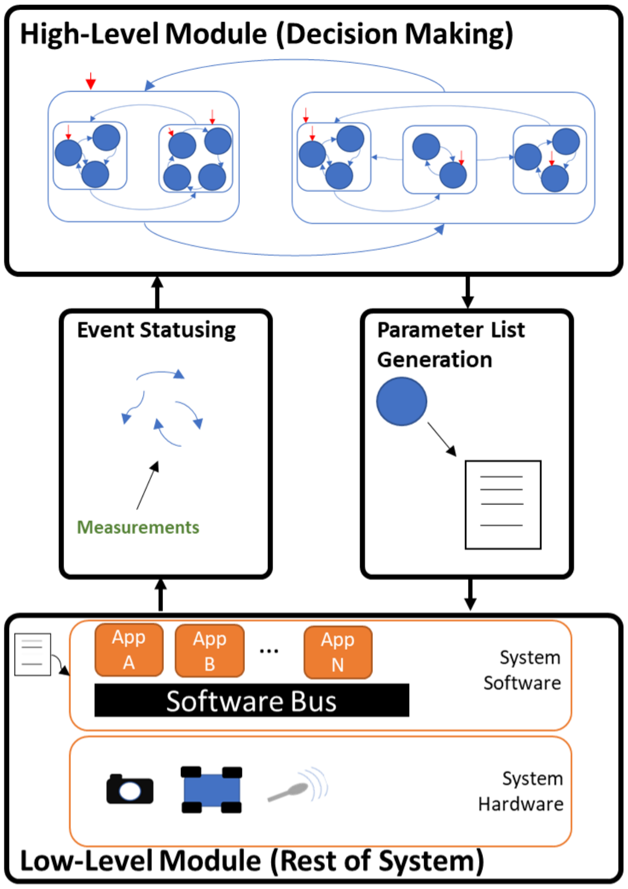
Natural language requirements
FRET
Lustre (CoCoSpec) specification
CoCoSim
Observer
Simulink model

Fig. 1 The requirement specification and verification toolchain

II. Decision Making Design Procedure

A. System Functional Decomposition and Functional Finite State Machine Construction

Hybrid-State Systems (HSS) serve as an ideal framework for the representation of Autonomous Systems that include a combination of continuous-time (system models/components, applications, control systems) and discrete-time behaviors (decision making) [1]. A HSS is composed of several layers as seen in Figure 2. The High-level layer dictates the mode of operation (e.g. Drive, Science Tasks, Wait, etc) or the phase of a mission an autonomous vehicle is in, whereas the low-level layers contain the low-level software components (health monitor, controllers, communication applications, etc.) along with actual system or modeled system dynamics. An interface is used to connect the high-level layers to the low level ones. The purpose of such an interface is twofold: 1) provide the high-level module with the necessary information required for decision making (events that trigger state transitions), and 2) provide low-level control modules with the phase-specific set-points and controller parameters (velocity set-points, controller mode, communication actions, etc.).



High-Level Module (Decision Making)
Event Statusing
Measurements
Parameter List Generation
App A
App B
...
App N
Software Bus
System Software
System Hardware
Low-Level Module (Rest of System)

Fig. 2 Decision Making Hybrid-State System Structure

The proposed Decision Making (DM) scheme is represented by the high-level layer of the system and is made up of a Hierarchical Finite State Machine (H-FSM). A H-FSM is made up of a collection of meta-states, which are states that are made up of states. The states that a meta-state is made up of are referred to as sub-states, these sub-states form internal state machines that define the sequence of actions to be executed in a meta-state. Note that sub-states may also have internal states (no hard limit on depth).

The choice of meta-states and the corresponding sub-states needs to be conducted in a systematic and comprehensive manner that is informed by the system capabilities and the mission requirements. A functional hierarchy is used for such a detailed decomposition of the system in order to arrange the decision making process into a hierarchical tree which can be then translated into a H-FSM [2]. For systematic decomposition of the system, six primitives are defined at each node of the decomposition [3]: Goals, Tasks, Procedures, Measurements, Constraints, and Resources. Goals represent the end results that need to be obtained at a node. Tasks represent the elementary job descriptions at the node. Procedures represent detailed methods of accomplishing tasks. Measurements represent data available to the decision making module. Constraints represent task restrictions or exemptions. Resources represent task restrictions or exemptions related to the use of procedures.

The top node in a functional hierarchy represents the main objective. The goal at that top node is first defined. Accessible measurements, available resources, and constraints imposed on the vehicle are then globally identified. A set of tasks are then identified that lead to the satisfaction of the goal at the top node. These tasks defined on the top node are then used to decompose the mission into a set of phases, which form the nodes on the next level of the functional hierarchy. For each of those phases, the same set of primitives are defined leading to the further decomposition of the mission into sub-phases. This decomposition is carried out until the lowest level of the decomposition is reached, with very elementary tasks defined at that level. Procedures that are required for the completion of the tasks under the defined measurements, resources, and constraints are then identified. A general form of a functional hierarchy can be seen in Figure 3. The process of converting the functional hierarchy to a decision making system containing the H-FSM can be seen in Figure 4. Nodes with children are transformed into meta-states containing sub-states. The transitions within these sub-states are determined by the derived procedures. The lowest level nodes of the functional hierarchy form the lowest-level states of the H-FSM where the procedures of each such node can be sufficiently represented by a parameter list (otherwise further decomposition is needed). In order to keep the H-FSM readable and configurable, events are defined as functions of logical flags. Such flags are statused by the lower-level system measurements and represent the basis set of conditions on which events are defined. Also note that each meta-state has a sub-state it defaults to as defined by the procedures for the corresponding node.

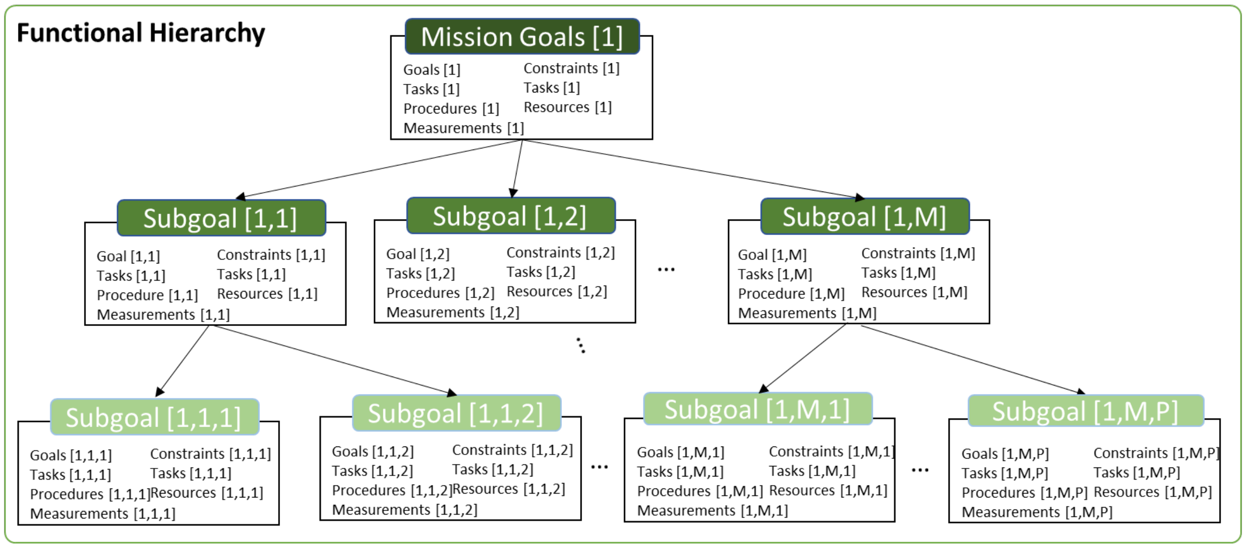


Fig. 3 Generic Functional Decomposition

B. H-FSM Definitions and Notations

Below we list the keywords, their meanings and the notation that we use throughout the paper.

- Meta-state: A state that contains an internal state machine with at least one sub-state.
- Sub-state: A state that is within another meta-state.

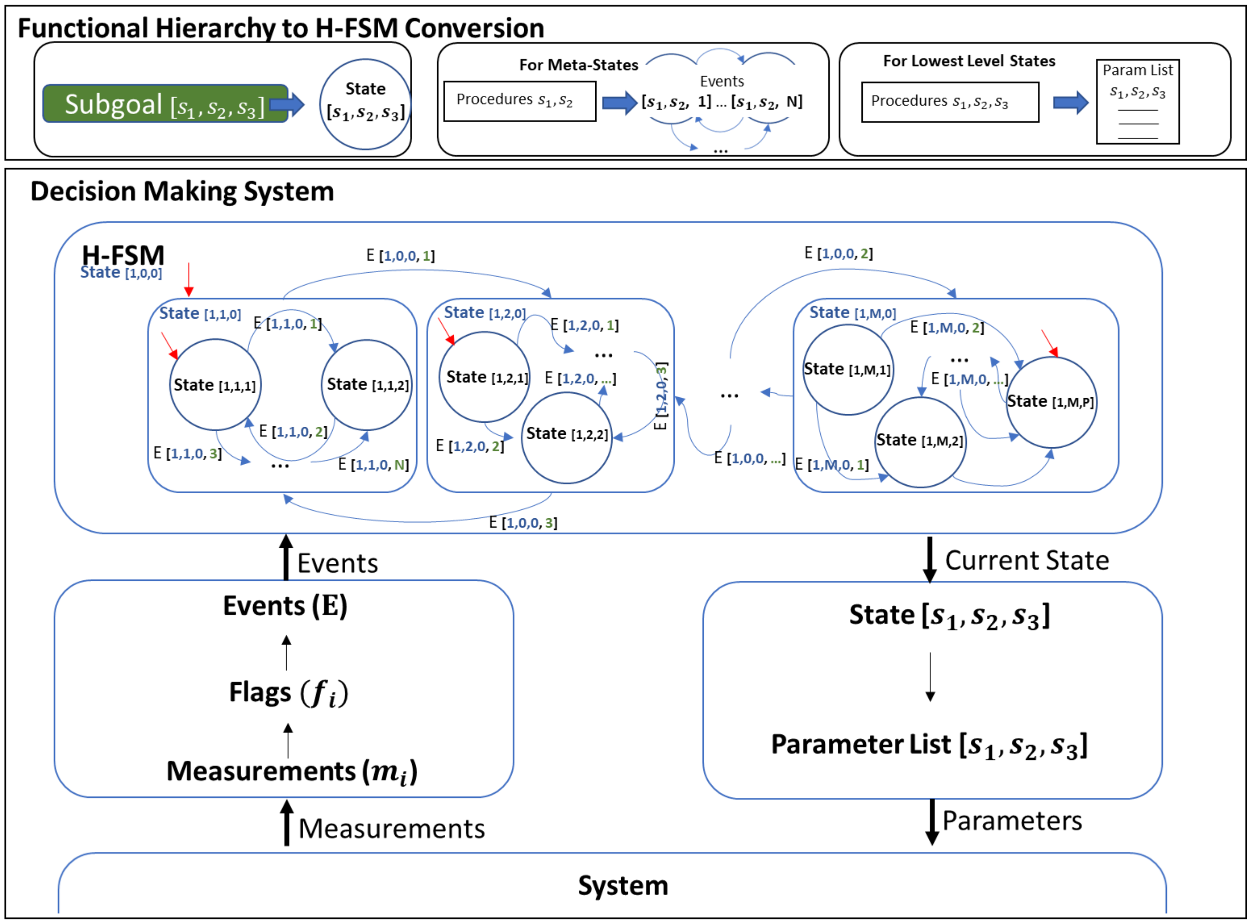


Fig. 4 Decision Making System Construction from Functional Decomposition

- State level: The layer of the H-FSM that the state belongs to. The Top most level is 1 and represents the state machine as a whole.
- Flag: A variable used to map system measurements to Boolean expressions.
- Event: A Boolean variable that is a logical combination of Flags.

For an arbitrary H-FSM that contains L levels, the following notations are used:

- $\mathbf{S}$ is an integer tuple representing an arbitrary state such that $\mathbf{S} = [s_1, s_2, s_3, \ldots s_L]$. s_i is a positive integer value that represents the s_i^{th} state on the i^{th} level of the H-FSM. Note that for a state on the j^{th} level such that $j < L$, $s_{j+1}, \ldots, sL = 0$.
- $\mathbf{E}$ is an integer tuple representing the e^{th} event within a state $\mathbf{S}$ such that $\mathbf{E} = [\mathbf{S}, e]$. Let f_S be a mapping from $\mathbf{S}$ to a state enumeration value $enum_{State}$, and f_E be a mapping from the event tuple to an Event enumeration value $enum_{Event}$.
- f_i is a Boolean value that represents the i^{th} flag state of the system.
- m_i is a real number representing the i^{th} system measurement available to the H-FSM.

C. Hierarchical Finite State Machine Requirement Elicitation in Natural Language

The systematic and thorough decomposition procedure with which the H-FSM was constructed especially proves practical when eliciting requirements on the designed system. Namely, there are four main mechanisms that represent the underlying operation of the H-FSM at any given state: H-FSM state-to-state transitions (Events), meta-state default state entryand Interface parameter list generation. By eliciting these four types of requirements for the entire Decision Making system structure, requirement coverage is provided for all system procedures, goals, and tasks under available measurements, resources, and constraints.

In particular, we elicit the following four requirement types:

1) DM Default State Entry: Each meta-state shall default to a single sub-state upon entry.
2) DM State-to-State Transitions: A state shall transition to a neighboring state if the event triggering the transition is activated and no parent events have been activated.
3) DM Event Statusing: An event shall be set to true if the system is in the relevant originating state and the event flag logic holds true over the last T consecutive time-steps.
4) DM Parameter List Generation: States that are not meta-states shall have parameter lists associated to them.

III. Specification and Formalization of System Requirements

In this Section, we will take the high-level natural language description of the four requirement types provided in the previous Subsection and specify them using the NASA Ames' FRET tool. FRET [4, 5] is an open-source tool [6] for writing, understanding, formalizing, and analyzing requirements.

In practice, requirements are typically written in natural language, which is ambiguous and, consequently, not amenable to formal analysis. Since formal, mathematical notations are unintuitive, requirements in FRET are entered in a restricted natural language named FRETish [7] with rigorous semantics. For each FRETish requirement, FRET generates formulae in a variety of formalisms including metric Linear Temporal Logic (LTL) and Lustre [8] code.

A. The FRETish language

A FRETish requirement is composed using up to six distinct fields (the * symbol designates mandatory fields): 1) `scope` specifies the time intervals where the requirement is enforced, 2) `condition` is a Boolean expression that either a) Upon keyword: triggers the `response` to occur at the time the expression's value becomes true, or is true at the beginning of the scope interval, or b) Whenever keyword: triggers the `response` to occur every time the expression's value is true 3) `component*` is the system component that the requirement is levied upon, 4) `shall*` is used to express that the component's behavior must conform to the requirement, 5) `timing` specifies when the response shall happen, subject to the constraints defined in `scope` and `condition` and 6) `response*` is the Boolean expression that the component's behavior must satisfy.

B. Specification of DM requirement types in FRETish

Below we work on each of the four requirement types and specify them as FRETish templates. We denote template placeholders with angle brackets, i.e., <PLACEHOLDER>.

We start with the **DM Default State Entry** requirement type. Here we want to express that upon entering a meta-state, the FSM shall begin execution at a default sub-state. For this, we use a variable, namely FSM_State_<l> to indicate the state that the FSM is currently at. FSM_State_<l> may be used to indicate either a meta-state or a sub-state depending on the value of l. In particular, $l \in \{1..L\}$ denotes the level from which the state was extracted. We write this requirement template in FRETish as follows:

DM Default State Entry:
Upon FSM_State_<l>=<integer_representing_Meta_State> <DM> `shall` immediately satisfy FSM_State_<$l+1$>=<integer_representing_Default_Sub_State>

In this template, we use five out of the six FRETish fields. In particular, we use the `condition` field to check when the `<DM>` component enters a specific meta-state (Upon FSM_State_<l>=<integer_representing_Meta_State>), which is indicated by an integer. When the component enters the meta-state, then at the same timestep (immediately) the component shall default to a specific sub-state (FSM_State_<$l+1$>=<integer_representing_Default_Sub_State>) (also indicated by an integer).

Next, we continue with the specification **DM State to State Transitions** requirement type. We create a dedicated requirement template in FRETish as follows:

DM State to State Transitions:
Upon FSM_State_<l>=<integer_representing_current_State> & <Event_from_current_state_to_new_State> <DM> `shall` at the next timepoint satisfy FSM_State_<l>=<integer_representing_new_State>

We require that when <DM> enters a particular state (Upon FSM_State_<*l*>=<integer_representing_current_State>) and a certain event is activated (<Event_from_current_state_to_new_State>), then <DM> shall transition to a new state (FSM_State_<*l*>=<integer_representing_new_State>) at the same time step (immediately).

Next, we specify FRETish templates for the **DM Event Statusing** requirement type:

DM Event Statusing:
<DM> `shall` always satisfy E_<Event> <=> persisted(T-1,<Flags_Logical_Expression_Relevant_to_Event>)

Notice the use of the persisted function in the `response` field of **DM Event Statusing**. The semantics of persisted is as follows: **persisted(n,p)** becomes true at the time the Boolean expression **p** has held in the previous n time steps and also holds at the current time step, for a total of **n+1** time steps, meaning a duration of **n** time units. Additionally notice that both of these templates use the equivalence operator <=>.

Finally, we specify the last requirement template, namely **DM Parameter List Generation**. This template describes that states, which do not belong to meta-states, shall be associated with a list of parameters with specific valuation upon entry in a state.

DM Parameter List Generation:
Whenever FSM_State_<*L*>=<integer_representing_bottom_State> <DM> `shall` immediately satisfy <param_list_entry1_name>=<Entry1Value> & <param_list_entry2_name>=<Entry2Value> & <param_list_entryN_name>=<EntryNValue>

The immediately timing requires that the response is satisfied at the same time point whenever the condition (FSM_State_<*L*>=<integer_representing_bottom_State>) is true. FSM_State_<*L*> represents the state at the bottom level of the FSM.

IV. Verification of Formalized Requirements

The design presented in Section II can be implemented in a straightforward manner using model-based development tools. More importantly, resulting models from this development process are amenable to automated formal analysis. The main objective at this step is to provide a formal proof of the correctness of the model, considering instantiations of the formal requirement types defined in Section III.B. Our workflow depends on MATLAB Simulink/Stateflow for the implementation of models, while our in-house solution, namely CoCoSim [9], enables scalable formal compositional verification of such models.

CoCoSim (Contract-based Compositional verification of Simulink models) is an open-source plugin for MATLAB Simulink/Stateflow, that allows users to formally verify requirements expressed in the form of Assume-Guarantee contracts [10, 11]. To tackle scalability issues, verification can be achieved using compositional reasoning over contracts defined for the system as a whole, as well as its subsystems. Complexity induced by verbose subsystems can be abstracted away, by replacing the model with its corresponding contract in the proof. Besides compositional verification, CoCoSim provides means to translate Simulink/Stateflow models into equivalent implementations in Lustre and C, and limited support for test-case generation.

A particularly important CoCoSim feature for the purposes of this work, provides the ability for users to import existing formalized requirements, with CoCoSim automatically attaching them to relevant Simulink models as synchronous observers [12]. Input requirements are expressed in CoCoSpec, a Lustre specification format that adheres to the Assume-Guarantee paradigm [13]. Previous work by Mavridou et al. [14, 15] implemented and applied this feature to industrial-level problems, while establishing a stronger connection between FRET and CoCoSim. As a result, FRET users can export FRETish requirements into CoCoSpec, which can be subsequently attached to a Simulink model using CoCoSim's translation scheme to Simulink code. As an example, the State to State Transition requirement type from Section III.B is automatically translated by FRET into the following guarantee in CoCoSpec:

```
-- DM State to State Transition:
guarantee H((((FSM_State_<l> = <integer_representing_current_State>) and
            <Event_from_current_state_to_new_State>) and
                ((YtoPre( not ((FSM_State_<l> = <integer_representing_current_State>) and
                    <Event_from_current_state_to_new_State>))) or FTP)) =>
                        (FSM_State_<l> = <integer_representing_new_State>));
```

The CoCoSpec guarantee is a direct result of the translation of the original FRETish requirement in Past Time Metric Linear Temporal Logic (pmLTL). The pmLTL formula is expressed in Lustre, with the node (function) calls *H* and *YtoPre* corresponding to the application of predefined Lustre nodes for the pmLTL operators 'Historically' and 'Yesterday'. *FTP* is an internal boolean variable that is only true in the first time point of a given execution.

Given the formal model and the imported FRETish requirements for the Decision Making system, CoCoSim can be invoked to formally verify the model against the requirements. During this process, CoCoSim translates the combination of the model and the contract into equivalent code in Lustre, enabling thus the usage of state-of-the art model checkers such as Kind 2 [16]. The result of a complete verification task is either a proof that the model conforms to the given requirements or, otherwise, an indication of how (a subset of) the requirements are violated using actual execution traces as counterexamples.

V. Case Study: Dynamic Zonal Relay Stage of the Troupe System

This section will present the DM design process followed for the NASA Troupe project case study. A functional decomposition is performed on the Dynamic Zonal Relay (DZR) stage of the Troupe system, followed by H-FSM construction and implementation in Simulink. Requirements are formalized for DM in FRET and verified using CoCoSIM.

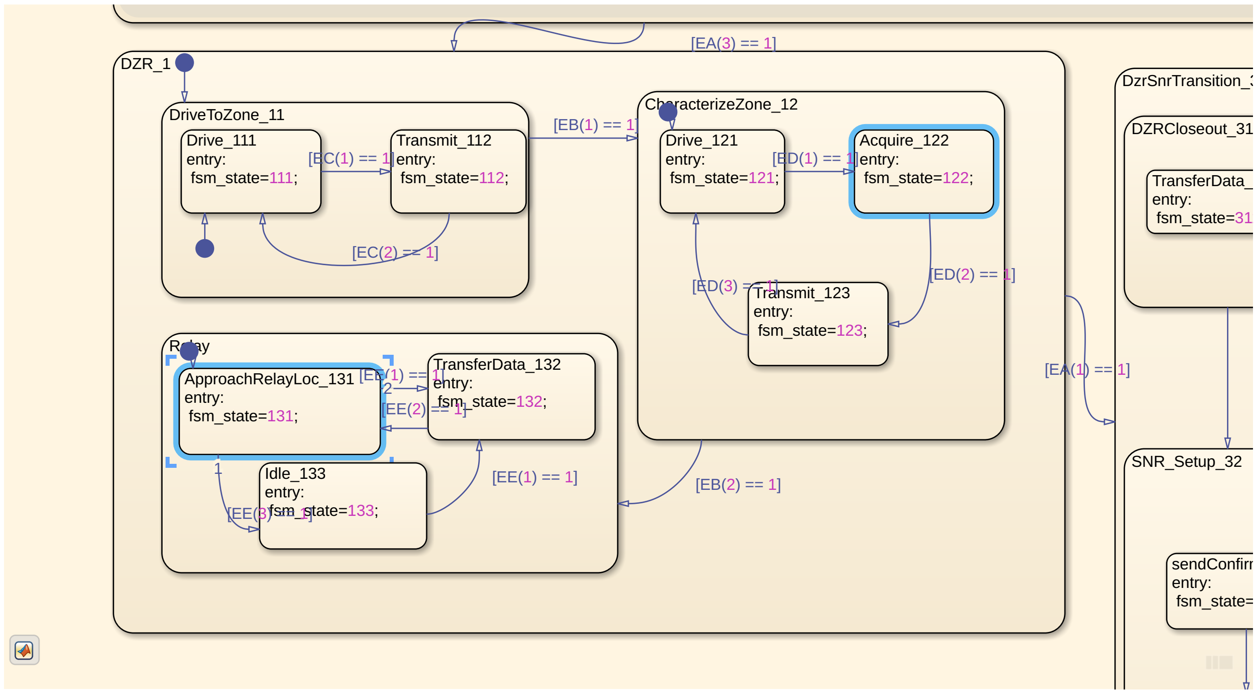


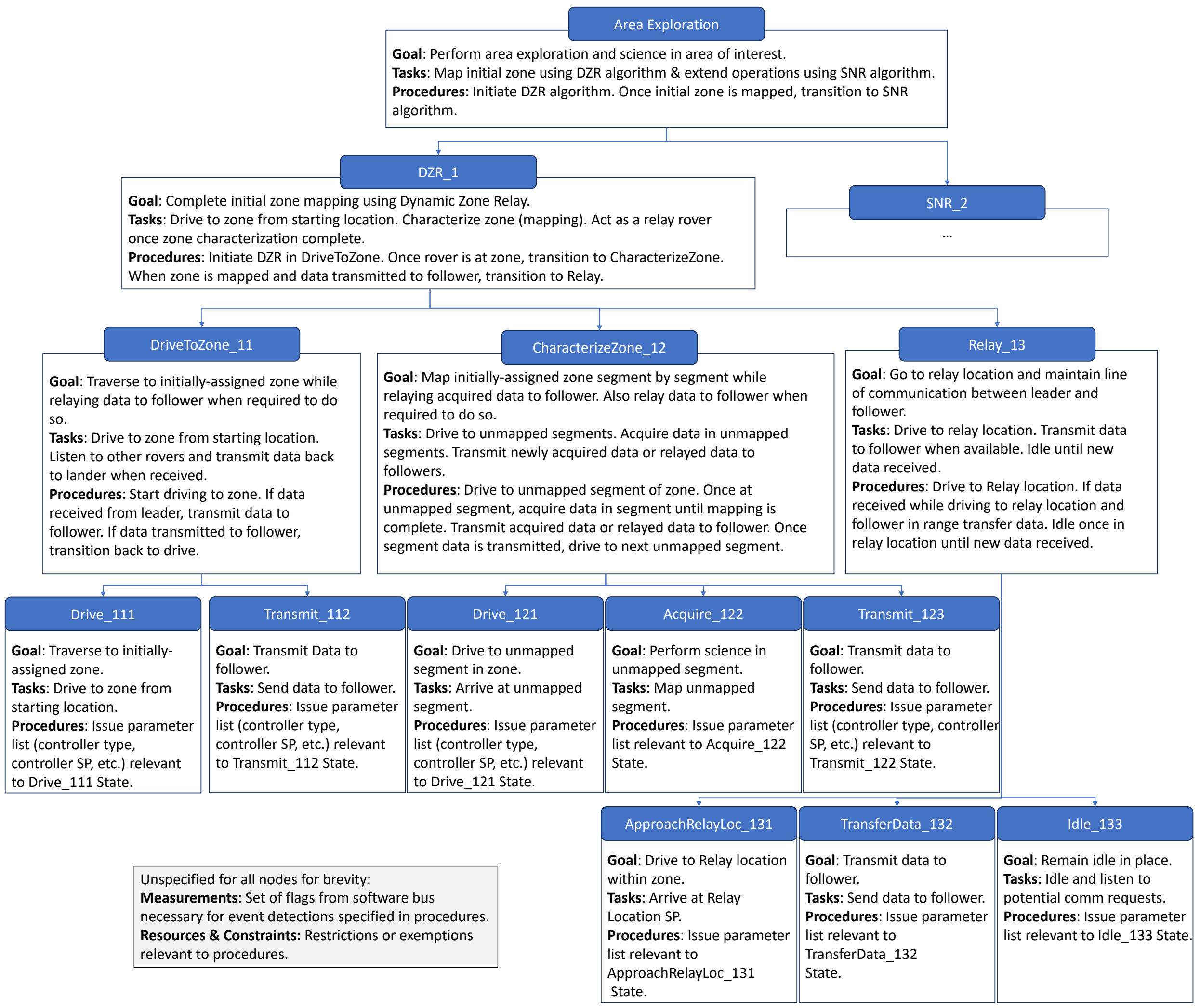
Fig. 5 The DZR meta-state in the H-FSM subsystem.

A. Case Study Description

Troupe is a NASA Ames Research Center Project that aims to develop a system of 4-10 rovers capable of coordination for the autonomous mapping of their environment. The autonomy, coordination, and collaboration algorithm for the rovers is based on the Dynamic Zonal Relay (DZR) and Sneaker-Net Relay (SNR) algorithms developed at NASA's Jet Propulsion Lab (JPL) [17].

The DZR+SNR algorithm is made up of two distinct stages.

In the first stage, DZR, each rover is assigned to a zone along an area of interest and is tasked with performing science (e.g. mapping) while maintaining communication within that zone. In the second stage, SNR, rovers coordinate in a line formation to traverse to zones beyond the communication distance with the base station and perform science tasks. Within SNR, rovers are required to coordinate to relay data back to the base-station.


Area Exploration

Goal: Perform area exploration and science in area of interest.
Tasks: Map initial zone using DZR algorithm & extend operations using SNR algorithm.
Procedures: Initiate DZR algorithm. Once initial zone is mapped, transition to SNR algorithm.

DZR_1

Goal: Complete initial zone mapping using Dynamic Zone Relay.
Tasks: Drive to zone from starting location. Characterize zone (mapping). Act as a relay rover once zone characterization complete.
Procedures: Initiate DZR in DriveToZone. Once rover is at zone, transition to CharacterizeZone. When zone is mapped and data transmitted to follower, transition to Relay.

SNR_2

...

DriveToZone_11

Goal: Traverse to initially-assigned zone while relaying data to follower when required to do so.
Tasks: Drive to zone from starting location. Listen to other rovers and transmit data back to lander when received.
Procedures: Start driving to zone. If data received from leader, transmit data to follower. If data transmitted to follower, transition back to drive.

CharacterizeZone_12

Goal: Map initially-assigned zone segment by segment while relaying acquired data to follower. Also relay data to follower when required to do so.
Tasks: Drive to unmapped segments. Acquire data in unmapped segments. Transmit newly acquired data or relayed data to followers.
Procedures: Drive to unmapped segment of zone. Once at unmapped segment, acquire data in segment until mapping is complete. Transmit acquired data or relayed data to follower. Once segment data is transmitted, drive to next unmapped segment.

Relay_13

Goal: Go to relay location and maintain line of communication between leader and follower.
Tasks: Drive to relay location. Transmit data to follower when available. Idle until new data received.
Procedures: Drive to Relay location. If data received while driving to relay location and follower in range transfer data. Idle once in relay location until new data received.

Drive_111

Goal: Traverse to initially-assigned zone.
Tasks: Drive to zone from starting location.
Procedures: Issue parameter list (controller type, controller SP, etc.) relevant to Drive_111 State.

Transmit_112

Goal: Transmit Data to follower.
Tasks: Send data to follower.
Procedures: Issue parameter list (controller type, controller SP, etc.) relevant to Transmit_112 State.

Drive_121

Goal: Drive to unmapped segment in zone.
Tasks: Arrive at unmapped segment.
Procedures: Issue parameter list (controller type, controller SP, etc.) relevant to Drive_121 State.

Acquire_122

Goal: Perform science in unmapped segment.
Tasks: Map unmapped segment.
Procedures: Issue parameter list relevant to Acquire_122 State.

Transmit_123

Goal: Transmit data to follower.
Tasks: Send data to follower.
Procedures: Issue parameter list (controller type, controller SP, etc.) relevant to Transmit_122 State.

ApproachRelayLoc_131

Goal: Drive to Relay location within zone.
Tasks: Arrive at Relay Location SP.
Procedures: Issue parameter list relevant to ApproachRelayLoc_131 State.

TransferData_132

Goal: Transmit data to follower.
Tasks: Send data to follower.
Procedures: Issue parameter list relevant to TransferData_132 State.

Idle_133

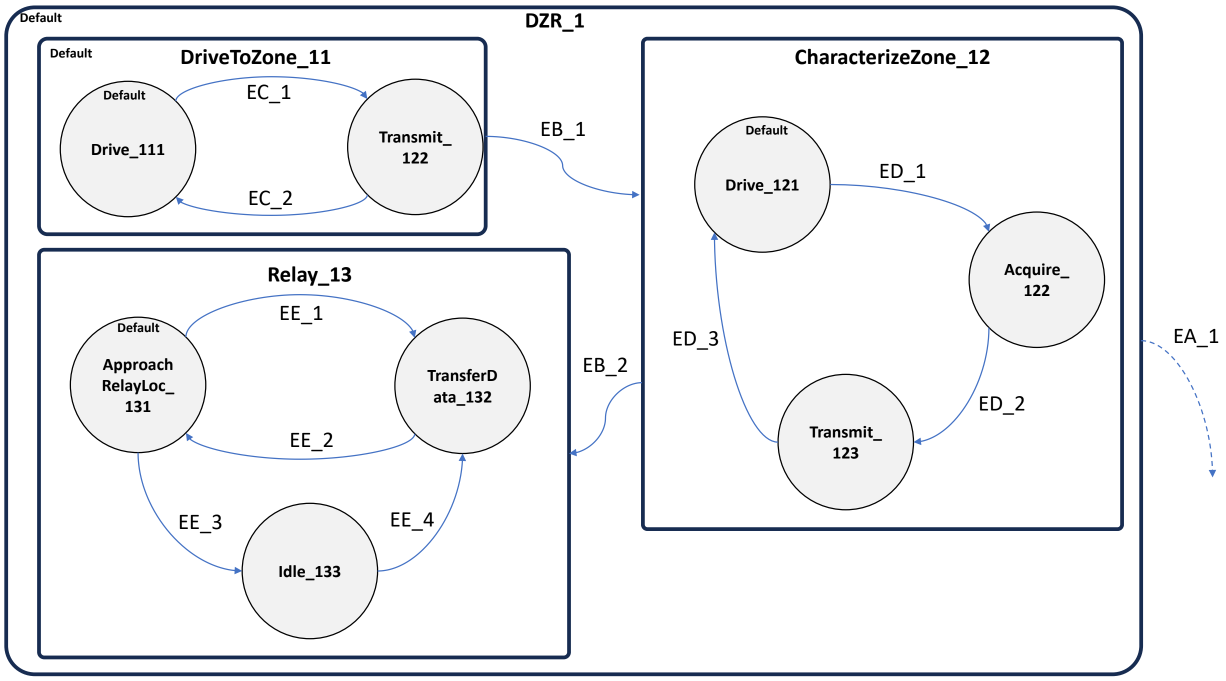
Goal: Remain idle in place.
Tasks: Idle and listen to potential comm requests.
Procedures: Issue parameter list relevant to Idle_133 State.

Unspecified for all nodes for brevity:
Measurements: Set of flags from software bus necessary for event detections specified in procedures.
Resources & Constraints: Restrictions or exemptions relevant to procedures.

Fig. 6 DZR Functional Hierarchy resulting from system decomposition.

Following the described System Functional Decomposition process described in Section II, the algorithms specified in [17] are decomposed into a functional hierarchy. DZR_1 is first decomposed to be made up of three distinct phases: DriveToZone_11, Characterize Zone_12, and Relay_13. A rover is first tasked with driving to a zone it is assigned to, then characterize the zone, and relay the data back to the follower rover. Within DriveToZone_11, a given rover is tasked with driving to its designated zone (Drive_111) while listening to data from other rovers and transmitting back to followers (Transmit_112) if data is being relayed to the lander. Once in the designated zone, the rover transitions to CharacterizeZone_12 where the goal of the rover is to drive (Drive_121) in the exploration pattern within the zone, acquire data (Acquire_122) on a segment-by-segment basis, and regularly transmit (Transmit_123) data when segments are mapped or when data is being received from a leader rover. On completion of zone mapping, the rover then transitions to the Relay_13 phase where it is tasked with traversing to a relay location (ApproacRelateLoc_131) within its zone, transfer data (TransferData_132) back to the lander, and remain idle (Idle_133) until SNR phase of the mission is initiated. While remaining idle the rover will transmit (Transmit_112) data towards the lander when it is received. The Functional Hierarchy structure can be seen in Figure 6. The H-FSM based DM app design can be seen in Figure 7, whereas the Simulink implementation is in Figure 5. The DZR implementation highlighted in Figure 7 results in the generation of 35 requirements. 5 of which are *'DM Default State Entry'* requirements, 11 *'DM State to State Transitions'* requirements, 11 *'DM Event Statusing'* requirements, and 8 *'DM Parameter List Generation'* requirements.

For the purposes of this case study, we will focus on the DZR H-FSM implementation. We present the formalized FRET requirements, as well as verification results using CoCoSim.



Events
EA_1: Flag for DZR completion.
EB_1: Flag logic for arrival at designated zone.
EB_2: Flag logic for completion of zone characterization.
EC_1: Flag logic for receipt of data to be transmitted to follower.
EC_2: Flag logic for completion of data transmission to follower.
ED_1: Flag logic for arrival at unmapped segment.
ED_2: Flag logic for completion of segment mapping.
ED_3: Flag logic for completion of mapped segment data transmission.
EE_1: Flag logic for receipt of data to be transmitted to follower.
EE_2: Flag logic for completion of data transmission to follower.
EE_3: Flag logic for arrival to relay location.
EE_4: Flag logic for receipt of data to be transmitted to follower.

Parameter Lists
Each State on the lowest level is mapped to a parameter list that statuses apps interfacing with Decision Making.

Fig. 7 DZR Functional Hierarchy resulting from system decomposition.

B. FRET requirements for DM

Using the templates in Section III.B, we derived 196 FRET requirements for the DM app. Note that these requirements are across all phases of operation and not only DZR. In the following, We provide example requirements, as well as a description of the intended behavior that each captures. Note that we use labels to refer to different (meta/sub-)states, instead of their corresponding integer value.

The requirement below captures the initial sub-state of the Relay phase in the DZR meta-state:

DM Default State Entry (DZR, relay phase):
Upon FSM_State_2 = DZR_Relay DM `shall` immediately satisfy FSM_State_3 = DZR_Relay_ApproachRelayLoc

During the DZR algorithm, the rover is expected to transmit data from the designated zone, assuming that the necessary information has been previously acquired and that input event vector ED was observed to have its second element set to true (*ED_2*). This is captured as a state transition requirement, as shown below:

DM State to State Transitions (DZR, transmit acquired data):
Upon (FSM_State_3 = DZR_CharacterizeZone_Acquire & ED_2) DM `shall` at the next timepoint satisfy FSM_State_3 = DZR_CharacterizeZone_Transmit

As previously mentioned, each event signal in the DM app has a predefined logic over the input flags. The requirement below showcases this for the aforementioned input vector ED, and more specifically the truth value of the second element of the vector, which is set to true if the segment the rover is in has been characterized (captured by flag *F_segmentCharacterizationComplete*):

DM Event Statusing (ED event):
DM `shall` always satisfy ED_2 <=> persisted(3,F_segmentCharacterizationComplete)

The DM app outputs relevant information related to its current state for other apps to consume (e.g. health monitor, rover dynamics, PID control systems). More specifically, each leaf-level state of the DM H-FSM determines the values of a list of output parameters. In the example below, we show how the state *DZR_CharacterizeZone_Acquire* dictates the values of parameters related to the mode of the underlying controller (*controllerType*), the *activity* the rover performs and finally its *velocity*:

DM Parameter List Generation (DZR, data acquired):
Whenever FSM_State_3 = DZR_CharacterizeZone_Acquire DM `shall` immediately satisfy (controllerType = 1 & activity = 4 & velocity = 1.0)

C. Formal Verification with CoCoSim

Each one of the 196 FRET requirements were imported in the DM Simulink model using the requirements importing feature in CoCoSim. The result of the importing process is the creation of CoCoSim *contracts* i.e., constructs that contain a Simulink representation for each requirement. The contracts are automatically attached to the corresponding Simulink subsystem. Figure 8 shows one such example of a CoCoSim contract attached to the H-FSM implementation of the DM app. Consequently, the Simulink model is amenable to analysis via formal verification against the attached contracts. More specifically, CoCoSim leverages the Kind 2 model checking tool to verify the implementation, and provide diagnostic results.

Table 1 shows the verification results per FRET template. The total analysis time was 48 minutes, 23 seconds, including the time taken for CoCoSim to translate the entire DM app to Lustre, the input format of Kind 2. As Table 1 shows, requirements of all but one template were proved to be valid. More specifically, requirements of the State to State transition template were originally proved to not hold. The verification results showed that additional constraints were missing from each requirement, that are necessary to ensure that the expected transition will indeed occur.

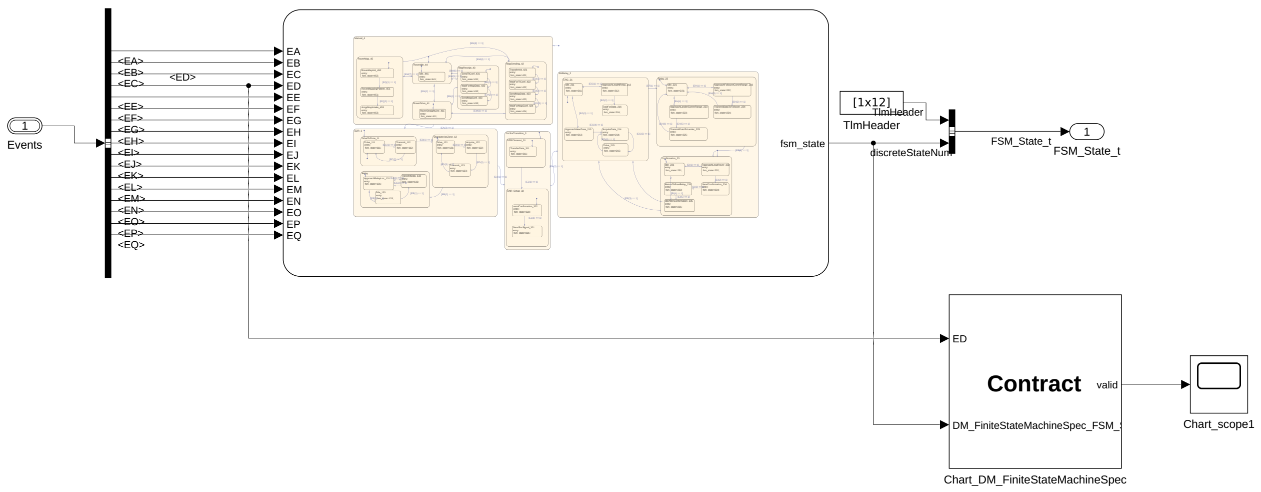


Fig. 8 *H-FSM* Stateflow model with attached CoCoSim contract.

Table 1 Verification results for DM app.

FRET DM Template	**No. of requirements**	**Avg. Analysis Time (seconds)**	**Verification Result**
Default State Entry	17	6.55	Valid
State to State transition	95	12.87	Invalid
Event Statusing	45	22.01	Valid
Parameter List Generation	39	6.05	Valid

A crucial artifact towards understanding this issue was the set of counterexamples provided by Kind 2 for the invalid requirements. Given an invalid requirement, a counterexample is an execution trace of the implementation, wherein the requirement can be shown to be violated. As an example, consider the following requirement from Section V.B:

Upon (FSM_State_3 = DZR_CharacterizeZone_Acquire & ED_2) DM `shall` at the next timepoint satisfy FSM_State_3 = DZR_CharacterizeZone_Transmit

The aforementioned requirement captures an expected transition between states *DZR_CharacterizeZone_Acquire* and *DZR_CharacterizeZone_Transmit*, under the premise that the second element of the input event vector ED (*ED_2*) is set to true. The corresponding verification task showed that the requirement is invalid, and a counterexample was generated. Table 2 shows the counterexample trace, consisting of four state transitions starting from the initial state of the hierarchical finite state machine in the DM app (*Manual_RoverIdle_Idle*). At the third step of the execution, the state machine is in state *DZR_CharacterizeZone_Acquire* and considering the requirement, the expected next state should have been *DZR_CharacterizeZone_Transmit*, given that *ED_2* is true. In the last step of the counterexample, input *ED_2* is indeed true, but the state machine exercises an unexpected transition into state *DZR_Relay_ApproachRelayLoc*, which initiates the 'Relay' phase of the DZR algorithm. A review of the state machine quickly uncovered the reasons. First, the requirement in its original form considers the value of the *ED* event vector in the previous execution step, whereas the Stateflow model considers only current inputs for its state transitions. Second, in the meta-state that corresponds to

Table 2 Counterexample trace for invalid requirement in the H-FSM model.

Signal	**Step 0**	**Step 1**	**Step 2**	**Step 3**	**Step 4**
EA_1	false	false	false	false	false
...	...	...	...	...	...
EB_2	true	false	false	false	true
...	...	...	...	...	...
ED_2	false	true	false	true	true
FSM_State	Manual_RoverIdle_Idle	DZR_DriveToZone_Drive	DZR_Characterize_Zone_Drive	DZR_Characterize_Zone_Acquire	DZR_Relay_ApproachRelayLoc
Requirement	true	true	true	true	false

the DZR algorithm, the unexpected transition has a higher priority than the one defined in our example requirement. As such, when *EB_2* is true, the state machine prioritizes the transition to *DZR_Relay_ApproachRelayLoc*. Figure 5 shows the exact representation of the DZR meta-state, where transitions at higher state levels have higher priorities when compared to states in lower levels.

Considering the above, each instantiated requirement of the State to State transition template needed to be refined to address the two observations. More specifically, their preconditions needed to be strengthened with additional constraints that prevent transitions with higher priority to be possible when verifying a given requirement. Additionally, the requirement should consider the current inputs instead of the corresponding values in the previous execution step. For our example, these adjustments meant that we had to refine the requirement into the following:

Upon (FSM_State_3 = DZR_CharacterizeZone_Acquire) DM `shall` at the next timepoint satisfy (ED_2 & ! EA_1 & ! EB_2) => FSM_State_3 = DZR_CharacterizeZone_Transmit

The updated requirements were subsequently proved to be valid by CoCoSim, following the same verification workflow.

VI. Conclusion

We introduced a methodology for formally designing, specifying, and verifying a Decision Making application intended for autonomous systems. Our demonstration showcased the functional decomposition of the system's overarching objectives through the utilization of a functional hierarchy. This resultant structure was then translated into a Hierarchical Finite State Machine, with functionality comprehensively addressed by four types of requirements expressed in natural language. The proposed approach and the ensuing requirements enabled us to employ the connection between the FRET and CoCoSim tools to specify, formalize and verify properties of interest against the implementation. In particular, we first elicited four requirement types in natural language and then specified these requirement types in the FRETish language. To cover the complete DM app, we instantiated 196 requirements from these four templates and then FRET automatically generated CoCoSpec specifications that cover all 196 requirements. The CoCoSpec specifications were subsequently given as input to the CoCoSim tool, where model checking actions proved the requirements' validity against the Simulink model of the DM app. The outcome is a comprehensive end-to-end process that can be applied and customized for the design of Decision Making systems, ensuring formal requirements and verifiable implementations.

References

[1] Kurt, A., and Özgüner, Ü., "Hierarchical finite state machines for autonomous mobile systems," *Control Engineering Practice*, Vol. 21, No. 2, 2013, pp. 184–194.

[2] Hejase, M., Oguz, A. E., Kurt, A., Ozguner, U., and Redmill, K., "A hierarchical hybrid state system based controller design approach for an autonomous UAS mission," *16th AIAA Aviation Technology, Integration, and Operations Conference*, 2016, p. 3294.

[3] Acar, L., and Ozguner, U., "Design of knowledge-rich hierarchical controllers for large functional systems," *IEEE transactions on systems, man, and cybernetics*, Vol. 20, No. 4, 1990, pp. 791–803.

[4] Giannakopoulou, D., Mavridou, A., Pressburger, T., Rhein, J., Schumann, J., and Shi, N., "Formal Requirements Elicitation with FRET," *REFSQ*, 2020.

[5] Katis, A., Mavridou, A., Giannakopoulou, D., Pressburger, T., and Schumann, J., "Capture, Analyze, Diagnose: Realizability Checking Of Requirements in FRET," *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part II*, Springer, 2022, pp. 490–504.

[6] "FRET: Formal Requirements Elicitation Tool," , 2023. URL `https://tinyurl.com/ycxe9fv4`.

[7] Giannakopoulou, D., Pressburger, T., Mavridou, A., and Schumann, J., "Automated formalization of structured natural language requirements," *Information and Software Technology*, Vol. 137, 2021, p. 106590. https://doi.org/10.1016/j.infsof.2021.106590.

[8] Jahier, E., Raymond, P., and Halbwachs, N., "The lustre v6 reference manual," *Verimag, Grenoble, Dec*, 2016.

[9] "CoCoSim: Contract-based Compositional verification of Simulink models," , 2023. URL https://github.com/NASA-SW-VnV/cocosim.

[10] Bourbouh, H., Garoche, P.-L., Loquen, T., Noulard, É., and Pagetti, C., "CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models," *Embedded Real Time Systems (ERTS) 2020*, , No. ARC-E-DAA-TN74591, 2020.

[11] Bourbouh, H., Garoche, P.-L., Garion, C., Gurfinkel, A., Kahsai, T., and Thirioux, X., "Automated analysis of Stateflow models," *21st International conference on logic for programming, artificial intelligence and reasoning (LPAR 2017)*, Vol. 46, 2017, pp. 144–161.

[12] Halbwachs, N., Lagnier, F., and Raymond, P., "Synchronous observers and the verification of reactive systems," *Algebraic Methodology and Software Technology (AMAST'93) Proceedings of the Third International Conference on Algebraic Methodology and Software Technology, University of Twente, Enschede, The Netherlands 21–25 June 1993*, Springer, 1994, pp. 83–96.

[13] Champion, A., Gurfinkel, A., Kahsai, T., and Tinelli, C., "CoCoSpec: A mode-aware contract language for reactive systems," *Software Engineering and Formal Methods: 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, Springer, 2016, pp. 347–366.

[14] Mavridou, A., Bourbouh, H., Garoche, P. L., Giannakopoulou, D., Pessburger, T., and Schumann, J., "Bridging the gap between requirements and simulink model analysis," *Joint 26th International Conference on Requirements Engineering: Foundation for Software Quality Workshops, Doctoral Symposium, Live Studies Track, and Poster Track*, 2020.

[15] Mavridou, A., Bourbouh, H., Giannakopoulou, D., Pressburger, T., Hejase, M., Garoche, P.-L., and Schumann, J., "The ten lockheed martin cyber-physical challenges: formalized, analyzed, and explained," *2020 IEEE 28th International Requirements Engineering Conference (RE)*, IEEE, 2020, pp. 300–310.

[16] Champion, A., Mebsout, A., Sticksel, C., and Tinelli, C., "The Kind 2 model checker," *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, Springer, 2016, pp. 510–517.

[17] Vaquero, T., Troesch, M., and Chien, S., "An approach for autonomous multi-rover collaboration for mars cave exploration: Preliminary results," *International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS 2018). Also appears at the ICAPS PlanRob*, 2018.