ARL-TR-10321 ● APR 2026

The Development of a Modular Trigger Response System

by Shan G. Lakhmani, David Chhan, Justin Brody, Bret Kellihan, Justin Jagielski, Joseph Campanelli, Joe Rexwinkle, and Gregory Gremillion

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-10321 ● APR 2026

The Development of a Modular Trigger Response System

Shan G. Lakhmani, David Chhan, Joe Rexwinkle, and Gregory Gremillion
DEVCOM Army Research Laboratory

Justin Brody
Fibertek, Inc.

Bret Kellihan, Justin Jagielski, and Joseph Campanelli
DCS Corp.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
April 2026	Technical Report	1 November 2022	30 September 2025

4. TITLE AND SUBTITLE
The Development of a Modular Trigger Response System

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Shan G. Lakhmani, David Chhan, Justin Brody, Bret Kellihan, Justin Jagielski, Joseph Campanelli, Joe Rexwinkle, and Gregory Gremillion

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DEVCOM Army Research Laboratory
ATTN: FCDD-RLA-FD
Aberdeen Proving Ground, MD 21005

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-10321

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCIDs: Shan Lakhmani, 0000-0001-6052-439X; David Chhan, 0000-0003-2470-8663; Joe Rexwinkle, 0000-0002-3856-101X; Gregory Gremillion, 0000-0002-0205-688X

14. ABSTRACT
We developed a Modular Trigger Response System that allows Soldiers to customize the deployment of human and system resources according to mission factors, platoon-specific operating procedures, and personal preference. We document the developmental process to date, including the following: the initial design of the software's back end, the design of a front end utilizing a trigger-action programming approach to end-user development, and the use of this software in a simulation experiment. Subsequently, we describe the redesign of the software—based on experiment results and Soldier feedback—the use of the software in another simulation experiment, and feedback from that experiment. Finally, we discuss our latest work in using large language models to expand the software capability to include voice input. Documentation for the software (back end, schema, and front end) are included in the appendixes.

15. SUBJECT TERMS
Humans in Complex Systems, trigger-action programming, end-user development, interface design, next-generation combat vehicle, workload transition

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	81

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Shan G. Lakhmani	(520) 691-4900

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

1. Introduction

The management of resources within human–machine integrated (HMI) formations—such as crewed vehicle platforms, uncrewed vehicle platforms, platform subsystems, AI-enabled mobility technologies, sensing technologies, and humans in need of tasking—can be cognitively burdensome, which, in turn, can hamper performance during situations with high task loads. Automated tactical decision making has been proposed as a potential solution to future performance problems in HMI formation tasks. The Modular Trigger Response System (MTRS) is one approach we use to implement automated tactical decision making, where non-expert users can customize system functions and behaviors, using an end-user programming environment.

The MTRS was developed as an expert system that would allow users to rapidly pair METT-TC (Mission, Enemy, Terrain, Troops, Time, and Civil Considerations) factors with automated behaviors. Trigger–action programming ([TAP] "if," "then"), a common end-user development (EUD) approach (Ur et al. 2014), allows users to couple triggers and actions together to create "rules" that the system will follow, even when the users are not directly controlling the system. These rules can include relatively simple actions, but they can be extended to include more complex and specific outcomes. The initial drive to develop this technology was to allow Soldiers to set conditions for automated re-tasking in the next-generation combat vehicle (NGCV), although the goal for the program has been to create a software that can be implemented on other platforms. Unlike an autonomy stack, MTRS is meant to act as a watchdog for a system. Using a schema model of expected inputs and system capabilities, the MTRS sits on top of the system, but separate from it, observing inputs from the system's sensing models. When the MTRS detects a trigger used in an active rule, the MTRS generates appropriate outputs to the system's external models that will execute desired behaviors. Because the MTRS is external to the system—all computation for sensing and automated behavior occurs outside the MTRS—it is a lightweight solution that can be implemented with several different systems. The MTRS can also adjust to additional capabilities, only needing updates to the system schema, so that it can link new sensors and capabilities to the sensor and behavioral models it observes. The MTRS is a lightweight, modular, and scalable solution that can allow end-users to customize system responses, so that they can respond effectively during busy, high-stress situations.

In this report, we discuss the underlying concept of EUD and how the choice of TAP as its implementation was expected to yield effective performance. Next, we discuss the initial design and development of the MTRS and its implementation

into a simulation experiment (SIMEX) for the NGCV. Afterward, we discuss MTRS's redesign, the updates to its development, and its integration into a subsequent SIMEX for the NGCV. We conclude with the lessons learned from that simulation work, how that informed work on an increasingly usable MTRS interface, and how the software can be implemented with other, non–NGCV-oriented systems.

2. Background

One of the motivations behind the development of the MTRS was to alleviate negative repercussions of a workload transition. Workload can be described as the proportion of attentional resources demanded by a task or set of tasks (Stanton et al. 2017, p. 283), which can be mediated by task demands, external support, or past experience (Tao et al. 2019). However, the attentional resources demanded by a task can vary over time, so the attentional resources demanded by a task can suddenly change (Moacdieh et al. 2020). The shift between two workload levels in a continuous period is defined as workload transition (Devlin et al. 2023). Technology can be used to mitigate changes in task demand. Unfortunately, integrating humans and autonomous systems can, in fact, create *new* sources of cognitive load (Johnson and Marathe 2025). Creating a tool that would allow users to customize the automation to activate during periods of workload transition could potentially mitigate the inconsistent, and potentially deleterious, effects that workload transition has on performance (Moacdieh et al. 2020; Devlin et al. 2023). Allowing end-users to customize when autonomous systems would activate to mitigate workload transition allows these autonomous systems to meet the various cognitive needs of different users at different times. To deal with the fact that few Soldiers would have the coding skills to program these autonomous skills directly, an interface using EUD was explored.

EUD has been defined as "a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artefact" (Lieberman et al. 2006, p. 2). End-users may or may not have programming experience, so any techniques or tools provided to them must address the barriers to novices (Reisinger et al. 2017). Early approaches to EUD focused on spreadsheet-oriented approaches (e.g., Lotus 1-2-3, Excel) and visual programming approaches (e.g., Scratch) (Paternò and Santoro 2019). The mainstreaming of technology related to the internet of things—which often involved objects with sensors and actuators that could interact with other objects, human beings, and the environment—facilitated greater use of TAP approaches to EUD. The TAP approach utilizes if–then structures to allow users to specify a system's state or behavior to act as an event (i.e., "trigger") and

use that to specify a corresponding response (i.e., "action") that will be taken once the event occurs (Ur et al. 2014). These combinations of triggers and actions can be referred to as rules (Reisinger et al. 2017). A TAP approach worked well for the MTRS, given the conceptual simplicity of TAP, coupled with the sensor-based systems and desired autonomous vehicle behavior in the Information for Mixed Squads (INFORMS) laboratory. As such, the MTRS was developed to allow human crew members to customize their vehicle's responses to events that occurred on the battlefield (Rexwinkle et al. 2024).

A major challenge for the NGCV project has been determining how to successfully integrate intelligent, autonomous technologies into military teams, using a mobile platform (Perelman et al. 2020a). The INFORMS laboratory has sought to examine the NGCV concept by simulating—among the smallest unit size that allows the testing of system- and unit-level interactions—dynamic task orchestration across members of a crew (Perelman et al. 2020a). To simulate these dynamic task orchestrations across teammates, the laboratory models two sections, each section consisting of one manned control vehicle (MCV) and two robotic combat vehicles (RCVs). To crew these simulated sections, the laboratory contains two sets of seven crew stations, each representing a Section (Perelman et al. 2020a). The frontmost station seats the simulator's section commander, and the remaining stations seat three dyads made up of gunner and driver pairs, with the first pair controlling the MCV, the next pair controlling the first RCV, and the third pair controlling the second RCV; two framed MCV mockups house seven crew stations each, totaling 14 seats (Perelman et al. 2020a; Crowell et al. 2023). Before the development of the MTRS, studies conducted in the INFORMS laboratory assessed the effectiveness of multiple technological interventions (e.g., transparent route planners, integrated playbook style command and control capabilities, and aided target recognition systems [(AiTRs]) in facilitating the "teaming" of humans and autonomous systems (Perelman et al. 2020a; Cox et al. 2022; Crowell et al. 2023). In the first of these sets of experiments, two sets of four civilian volunteers completed two 90-min blocks of scenarios in a simulated environment over an extended period of time (Chhan et al. 2020; Perelman et al. 2020b). The following year, 6 Soldier volunteers completed 14 simulated missions over the course of 2 weeks, with a 7th Soldier volunteer participating in the 2nd week (Cox et al. 2022). This report discusses the development of the design, development, and testing of the MTRS in SIMEX scenarios over the course of the next 2 years.

3. Design, Development, and Testing of MTRS: Year 1

The development of the MTRS was a multi-year effort. We discuss the first year's effort, which consisted of the initial design and development of the MTRS as well as the integration of the MTRS into a larger system for initial testing.

3.1 Year 1: Design and Development

The goal for the initial development for the MTRS was to create a rules-based software that would allow pairing of various potential triggers with autonomous platform behaviors. Integrating this technology into the platform—one of the goals for the NGCV, a U.S. Army modernization priority—was anticipated to improve vehicle and crew performance, awareness, and decision making, amongst crewed and uncrewed teams (Cox et al. 2022). The potential triggers included environmental states, vehicle states, and crew states. The autonomous behaviors consisted of crew re-tasking, generating situation awareness (SA) cues, allocating computer vision (CV) capabilities to different sensors, and CV alerts. The deliverable for the initial development of the MTRS was to integrate the MTRS into the INFORMS laboratory and demonstrate the function of the MTRS's paired triggers and responses. To meet the desired goal and produce the required deliverable, a back end had to be developed to identify the specified triggers and actions in INFORMS, send rules to the system in a way it can process, and activate the rules when the necessary conditions are met. Furthermore, a front-end user interface (UI) had to be developed so that users can create rules without having to code them directly into the system. Once these preconditions were met, the MTRS could be integrated into the INFORMS ecosystem and tested as part of a larger experimental effort.

3.2 Year 1: Back-End Design and Development

The back-end system enabled users to specify **triggers** as grounded first-order logical sentences and **responses** as simple strings. The design supported declarative rule specification while ensuring efficient execution against streaming data from a dynamic environment.

Triggers were expressed as conjunctions of atomic relations, often involving interval constraints. For example, the relation *MovementPositionLat(V1, I(40, 40.1))* specified that vehicle *V1* was within a latitude range between 40 and 40.1. More complex triggers combined such conditions, for example,

MovementPositionLat(MCV1, I(40, 40.1)) and MovementPositionLon(MCV1, I(90.55, 90.61))

and not VehicleTerrainHostile(MCV1)

This formula indicated that *MCV1* was located within given latitude and longitude intervals while not in hostile terrain.

Triggers could be entered with arbitrary logical complexity, but they were internally transformed into **disjunctive normal form**. This normalization is always possible in propositional logic and sufficient for system behavior, since a rule of the form

(T1 or T2) → R

is equivalent to the pair

T1 → R

T2 → R

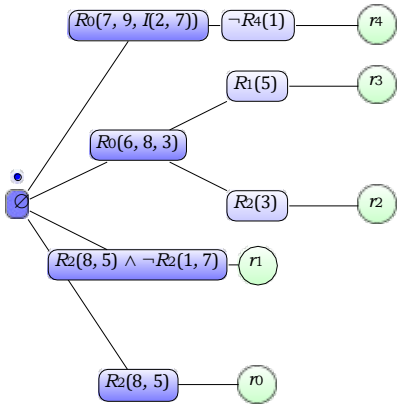
As a result, all triggers could be assumed to be conjunctions of literals, allowing uniform treatment and enabling efficient matching.

Environmental information was ingested via Kafka streams and aggregated into partial environmental models, which stored only the facts currently available about the environment. These models were inherently partial; for example, velocity updates might arrive frequently whereas terrain information was only sporadically available. This partiality was explicitly supported by the logical formalism and the matching algorithm. Internally, such partial models were represented as Python dictionaries. The keys of these were the various relations and the values were lists of tuples, each tuple denoting a set of related objects. Since dictionaries are based on hash-tables, this enabled efficient trigger matching as described below.

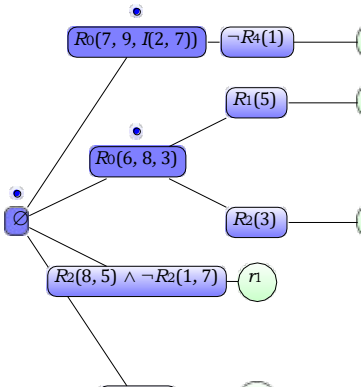
The back end introduced a distinctive **match tree algorithm** for efficient trigger evaluation. Instead of comparing each environmental model against the entire rule set, triggers were precompiled into a tree structure where each root-to-leaf path represented a trigger–response pair. Internal nodes corresponded to clusters of related conditions.

At run time, facts from the environment “pebbled” nodes in the tree, activating partial matches until full matches were discovered. This allowed responses to be determined incrementally and efficiently. For many common cases, the algorithm executed in $\Theta(|E|)$ time, scaling only with the size of the environment rather than

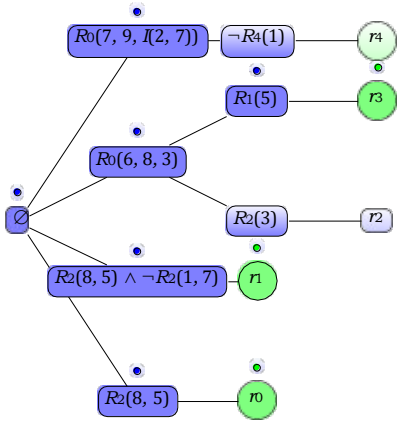
the size of the rule set.* Note that with smaller models and large rule sets, this is significantly better than a naive approach of iterating through the entire rule set. Even in the worst case, which involved interval conditions or negations, the runtime was bounded by O(|E||ρ|), where |E| is the environmental model and |ρ| is the rule set.† This algorithm is illustrated in Figure 1.



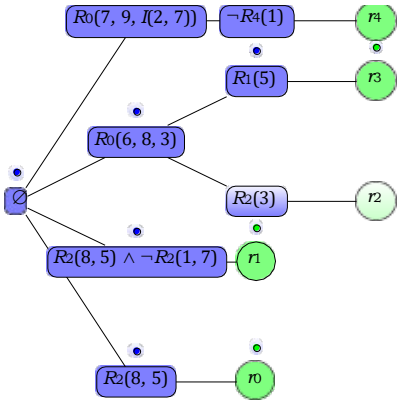
Initialization. The match tree representing the rule set is created; only the root node is active.



Partial Match. After processing R_0 from the environment, new nodes are activated. In the next iteration, R_2 will be processed to extend partial matches.



Response Match. On processing R_2, $R_2(3), R_2(8,5)$ hold while $R_2(1,7)$ does not. This activates new clusters that are the final clusters of triggers, resulting in r_1 and r_2 being added to the match set.



Final Matches. Finally, r_4 is added to the match set, which is finalized as $\{r_1, r_2, r_4\}$.

Figure 1. Response handling process.

In its initial implementation, responses were treated as raw strings, with a secondary process parsing and executing the specified action. This separation enabled the core

* The algorithm runs for a number of steps that is proportional to some logarithmic power of the number of edges multiplied by the number of edges.
† In the worst case, the algorithm runs for a number of steps that are proportional to |E||ρ|.

back end to remain optimized for matching while supporting independent evolution of the response-handling subsystem.

3.3 Year 1: Front-End Design and Development

The MTRS's back end was structured around customized responses to battlefield events, so this approach lent itself well to a TAP model (Ur et al. 2014; Rexwinkle et al. 2024). TAP approaches to EUD have users specify the behavior or state of a system as an event (i.e., *if* *trigger*) and a corresponding action to be taken (i.e., *then* *action*) when that event or state occurs (Ur et al. 2014; Reisinger et al. 2017). Given the popularity of the TAP approach to EUD, we decided to use the existing TAP interfaces as models for the MTRS UI. Previously, TAP has been used in EUD contexts such as the following: Smart Home management (Reisinger et al. 2017); computer game designs (Resnick et al. 2009); and social media management (Ur et al. 2014). We examined more language-oriented approaches, such as a "magnetic poetry" interface (a natural language expression approach with a constrained vocabulary) (Truong et al. 2004) or form-filling (a predefined structure with explicit logic markers, where specific triggers and actions can be edited using drop-down or drag-and-drop menus (Reisinger et al. 2017), eventually settling on a form-filling approach. Our design concepts combined a grid–canvas, a segmented canvas split into two sides with *ifs* on the left and *thens* on the right, and a form-filling approach consisting of a series of branching drop-down menus in a predefined structure (Reisinger et al. 2020) (see Figures 2 and 3 for details).

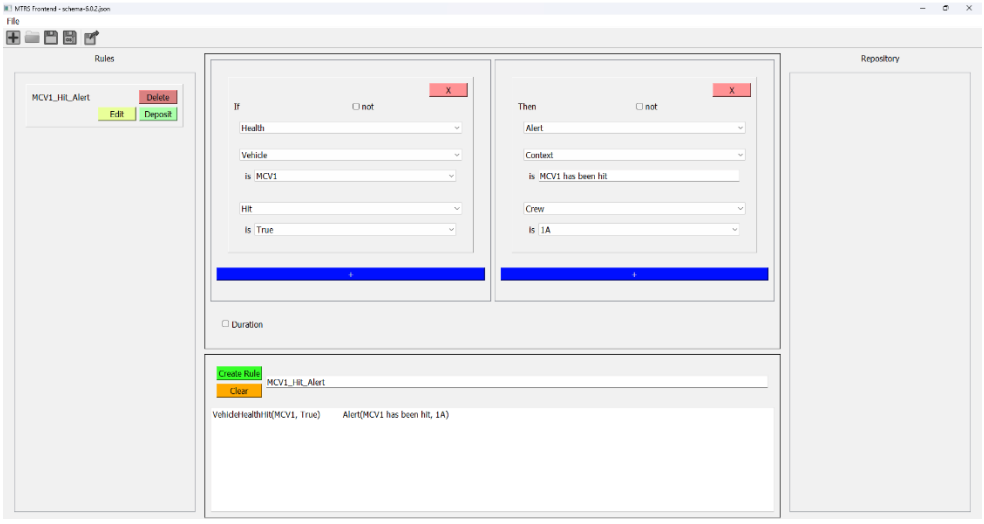


Figure 2. Initial design of MTRS. The leftmost column includes a set of completed rules. The rightmost column includes repository of rule sets. The center column includes the grid-canvas with triggers (entitled *If*) on the left side and responses (entitled *then*) on the right. A detailed discussion of this section is shown in Figure 3. At the bottom, the code being sent to the back end is displayed. In this area, one can name the rule and click the "Create Rule" button to create the rule, sending the completed rule to the "Rule" column.

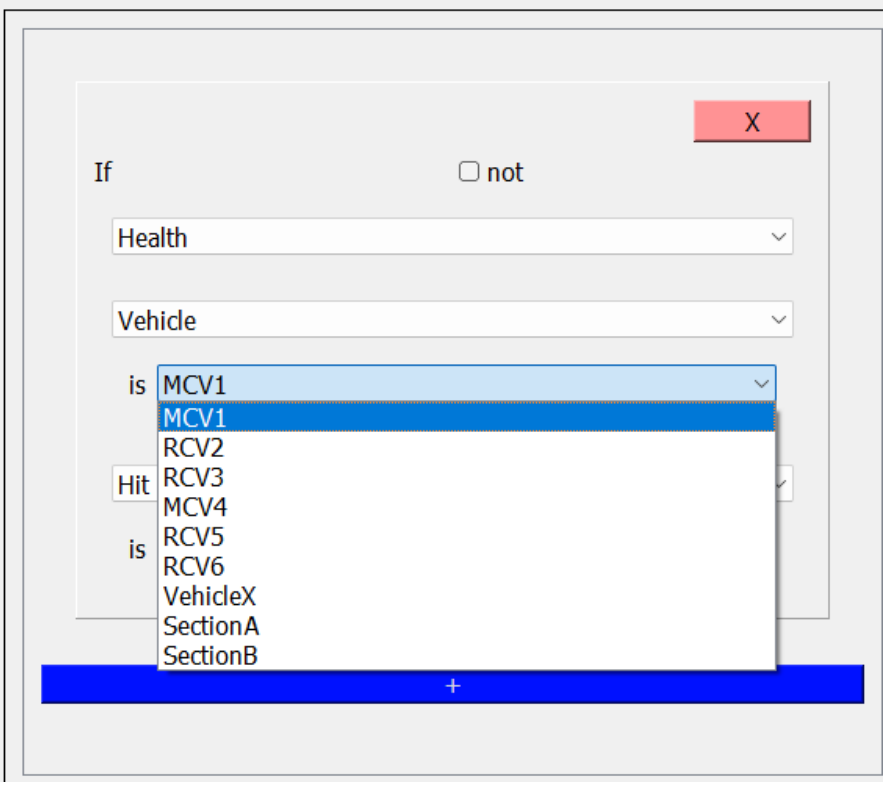


Figure 3. Detailed view of trigger canvas. The "not" checkbox allows the reverse of a rule (e.g., if vehicle is *not* hit). Each of the drop-down menus has context-specific choices, initially set up in the schema. The Blue button with the "+" can add a new trigger, linking it with the current one using a Boolean operator (AND, OR, and XOR). The red "x" button can remove a trigger. These capabilities are mirrored for the response canvas.

Once the UI was designed, it was developed into an interactive prototype, then integrated into the larger experimental set-up in the INFORMS laboratory. The MTRS was used as part of the larger INFORMS experiment as one of the technologies that was being tested. The use of the MTRS in the INFORMS experiment and response to the technology is detailed below.

3.4 Year 1: FY23 Study

The MTRS was initially conceptualized, developed, and subsequently implemented for the primary purpose of supporting the test and evaluation of the integrated adaptive AiTR system in the FY23 study. This evaluation was conducted within a simulated experiment for the NGCV program. The fundamental objective of the study was to assess the efficacy of the AiTR system, which is designed to enable military personnel—specifically those without specialized backgrounds in AI, ML, or computer science—to effectively train a CV-based AiTR model. One approach to effect the desired outcome was to create an adaptive AiTR retraining pipeline that let Soldiers "teach" the AiTR to recognize novel targets during short after-

action reviews (AARs). This training allows the system to identify and classify previously unseen targets through brief, intermittent training sessions conducted between operational missions. Another approach to affect the desired outcome was to use the MTRS to customize the deployment of autonomous resources (e.g., allocating CV capabilities to specific cameras) in response to changing mission factors. Details about the implementation of the AiTR and the MTRS in the specific software ecosystem used in the INFORMS laboratory are described in Rexwinkle et al. (2024).

3.5 Year 1: FY23 Study Results

Discussion of the results of the FY23 study, where the MTRS was used during a SIMEX, can be found in Rexwinkle et al. (2024).

3.6 Year 1: FY23 Soldier Feedback

In addition to the behavioral data about the MTRS acquired from the FY23 study, we solicited feedback from the Soldier participants and the experimenters who were interacting with the MTRS. In the study, experimenters acted as intermediaries for Soldiers, processing their needs, using that information to create rules, and discussing with Soldiers how they turned feedback into rules. In general, Soldiers expressed that they had difficulty fully understanding the rules that were implemented, although they responded positively toward rules made to directly deliver SA cues to them, saving them the effort of having to hunt down the information manually.

Although both the front-end interface and back-end software were functional, testing revealed that the interface for creating and editing rules had shortcomings in readability as well as interaction and workflow efficiency. Experimenters had difficulty efficiently editing rules distributed vertically in the grid canvas. This problem worsened under time pressure. Experimenters had to simultaneously translate the rules to Soldiers and edit them in real time. This led to situations in which the swamped experimenters resorted to directly editing code, rather than modifying rules using the UI, to save time. This feedback suggested that the front-end interface should be reworked to increase readability and that the vertical distribution of rules should be condensed, especially when more complex, multi-trigger, multi-action rules are involved.

4. Redesign, Development, and Testing of the MTRS: Year 2

The MTRS featured more prominently in FY24, with the set of possible actions being expanded to include more complex, automated behaviors. We discuss the second year's effort, which consisted of an expansion of the MTRS's capabilities with the back end, a redesign of the front end to improve ease of use, and a more direct examination of the technology in the INFORMS laboratory.

4.1 Year 2: Back-End Design and Development

In the second year, the matching algorithm for the back end was substantially improved and several new simulation-specific features were added to the back end. In particular, the matching-tree data structure was streamlined, and recursive grammar was developed to allow for more complex responses. The latter were re-envisioned as a kind of behavior tree, where a complex response was imagined as a list of responses, each to be executed in sequence or parallel. Each of these responses was conceived of as a complex response, so that the execution was given a recursive tree-like structure.

4.2 Year 2: Front-End Design and Development

Although the initial iteration of the MTRS's UI was functional, use in the FY23 study suggested that it had shortcomings in readability as well as interaction and workflow efficiency. Given these findings from the previous study and user feedback on the interface, we sought to redesign the interface to address the aforementioned shortcomings (see the restructured interface in Figure 4).

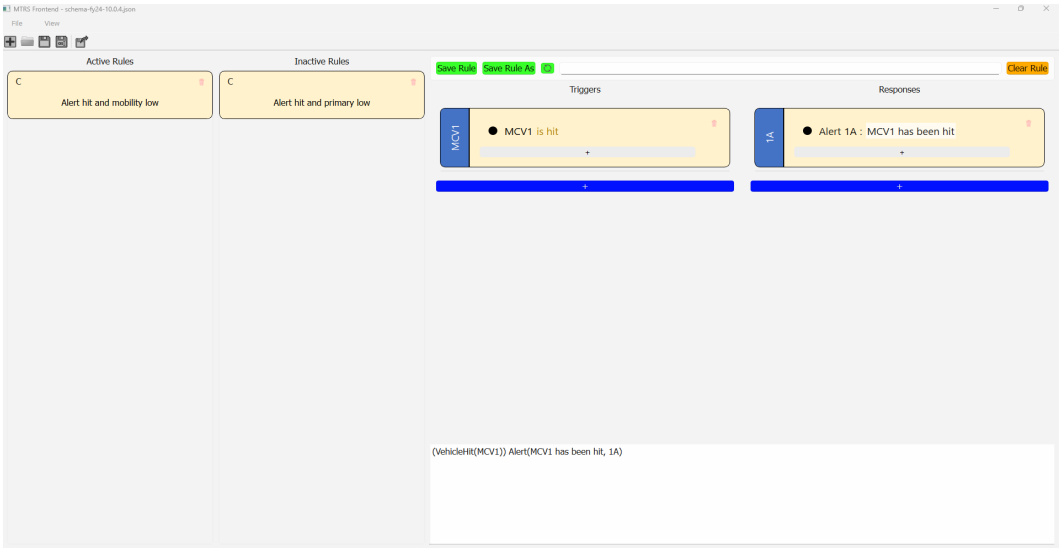


Figure 4. A screenshot of the updated MTRS interface. This updated interface includes active and inactive rule columns and object-based modules for both triggers and responses.

The repository column was removed as we shifted from a centralized repository approach to individualized repository files that can be saved and loaded. To further facilitate rule management, the rules column was expanded to include both an active and inactive rule column. For rules in the active rule column, the MTRS monitors incoming data and activates relevant response when each trigger is detected. Rules in the inactive rules column are ignored. One can drag a rule from the active rules column to the inactive rules column and vice versa. In addition, the rule-naming functionality has been moved from the bottom of the window to the top.

To reduce the overwhelming verticality of each rule and to improve readability, we restructured the presentation of the rules to focus on the object of the trigger or action, rather than using the hierarchical organizational approach used the previous year. This object-centered approach is more modular than the previous one, with multiple labels identifying the object around which the trigger or action is centered. Furthermore, a rule with multiple triggers centered around the same object is grouped together; this grouping in the interface facilitates the development of good mental models (Endsley 2017), allowing for better understanding of individual rules as they are composed and read. This approach also allows for easier user review of individual rules, so that users could more easily spot errors, contradictions, or redundancies (see Figure 5 for details).

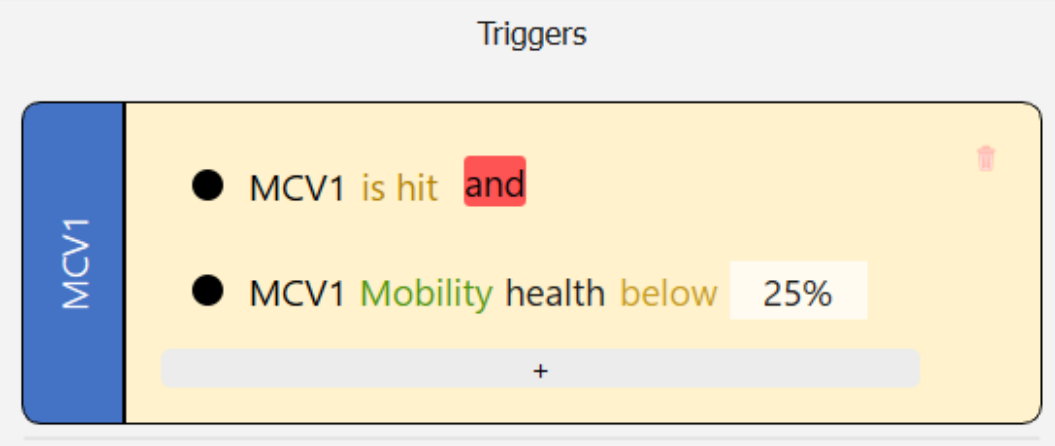


Figure 5. A module in the second iteration of the MTRS interface, depicting a TAP rule that triggers when the first manned control vehicle (MCV1) is hit and MCV1's Mobility health is below 25%. The module is labeled by the object of the trigger (MCV1).

In addition, the rule creation process was broken up into several distinct parts. When creating a rule, users click the blue plus button, which activates a pop-up menu, detailing the different categories of subjects that are available; users can select the desired subject within this series of nested menus (see Figure 6). After an object is selected, an accordion menu is displayed, detailing categories of triggers

or responses. After users select a category, the menu expands to show the available triggers or responses under that category (established using the schema) (see Figure 7). When a trigger or response from the accordion menu selecting, then populates the grid canvas with a module containing an editable template.

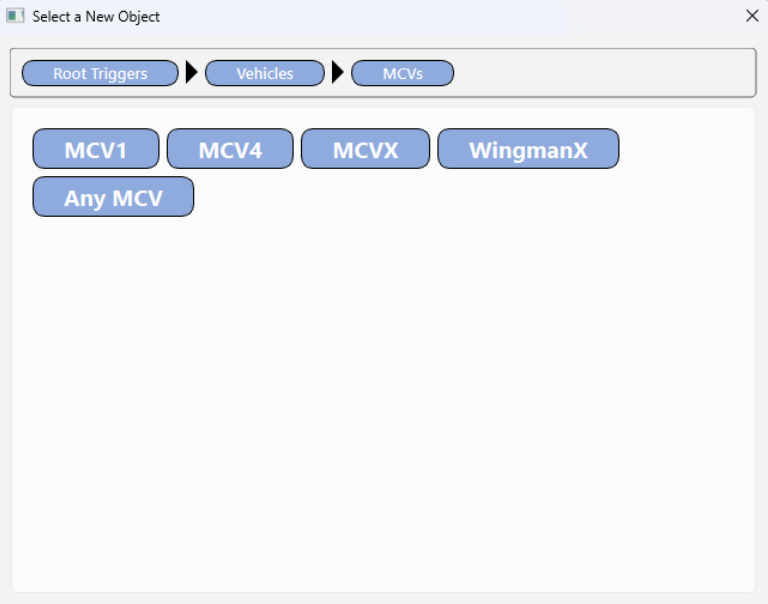


Figure 6. Pop-up menu detailing the objects for a trigger. The current screen shows the possible MCVs available. A breadcrumb menu above allows users to return to a previous pop-up menu.

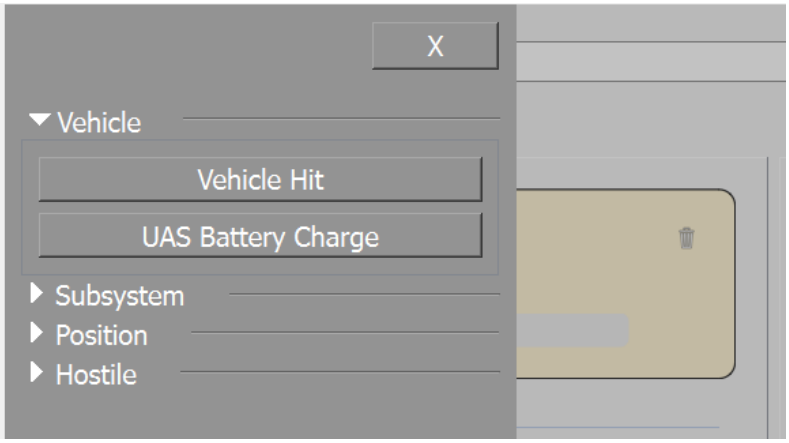


Figure 7. Accordion menu describing categories of triggers available for MCV1. Under the "Vehicle" category, the "Vehicle Hit" and "UAS Battery Charge" triggers are visible.

For the construction of the individual rules, we focused on a more natural-language approach—taking inspiration from the Truong et al. (2004) magnetic poetry approach. However, rather than using the more flexible structure of their Capture and Access Magnetic Poetry interface, we kept the more rigid structure of the form-filling approach (Reisinger et al. 2020), yielding an interface that more closely

resembles Mad Libs. This Mad Libs approach uses sentence templates for each trigger or action, with keywords that can be adjusted using toggles and drop-down menus. As seen in Figure 8, the more sentence-based structure of the interface allows users to toggle verbs to indicate confirmatory or contradictory (e.g., "is hit" vs. "is not hit"), unlike the previous version in which users marked a checkbox to indicate that the rule was contradictory. When relational operators are used (or other operators with more than two options), then this version of the interface uses a drop-down menu, which provides options from which users can select (see Figure 9).

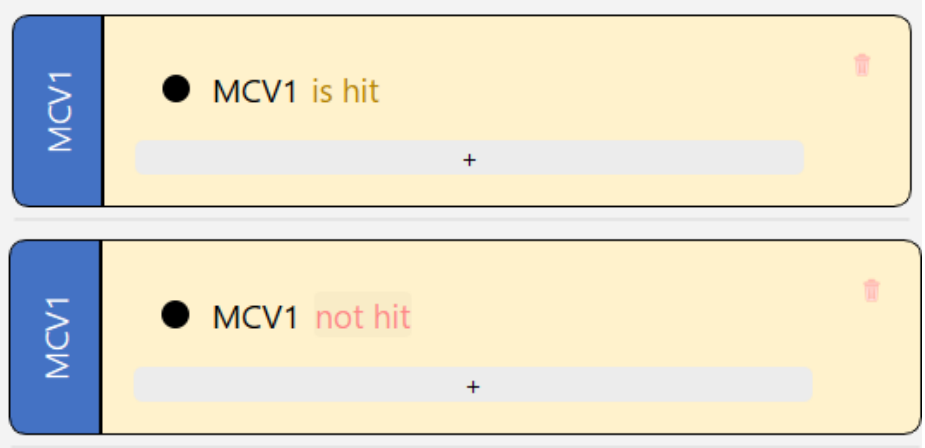


Figure 8. A comparison of trigger modules, illustrating the use of toggleable language. The sentence template defaults to providing confirmatory language. When applicable, users can toggle to a negative.

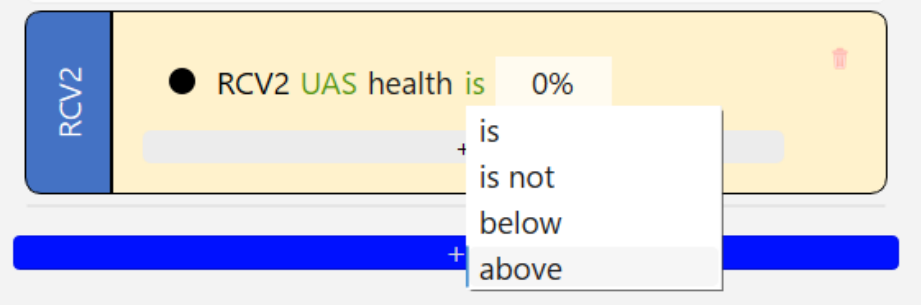


Figure 9. An example of the drop-down menu in the second MTRS interface iteration. The image depicts relational operators used to indicate the health of an unmanned aerial system (UAS) associated with an unmanned RCV (RCV2).

4.3 Year 2: FY24 Study

In the subsequent FY24 study, the MTRS underwent significant expansion and refinement. These enhancements were applied to both its front-end and back-end components—that is, the UI and the underlying algorithmic architecture, respectively—to introduce a robust dynamic resource allocation capability. The primary objective of the FY24 evaluation was to test and validate this enhanced

MTRS as the core technology enabling Soldiers to customize the dynamic tasking of crew members and the deployment of autonomous resources. This customization was designed to be responsive to a variety of inputs, including changing mission factors, platoon-specific standard operating procedures, and individual operator preferences.

Consistent with the methodology of the preceding FY23 study, the FY24 Soldiers experimentation involved 2 groups, each comprising 14 Soldiers who performed ground maneuver combat operations within the NGCV-simulated environment. The central goal of this study was to quantitatively and qualitatively evaluate the operational effect of the re-tasking technology integrated into the MTRS. The initial re-tasking behaviors of the MTRS were configured based on data and feedback collected during extensive consultations with military subject matter experts and through rigorous pilot testing of the fully integrated system.

During the formal SIMEX, Soldiers interacted with the MTRS technology throughout the execution of their missions and were provided opportunities to adapt the system's automated behaviors during AARs. In the execution phase, Soldiers conducted unscripted, dynamic combat operations against a human-controlled opposition force (OPFOR). The mission scenarios and OPFOR actions were specifically designed to introduce events that would necessitate resource re-allocation. In response to these trigger events, the MTRS would dynamically execute the re-allocation, either autonomously or after a crew member's acceptance of a cued suggestion presented within their UI.

To rigorously assess the system's impact, three distinct technology capability conditions were tested. This design allowed for the isolated evaluation of both the effects of team reconfiguration and the specific contributions of the MTRS.

- **Baseline Condition:** This condition represented the control group with minimal technological augmentation. Soldiers had no capability to be re-tasked and were restricted to a limited suite of predefined autonomy behaviors.
- **Control Condition:** This condition provided full manual control. It granted Soldiers the full capability to manually re-task themselves and other crew members, along with access to the complete suite of available autonomy behaviors but without MTRS assistance.
- **Experimental Condition:** This condition introduced the MTRS. It included all capabilities of the control condition (manual re-tasking and full autonomy suite) and added the MTRS to support and automate the re-tasking process and the engagement of autonomous systems.

The overarching research objective was to determine whether the MTRS could facilitate a more flexible and efficient allocation of critical resources to meet evolving mission demands. Furthermore, the study aimed to confirm that the system could be easily adapted by its end-users, with the ultimate goal of increasing the mission effectiveness of the HMI platoon, particularly in operational contexts characterized by reduced Soldier-to-platform ratios.

4.4 Year 2: FY24 Study Results

As reported in the technical note (Gremillion et al. 2024) summarizing the FY24 study findings—response time, operational performance measures, and subjective measures—the MTRS technology had a significant impact on Soldiers' performance.

In the qualitative analysis conducted during AARs, Soldiers from both Group 1 and Group 2 emphasized the critical role of the MTRS technology in effectively managing and preserving their unmanned aerial vehicle (UAV) assets. Specifically, Soldiers noted that the MTRS was instrumental in mitigating UAV attrition caused by telemetry link failures or battery depletion.

The importance of the MTRS was further corroborated through post-mission, self-report questionnaires. On average, Soldiers provided favorable ratings for the system's cueing capabilities, particularly in terms of its timeliness, predictability, and overall helpfulness in operational contexts. These findings underscore the operational utility of the MTRS in enhancing UAV survivability and mission effectiveness.

4.5 Year 2: FY24 Soldier Feedback

One of the goals of the MTRS was to allow for human crew members to customize their vehicle's responses to events that occurred on the battlefield. This capability was meant to alleviate surprises and other negative performance outcomes stemming from inappropriate allocation of cognitive resources when moving from low-demand tasks to high-demand situations and vice versa (Johnson and Marathe 2025). After the Soldiers completed the experiment, we solicited feedback on the MTRS's capabilities and interface. They provided information concerning what they liked about the tool, what shortcomings they found, and potential design and capabilities they would like to see in the future.

Soldiers found MTRS alerts to be a useful function. They reported that vehicle health status being cued up under specific conditions helped them keep track of their teammates' status when the situation was hectic and communication was

harried. Improved crew coordination is expected to help counteract detrimental effects of workload transition (Wickens and Huey 1993, p. 4). Furthermore, Soldiers found that the MTRS was critical to managing and preserving UAS assets as well. The UAS had limited battery life and had a maximum range that could not be exceeded without losing control of it. MTRS rules could be used to alert users of low battery or link loss, allowing them to keep these assets in play longer than they would have otherwise. Although these alerts were found to be useful, rules where the UAS automatically redocked on its relevant vehicle to recharge were avoided.

Although task switching, in and of itself, was considered a valuable capability, the value of using the MTRS to initiate task switching was more uncertain. Soldiers did not report difficulty with that functionality, but they reported a preference for manual re-tasking, finding it to be faster in some cases than navigating to where they were cued and accepting the task change.

Regarding future functionality, Soldiers asked for greater customizability. Regarding triggers, Soldiers requested the ability to tie rules to buttons on the controller they used to control the vehicles, so that they could, in essence, use the MTRS to create custom keyboard shortcuts. Regarding responses, Soldiers requested greater customizability for alerts, desiring the ability to change an alert's color, font size, and duration of display. They also suggested the addition of configurable auditory tones that users could add to alerts as desired.

Regarding the desired functionality for future iterations of the MTRS, Soldiers also sought capabilities that improve understanding or organization of rules. One request was some method to generate big picture summaries of currently configured rules. In addition, they asked for ways to reorder items within a rule configuration. Another desired feature was a way to filter rules by the type of trigger, action, object, individual, or role. Thus, the desired outcome for future iterations of the MTRS are ways of facilitating understanding and increasing ease of use, using this feedback as a guide.

5. Current Work

The goal of the MTRS is to facilitate Soldier-guided adaptation of a system using a TAP-based approach. Based on the results of the previous study and the requested feedback, we sought to expand the functionality of the MTRS and improve its ease of use, and we have been doing so through the implementation of audio input and large language model (LLM)–based rule summaries.

5.1 Audio Interface for MTRS

As previously stated, one of the major goals, when iterating on the MTRS interface, is to improve ease of use for end-users. From FY23 to FY24, we implemented the Mad Libs–style interface, which is an approach that was more natural language based than its previous iteration. We sought to extend this approach further by implementing an audio-based interface, where users can create and edit TAP rules using spoken language. This approach is an addition to the existing visual interface (see system figure for audio interface in Figure 10).

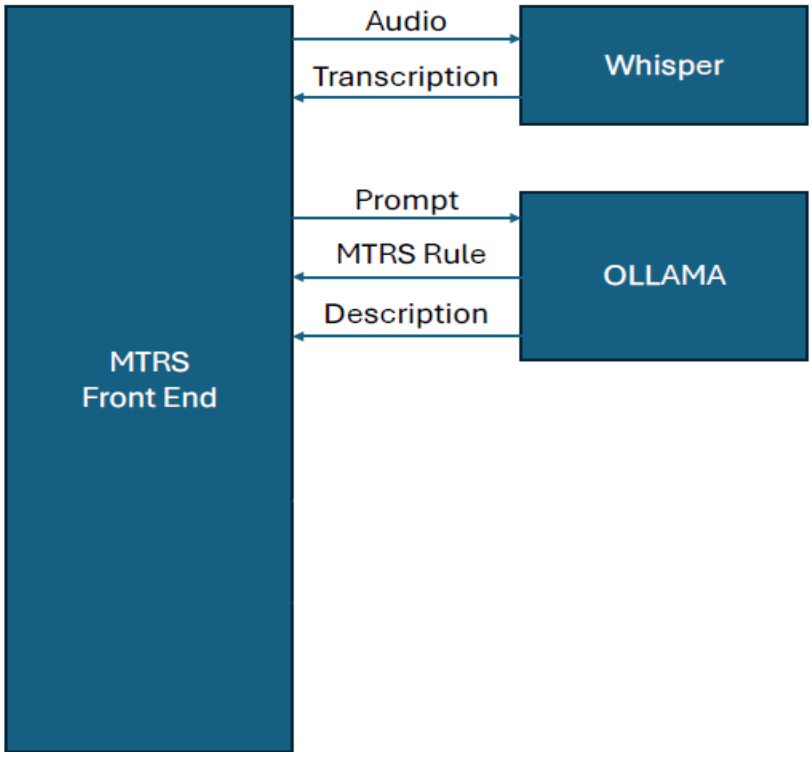


Figure 10. System figure for the audio interface, detailing the relationship among the front-end interface, the transcription element, and the LLM element.

This updated interface contains multiple pieces to accomplish the goal of creating a rule using the user's voice. First, it uses an automatic speech recognition (ASR) system—Whisper—to transcribe users' voice data. The second piece is a local instantiation of an LLM—specifically MetaAI's LLaMA LLM— hosted within an open source wrapper for running LLMs called Ollama. The transcription of the user's request from Whisper is sent to the LLM with a prompt describing the MTRS language and instructions for translation of natural language text into the formal MTRS language. If the verbal command or transcription is ambiguous, the system will either match the command to existing rule language as best it can (and display that rule using the visual interface developed in FY24), or it will return blanks

around the rule structure it can parse. Any blanks or errors can be fixed by editing, using the existing visual interface, or by restating (or iterating upon) the previous verbal command. The interface is shown in Figure 11.

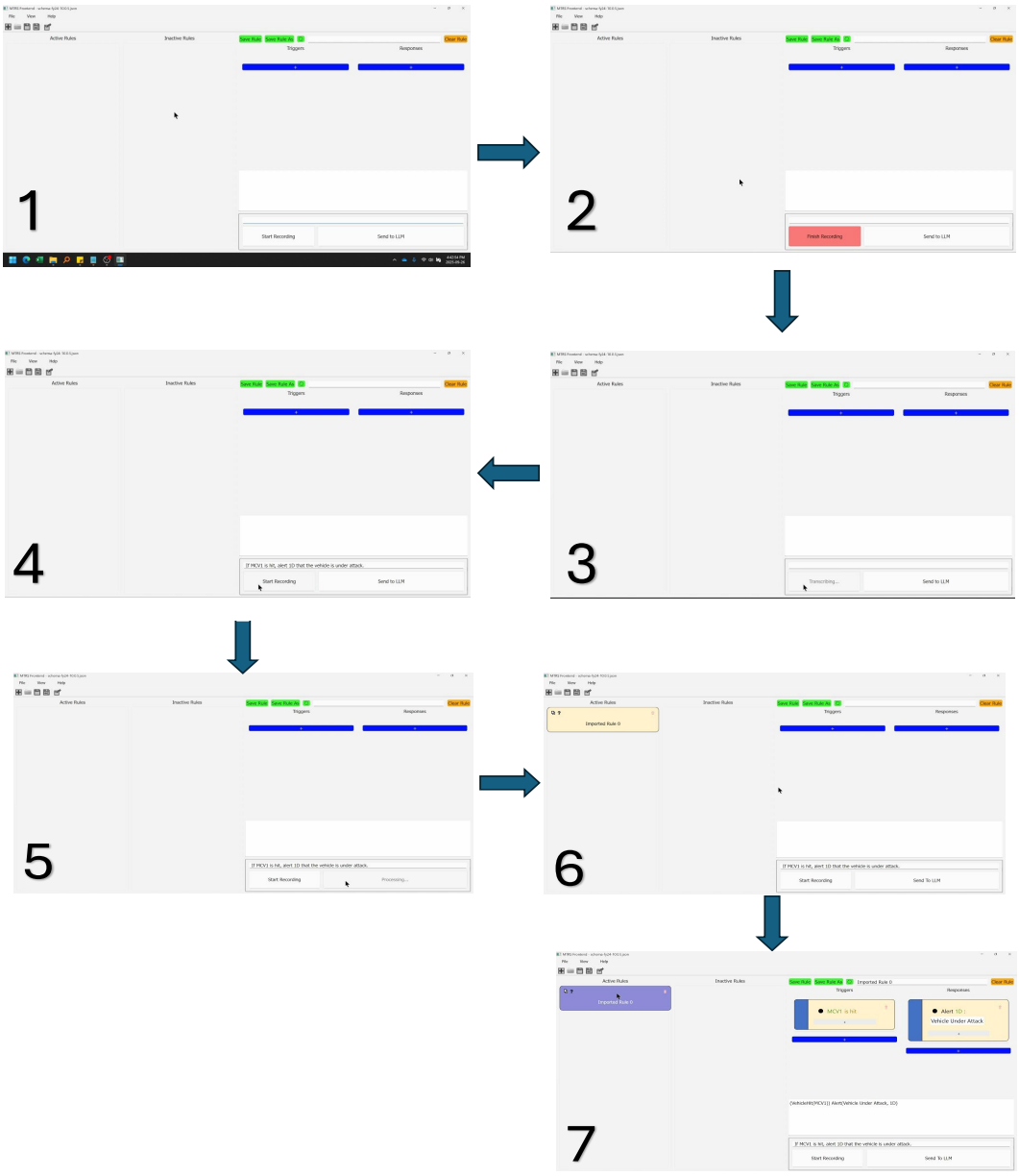


Figure 11. Interface for using the audio interface of the MTRS. 1) The interface before the user interacts with it. 2) The interface while one vocally conveys the rule they want. 3) Whisper processing the audio input for transcription. 4) The transcribed text produced by Whisper. 5) The transcribed text being sent to Ollama to be converted into a rule. 6) The new rule added to compendium of active rules. 7) The newly created rule selected, with the specific layout available for review.

5.2 Rule Summarizer

After the FY24 study, Soldiers specifically requested capabilities that improved understanding of the rules. As part of the effort to integrate LLM technology into the MTRS, we sought to pursue this goal by using an LLM-based Rule Summarizer. Using LLMs to automatically summarize texts outperforms existing state-of-the-art automatic summary techniques, and this promises to be an effective use case for LLMs (Zhang et al. 2024). The Rule Summarizer uses the aforementioned LLM to generate a summary of the selected rule. The user selects a rule and after they press a button to start the process, the MTRS sends the rule's code to the LLM. This code is embedded in a compound prompt, hidden from the user, that defines the relevant terms and shapes the outcome, so users receive a summary of what the rule does in plain language. This tool would make it easier to parse and review individual rules. As such, users creating new rules could confirm the rule outputs match their expectations. In addition, users could use this tool to review existing rules, which could be useful to remind the users of rules they previously made or to onboard new users to a set of rules they had not yet encountered. However, note that LLMs can hallucinate, that is, they can provide inaccurate information (Gao et al. 2024). As such, we are assessing the feasibility of its use in the field and the appropriate contexts in which to implement this tool.

5.3 Homework Checker

In previous MTRS iterations, the back ends were established to both facilitate the capabilities of the MTRS and to meet the needs of that year's INFORMS experiment. In the absence of an INFORMS experiment, the next iteration of the MTRS back end not only facilitates the capabilities of the MTRS, but it also involved the creation of a "homework checker." This back-end development allows for the assessment of speed and accuracy in rule creation by, essentially, acting as a unit test system, comparing the output of the input rule set with a prespecified desired output. This approach is expected to be useful for assessing user behavior under fixed, experimental contexts.

One can set up prespecified outcomes for fixed circumstances, but using the homework checker for more dynamic circumstances may be less useful. If one wanted to use this back-end module as an automated redundancy or contradiction checker, developers and scenario designers would need to know the desired outcome of each rule set and develop test sets to which human-constructed rule sets could be compared. As such, given the current iteration of the MTRS back end, human supervision over the management of rules and rule sets is paramount.

6. Unpacking the Audio Interface

To improve usability, the system incorporated an audio-based rule creation interface. This interface allowed users to speak both the trigger and the corresponding response for a rule, eliminating the need to type logical expressions manually under time-sensitive or high-stress conditions.

Incoming speech was processed in real time using OpenAI's (2022) Whisper ASR model. For efficiency and reliability, an English-only variant of Whisper was deployed. Broader multilingual models were found to occasionally misinterpret certain English utterances as foreign-language speech, introducing unnecessary errors. Restricting the model to English improved transcription accuracy and robustness in operational contexts.

The ASR model was deployed as a Representational State Transfer (REST) service running in the background. Instead of loading the model on each request, which would incur substantial latency, the service kept the model resident in memory. Calls were then made through a lightweight web-based interface, ensuring near-instantaneous response times and reducing computational overhead. This service-oriented architecture allowed multiple clients to interact with the model concurrently while maintaining consistent performance.

In addition to audio input, the interface supported multimodal extensions, enabling Soldiers to supplement spoken rules with structured visual or textual elements. This multimodal capability provided flexibility: audio commands could capture natural user intent, whereas structured elements ensured precision in defining triggers and responses.

6.1 Using LLMs for an Audio Interface

Although the MTRS language specification is precise and unambiguous, a priori specifying rules with requisite precision is a specialized skill that not all Soldiers can be expected to possess. To aid in crafting MTRS rules, we developed an LLM interface to the system that takes the natural language rule descriptions as input and then outputs formal MTRS rules.

The interface translates free-form English descriptions of triggers and responses into formally valid MTRS rules. It runs a locally hosted or remotely served LLaMA-family model guided by a structured "trigger prompt," and then it validates the model's output with a strict grammar implemented in PLY (Python Lex/Yacc). Invalid outputs are automatically re-prompted until they conform to the grammar, yielding rules that the back end can compile and execute.

The code supports two execution modes: “remote server” and “local mode.” In the remote server mode, requests are forwarded to a hosted service that maintains a warm instance of the model; that is, the model is loaded once and maintained as a service. This avoids repeatedly incurring initialization costs associated with loading the model and adapts easily to a robot operating system-based environment as is often used in robotics. This also allows much quicker responses and enables multiple clients to share a single model instance efficiently. In the local mode, the system builds a Transformers pipeline and optionally loads a quantized large model. The default model is Meta’s LLaMA 3.1 7B Instruct, although variants including 3B and 70B models are supported, the latter with 4-bit quantization to reduce memory demands.

The MTRS trigger language is defined by a Backus–Naur Form grammar. This grammar specifies valid rule structures and ensures consistency across the system. A lexer and parser implemented with PLY enforce this grammar. Each generated rule is passed through the parser, which automatically checks for syntax errors. Malformed rules are rejected and regenerated until they parse correctly. This process guarantees that only syntactically valid triggers and responses are admitted. Although there is no explicit process to ensure the initial intent is maintained, initial testing did not reveal much rule mutation.

The trigger prompt is a carefully constructed specification provided to the language model to constrain its outputs.

1) **Header and instruction.** The prompt begins with an explicit statement of the task: translating natural language descriptions of events into the formal trigger language.
2) **Language description and predicate inventory.** The prompt enumerates all available predicates, constants, and intervals and defines interval semantics (e.g., I(a, b) means $a \leq \times < b$). This inventory, drawn from mission-relevant constants such as vehicle identifiers, subsystems, and status flags, constrains the model’s output and reduces hallucinations.
3) **Examples and formatting.** The prompt provides examples of natural language and their corresponding translations. It requires that the model’s output be enclosed in square brackets [. . .]. This enables the runtime system to reliably extract only the formal rule.
4) **Contextual grounding.** The prompt situates the rules in operational contexts, such as robotic versus manual vehicles, so that the model selects constants appropriately.

This structured design turns the language model into a deterministic translator over a closed vocabulary, improving reliability and reducing variation.

Like the trigger, the response is also described to the model in detail. Responses are defined in a parallel formal language. The language supports two connectives: and (parallel execution) and then (sequential execution), with clearly defined precedence. The response prompt enumerates all permissible actions and provides examples, ensuring clarity in sequencing and parallelization.

6.2. The Translation Pipeline and Best Practices When Using LLMs for an Audio Interface

Putting together all these mechanics, we can describe the process by which the audio interface uses an LLM to convert audio input to formal language. This "Translation Pipeline" is described below:

1) Natural language input is translated into a candidate trigger or response using the long system prompt.
2) The candidate rule is extracted from square brackets.
3) The candidate is passed to the PLY parser for syntax validation.
4) If errors occur, the system issues a corrective prompt indicating the syntax error and requests a new translation. This loop continues until a valid rule is produced or a retry limit is reached.
5) For convenience, "if . . . then . . . " rules in English can be parsed directly, with the trigger and response extracted and translated separately.

To effectively translate an audio input to an appropriate rule, the prompt formed in the translation pipeline must be able to be parsed by the language model. Through multiple iterations of this process, we have developed several techniques to facilitate this pipeline, which we recommend as best practices for others who adapt this software for use with their own systems. We list some specific features that improve robustness of the system:

- **Bracketed extraction.** All rules are required to appear inside square brackets, enabling reliable string extraction and minimizing stray text.
- **Closed-world vocabulary.** Enumerated constants constrain possible outputs, reducing invalid tokens.
- **Strict parsing gate.** PLY ensures that only grammar-compliant rules enter the back end.
- **Error-correction loop.** The system automatically regenerates malformed outputs until they pass syntax checks.
- **Response precedence.** Clear rules for *and* and *then* avoid ambiguity in execution order.

The system also supports converting formal rules back into natural language. A dedicated prompt and examples guide the model to produce fluent, human-readable descriptions of formal rules. This feature improves usability by allowing Soldiers to confirm that the generated rules reflect their intent.

The following key points are worth highlighting:

- The combination of specification-style prompting and a strictly enumerated symbol set reduces creative drift and makes the LLM behave more like a compiler front end.
- PLY validation transforms probabilistic model outputs into deterministic, syntax-guaranteed rules.
- The REST-style deployment keeps the model resident in memory, eliminating repeated load-up times and supporting efficient concurrent use.
- The design provides a balance between the flexibility of natural language input and the precision required for formal logic.

We include a transcript from testing the pipeline (audio-based recording to MTRS triggers) for several rules in Appendix A. The transcript shows that the pipeline often produces correct responses and that the syntax correction can be effective.

7. Modularity in the MTRS

The MTRS front end is designed with a high degree of modularity, allowing it to be easily adapted for new use cases in new environments. The front-end software contains only the information needed to interact with the user, namely, the basic structure of MTRS rules. It does not contain information about the specific environment in which it is operating (e.g., INFORMS). Documentation explaining how the front end works and how to edit it is available in Appendix B. The information about the environment, such as specific triggers and responses and their parameters, is defined in a schema. The schema is written in JSON and can be easily edited in a text editor to adapt the front end for new triggers and responses as needed. A guide explaining the development of a schema for a specific use case is available to assist users in adapting it to their environments in Appendix C.

Likewise, to allow for flexibility, the MTRS front end interfaces with LLMs via Ollama, an open source wrapper for LLMs that provides a REST API for accessing them. This allows for 1) changing the LLM as needed to suit different use cases such as accessing remotely hosted LLMs on more powerful server hardware or 2) replacing locally hosted LLMs with more powerful LLMs as they become available without requiring software changes to the MTRS front end itself. Likewise, the instructions to the LLM for conversion of natural language to the MTRS formal

language, and vice versa, are contained in configuration files that can be easily updated as the MTRS schema evolves for different environments.

The MTRS back end, being decoupled from the front end, can be tailored to various environments and exchanged as needed. The connection between them is the completed rule-set definition output by the front end for use by the back end in operation. In the course of developing the MTRS, four back ends have been developed:

1) The original FY23 INFORMS back end that focused on allocation of AiTR resources;
2) The FY24 INFORMS back end focused on dynamic task allocation among NGCV crew members;
3) A port of the FY24 INFORMS back end from Python to C# to improve performance; and
4) A homework checker back end that was intended for use in an experiment to investigate the effect of the LLM on MTRS user speed and accuracy in rule creation.

The project overview documentation (i.e., the content of the Readme.md) is available in Appendix D.

The modularity of the MTRS means that specific needs can be met by customizing the back end. For a scientific experiment with relatively static scenarios, a homework checker can be useful to provide participants with performance feedback. For a more dynamic scenario, with more complex rule sets, alternate tools to improve user understanding of those rule sets and automate some level of rule review can be developed. To deal with redundant rules, for example, a back-end tool could be developed that could identify when overlapping conditions produce the same output or identify rules that trigger along with an initial rule but do not provide additional output. If there was an issue with contradicting rules, domain constraints can be placed on responses so conflicting responses from the same trigger can be detected. Using the supplied documentation as a guide, one can customize the back end to suit the needs of MTRS users.

8. Conclusion

The underlying reason for the development of the MTRS is to help Soldiers manage the increasing complexity of the battlefield and the tools available to them to help them navigate it. It does so by allowing Soldiers to automate behaviors, redirect attention, and facilitate technology-mediated teamwork behaviors. However, EUD is a solution for dealing with complexity in several different

contexts beyond the INFORMS laboratory (e.g., smart homes filled with smart products, spreadsheets festooned with data, and computer laboratories filled with children learning to code games) (Lieberman et al. 2006; Resnick et al. 2009; Reisinger et al. 2017), and, as such, the MTRS was developed to be a modular tool that can be adapted to different circumstances. We have discussed the development, testing, and redevelopment of the MTRS, how it is currently used, and how it can be adapted to different circumstances.

Software and necessary codes to implement the MTRS front end and back end are hosted in a GitLab repository. They can be made available upon request. Should you wish to obtain a copy or access, please contact the authors.

9. References

Chhan D, Scharine A, Perelman BS. Human-autonomy teaming essential research program project 2: transparent multimodal crew interface designs, technical note 3: multimodal cueing for transparency in mobility operations. DEVCOM Army Research Laboratory (US); 2020 May. Report No.: ARL-TN-1019.

Cox KR et al. A summary of the effects of technology aids and crew size on a next generation combat vehicle platoon section. DEVCOM Army Research Laboratory (US); 2022 Mar. Report No.: ARL-TN-1109.

Crowell HP et al. Potential indicators of team trust measured in a next generation combat vehicle simulator. DEVCOM Army Research Laboratory (US); 2023 Feb. Report No.: ARL-TR-9647.

Devlin SP, Brown NL, Drollinger S, Alami J, Riggs SL. Workload transition rate matters: evidence from growth curve modeling. Appl Ergon. 2023; 106:103885.

Endsley MR. From here to autonomy: lessons learned from human–automation research. Hum Factors. 2017;59(1):5–27.

Gao Y et al. ChatIoT: zero-code generation of trigger-action based IoT programs. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies. 2024;8(3):Article 103. https://doi.org/10.1145/3678585

Gremillion GM et al. Technologies for automated dynamic resource allocation in the next generation combat vehicle. DEVCOM Army Research Laboratory (US); 2024 Nov. Report No.: ARL-TN-1234.

Johnson CJ, Marathe AR. Cognitive workload considerations for human–machine integrated formations. DEVCOM Army Research Laboratory (US); 2025 Apr. Report No.: ARL-TN-1248.

Lieberman H, Paternò F, Klann M, Wulf V. End-user development: an emerging paradigm. End-User Development. Springer Netherlands; 2006. p. 1–8.

Moacdieh NM, Devlin SP, Jundi H, Riggs SL. Effects of workload and workload transitions on attention allocation in a dual-task environment: evidence from eye tracking metrics. J Cogn Eng Decis Mak. 2020;14(2):132–151.

OpenAI. Whisper. 2022 [accessed 2023 Mar 6]. https://openai.com/index/whisper/

Paternò F, Santoro C. End-user development for personalizing applications, things, and robots. Int J Hum-Comput Stud. 2019;131:120–130.

Perelman BS, Lakhmani S, Wright JL. Human–autonomy teaming essential research program project 2: transparent multimodal crew interface designs technical note 1: general overview. DEVCOM Army Research Laboratory (US); 2020a Jan. Report No.: ARL-TN-1003.

Perelman BS, Wright JL, Lieberman GA, Lakhmani S. Human–autonomy teaming essential research program project 2: transparent multimodal crew interface designs technical note 2: transparency in mobility planning. DEVCOM Army Research Laboratory (US); 2020b Jan. Report No.: ARL-TN-1004.

Reisinger MR, Schrammel J, Fröhlich P. Visual languages for smart spaces: end-user programming between data-flow and form-filling. Proceedings of the 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC); 2017 Oct 11–14; Raleigh, NC. IEEE; c2017. p. 165–169.

Reisinger MR, Schrammel J, Suette S, Fröhlich P. Home smart home: approachable interfaces for intelligibility, modification, and end-user programming. Interact Des Archit. 2020;45:226–245.

Resnick M et al. Scratch: programming for all. CACM. 2009;52(11):60–67.

Rexwinkle JT et al. Adaptive situation awareness technologies for next generation combat platforms. DEVCOM Army Research Laboratory (US); 2024 Jan. Report No.: ARL-TN-1187.

Stanton NA et al. Human factors methods: a practical guide for engineering and design. CRC Press; 2017.

Tao D et al. A systematic review of physiological measures of mental workload. Int J Environ Res Public Health. 2019;16(15):2716.

Truong KN, Huang EM, Abowd GD. CAMP: a magnetic poetry interface for end-user programming of capture applications for the home. Proceedings of the UbiComp 2004, 6th International Conference on Ubiquitous Computing. 2004 Sep 7–10. Nottingham, UK. Springer Berlin Heidelberg; c2004. p. 143–160.

Ur B, McManus E, Pak Yong Ho M, Littman ML. Practical trigger-action programming in the smart home. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 2014 Apr 26–May 1; Toronto, Ontario, Canada. SIGCHI; c2014. p. 803–812.

Wickens CD, Huey BM, editors. Workload transition: implications for individual and team performance. National Academies Press; 1993.Zhang T et al. Benchmarking large language models for news summarization. Trans Assoc Comput Linguistics. 2024;12:39–57.

Appendix A. Transcript from Testing the Translation Pipeline

Here, the transcript of tests of the “Translation Pipeline” for several Modular Trigger Response System (MTRS) triggers are displayed. The transcripts show the name of the audio file where a human voice is dictating a rule, the transcribed text of that audio file, followed by the system’s translation of that audio to the formal language that is legible to the MTRS.

```
Processing audio_files/trigger1.wav

Creating trigger

Transcribed trigger:  {'text': ' The battery on UAS6 is below 25.',
'chunks': [{'timestamp': (0.0, 4.0), 'text': ' The battery on UAS6
is below 25.'}]}

Received UASBattery(UAS6, I(0,25)) and UASBatteryDuration(UAS6,
I(0.0,4.0)) with invalid syntax. Retrying 1.

MTRS trigger:  UASBattery(UAS6, I(0,25))


Processing audio_files/trigger2.wav

Creating trigger

Transcribed trigger:  {'text': " The mobility on UAS-6 is set to
waypoint and mobility autonomy on UAS-6 has started. And the health
of UAS-6's auxiliary is less than 100.", 'chunks': [{'timestamp':
(0.0, 9.08), 'text': ' The mobility on UAS-6 is set to waypoint and
mobility autonomy on UAS-6 has started.'}, {'timestamp': (9.08,
12.76), 'text': " And the health of UAS-6's auxiliary is less than
100."}]}

MTRS trigger:  AutonomyStatus(UAS6, Mobility, Started) and
VehicleSubHealthValue(UAS6, Auxiliary, I(0,100))


Processing audio_files/trigger3.wav

Creating trigger

Transcribed trigger:  {'text': ' Vehicle UAS-6 has been hit and the
sub crew task to primary is not 1G.', 'chunks': [{'timestamp': (0.0,
7.24), 'text': ' Vehicle UAS-6 has been hit and the sub crew task
to primary is not 1G.'}]}

MTRS trigger:  VehicleHit(UAS6, true) and not
VehicleTaskSubCrew(UAS6, Primary, 1G)


Processing audio_files/trigger4.wav

Creating trigger

Transcribed trigger:  {'text': ' The hostile vehicle BMP3 is
anticipated at a distance of less than 500.', 'chunks':
[{'timestamp': (0.0, 6.56), 'text': ' The hostile vehicle BMP3 is
anticipated at a distance of less than 500.'}]}
```

Received HostileConfidence(BMP3, I(0,inf)) and DetectionTypePosition(BMP3, BMP3, I(0,500)) with invalid syntax. Retrying 1.

Received DetectionTypePosition(BMP3, BMP3, I(0,500)) and Hostile(BMP3, 'anticipated') with invalid syntax. Retrying 2.

Received HostileConfidence(BMP3, I(0,500)) with invalid syntax. Retrying 3.

Received Hostile(BMP3, true) and VehicleTerrainOpen(BMP3, true) and MovementPositionLat(BMP3, I(0,inf)) and MovementPositionLon(BMP3, I(0,inf)) and DetectionTypeConfidence(BMP3, BMP3, I(0,500)) with invalid syntax. Retrying 4.

Received HostileConfidence(BMP3, I(0,500)) with invalid syntax. Retrying 5.

MTRS trigger: Hostile(BMP3, true) and VehicleTerrainHostile(BMP3, true) and DetectionTypePosition(BMP3, 'Any', I(0,500))

Processing audio_files/trigger5.wav

Creating trigger

Transcribed trigger: {'text': ' Hostile fire is anticipated at a distance of less than 200 from MCV-4.', 'chunks': [{'timestamp': (0.0, 6.56), 'text': ' Hostile fire is anticipated at a distance of less than 200 from MCV-4.'}]}

Received DetectionTypeConfidence(MCV4, Hostile, I(0,inf)) with invalid syntax. Retrying 1.

Received DetectionTypePosition(MCV4, Hostile, I(0,200)) and DetectionTypeConfidence(MCV4, Hostile, I(0,inf)) with invalid syntax.
Retrying 2.

Received DetectionTypePosition(MCV4, Hostile, I(0,200)) with invalid syntax. Retrying 3.

Received Detection(MCV4, Hostile, Any, I(0,200)) with invalid syntax.
Retrying 4.

Received DetectionTypePosition(MCV4, Hostile, I(0,200)) with invalid syntax. Retrying 5.

MTRS trigger: HostileConfidence(Any, I(0,200))

Processing audio_files/trigger6.wav

Creating trigger

Transcribed trigger: {'text': ' Subcrew 4A is tasked to vehicle UA 6 auxiliary task and is not tasked to UAS 6 task.', 'chunks': [{'timestamp': (0.0, 8.54), 'text': ' Subcrew 4A is tasked to vehicle UA 6 auxiliary task and is not tasked to UAS 6 task.'}]}

MTRS trigger: VehicleTaskSubCrew(UAS6, Auxiliary, 4A) and not VehicleTaskSubCrew(UAS6, Primary, 4A)

Processing audio_files/trigger7.wav

Creating trigger

Transcribed trigger: {'text': ' RCV2 is an urban terrain.', 'chunks': [{'timestamp': (0.0, 2.68), 'text': ' RCV2 is an urban terrain.'}]}

MTRS trigger: VehicleTerrainUrban(RCV2, True)

Processing audio_files/trigger8.wav

Creating trigger

Transcribed trigger: {'text': ' The battery on UAS-5 is below 10, and either vehicle UAS-5 has been hit, or RCV-2 is not in urban terrain.', 'chunks': [{'timestamp': (0.0, 7.92), 'text': ' The battery on UAS-5 is below 10, and either vehicle UAS-5 has been hit, or RCV-2 is not'}, {'timestamp': (7.92, 8.84), 'text': ' in urban terrain.'}]}

MTRS trigger: UASBattery(UAS5, I(0,10)) and (VehicleHit(UAS5, true) or not VehicleTerrainUrban(RCV2, true))

Processing audio_files/trigger9.wav

Creating trigger

Transcribed trigger: {'text': ' The battery on UAS-5 is between 10 and 30. Vehicle UAS-5 has been hit and is in an urban terrain, and hostile fire is anticipated at a distance of less than 200 from MCB-4.', 'chunks': [{'timestamp': (0.0, 14.0), 'text': ' The battery on UAS-5 is between 10 and 30. Vehicle UAS-5 has been hit and is in an urban terrain, and hostile fire is anticipated at a distance of less than 200 from MCB-4.'}]}

Received UASBattery(UAS5, I(10,30)) and VehicleHit(UAS5, true) and VehicleTerrainUrban(UAS5, true) and Hostile(MCB4, true) and HostileConfidence(MCB4, I(0,200)) with invalid syntax. Retrying 1.

Received UASBattery(UAS5, I(10,30)) and VehicleHit(UAS5, true) and VehicleTerrainUrban(UAS5, true) and DetectionTypeConfidence(MCB4, Hostile, I(0,200)) with invalid syntax. Retrying 2.

Received UASBattery(UAS5, I(10,30)) and VehicleHit(UAS5, true) and VehicleTerrainUrban(UAS5, true) and Detection(MCB4, Hostile, Primary, I(0,200)) with invalid syntax. Retrying 3.

Received UASBattery(UAS5, I(10,30)) and VehicleHit(UAS5, true) and VehicleTerrainUrban(UAS5, true) and DetectionTypeConfidence(MCB4, Hostile, I(0,200)) with invalid syntax. Retrying 4.

Received UASBattery(UAS5, I(10,30)) and VehicleHit(UAS5, true) and VehicleTerrainUrban(UAS5, true) and HostileConfidence(MCB4, I(0,200)) with invalid syntax. Retrying 5.

MTRS trigger: UASBattery(UAS5, I(10,30)) and VehicleHit(UAS5, true) and VehicleTerrainUrban(UAS5, true) and DetectionTypeConfidence(MCB4, Hostile, I(0,200))

Appendix B. Modular Trigger Response System (MTRS) Front-End User Guide

This section describes the **MTRS front-end user interface (UI)** (the *Front end*) and serves as a guide for users operating the system.

B.1 MTRS Schema

The front end will not function unless a valid MTRS Schema (described in Appendix C) is present within the executable's folder. For consistency, the schema used in the creation of this guide is reproduced in Section B.12. Whenever this appendix refers to "the schema," this is the document being referenced.

B.2 Definitions

Rule: The compilation of Triggers and Responses

Condition: The logical association of assets, numeric comparators, Boolean logic, and other user-defined parameters comprising an individual Trigger or Response. Delineated in the MTRS front end with bullet points.

Repository: A container of Rules

B.3 Configuration File

Several features for the front end can be configured using the provided *config.json*. A default config could look like the following:

```
{
  "schema": "schema-example",
  "readOnly": false,
  "llmEditEnable": true,
  "llmExplainEnable": true,
  "llmDebug": false,
  "llmUri": "http://localhost:11434",
  "llmTriggerModel": "llama3.3",
  "llmResponseModel": "llama3.3",
  "llmSplitModel": "llama3.3",
  "llmExplainModel": "llama3.3",
  "contextLength": 4092,
  "enableLargePrompts": false
}
```

- schema: The name of the schema JSON the front end should load. The file should be in the same directory as the Front end's executable.
- readOnly: Setting this to true will disable most of the editing functionality, aside from the large language model (LLM) interactions if those are enabled.
- llmEditEnable: If this is true, the "Start Recording," "Send to LLM," and transcription boxes will be enabled, otherwise they will not appear.

- llmExplainEnable: If true, this will enable the "?" button on each rule item, otherwise they will not appear.
- llmUri: The Front end connects to an Ollama server running at this address.
- contextLength: The maximum number of tokens the Ollama model will keep in memory.
- enableLargePrompts: For systems with more memory, which can handle larger context sizes, the Front end includes larger pre-prompts that include additional examples for the LLM to improve its rule-making accuracy. This option toggles their availability.

LLM Model Selection

For best results, the MTRS Front end uses a multi-staged approach for its use of LLMs. Some models perform better at certain stages than others, so the config file has the ability to specify which model to use for which stage.

- llmTriggerModel: The model to use when building Triggers
- llmResponseModel: The model to use when building Responses
- llmSplitModel: An LLM is used to split the natural language prompt into "If/Then" portions, corresponding to Triggers and Responses. This option defines which model to use for this operation.
- llmExplainModel: The model to use when explaining an MTRS back end (*formal*) rule in a natural language format.

B.4 Main Screen

Once a valid MTRS Schema is implemented, the Front end of the MTRS interface will display. Figure B-1 exhibits the main screen, with each functionality numbered and described below.

Figure B-1. The MTRS front-end's main screen, as seen when the software is first started.

1) File control
    a. New
    b. Open
    c. Save
    d. Save As
2) Export MTRS Back-end File
3) Save Rule to "Active Rules"
4) Generate Rule Name
5) Rule Name Edit field
6) Clear all triggers and responses from the currently active rule
7) Add Trigger Condition
8) Add Response Condition
9) MTRS Output preview
10) LLM prompt/Voice transcript display
11) Voice recording control
12) Send currently displayed prompt to LLM
13) Active Rule Repository
14) Inactive Rule Repository

B.5 Rule Creation Workflow

In the MTRS front end, Triggers and Responses (collectively called "rules") are organized into "condition blocks" that each reference an asset as defined in the Schema.

To add a Trigger or Response, click the "+" button in the column corresponding to that which you wish to add.

In the window that opens, drill down the available categories to find which object you wish to make a Condition for; in this example (Figure B-2), we add a trigger for MCV1:

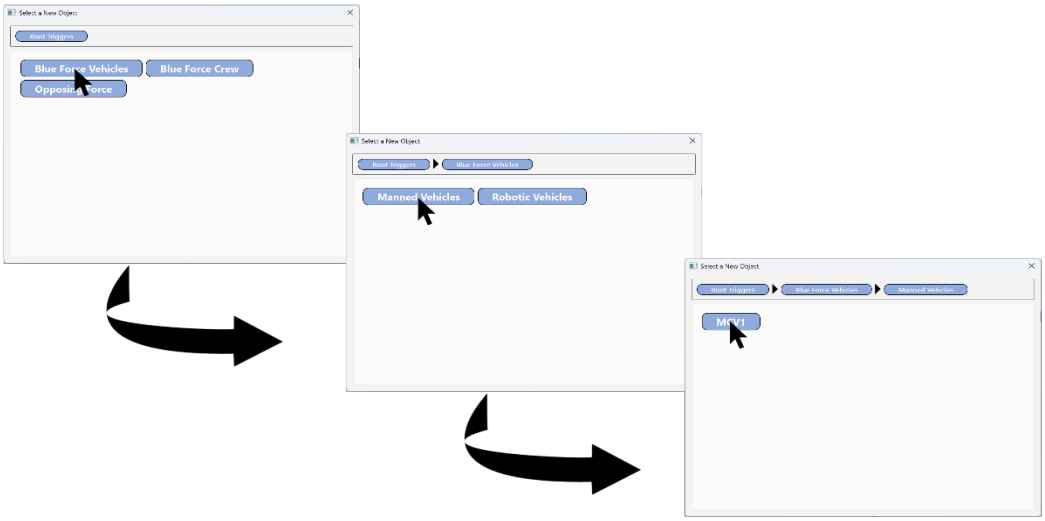


Figure B-2. Object creation workflow.

1) *Click "Blue Force Vehicles"*
2) *Click "Manned Vehicles"*
3) *Click "MCV1"*

Next, a menu will appear overlaying the Rule area; these are available Conditions for the asset you selected. For this example, we want to add a Trigger that will fire on low MCV health:

- *Select "Vehicle" to expand the available Triggers in the Vehicle category*
- *Select "Vehicle Health %" to add this trigger to the newly created block*

From here, the percentage of interest can be edited. The green "is" can also be clicked to see a drop-down list of other comparison options, the text of which is edited in the Schema. Further triggers and responses can be added using the same steps.

B.6 Condition Block

Each condition block is editable as defined in the schema. Figure B-3 shows each of the interactive elements and an example application. The **colored** texts are the interactive elements, and the text boxes can be edited freely. The text on the left, written vertically, is this block's "reference" asset, which will be explained further in the section on Condition Editing.

Each condition block contains one or more individual conditions that make up the block. Pressing the trash can button in the top right will delete the entire condition block, and pressing the bullet point to the left of the condition will delete just that condition. Additional conditions can be added by pressing the "+" button.

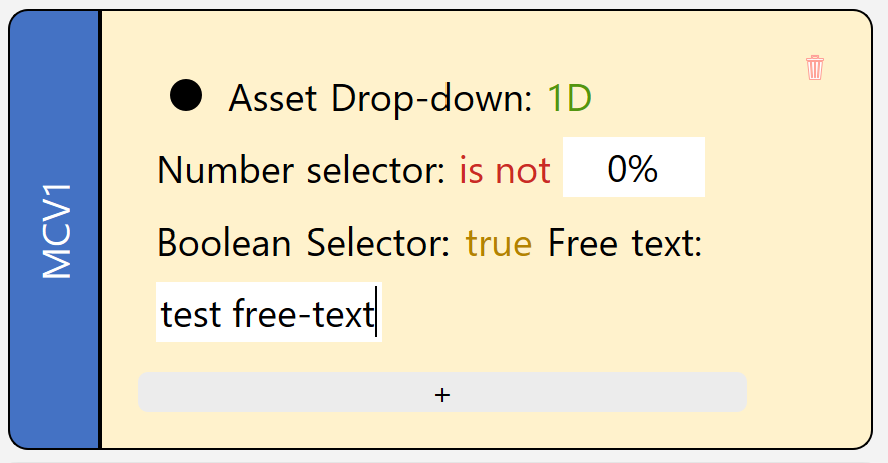


Figure B-3. An example of a condition block.

B.7 Condition Editing

Each condition has elements that can be edited depending on their type and how they were defined in the Schema. The types of interactive elements are listed below.

B.7.1 Drop-Down Selections

These are always colored green when editable. Clicking the green text will show a drop-down list of assets as defined in the Schema, any of which can be selected.

B.7.2 Referenced Assets

A special "variant" of the drop-down selection element. When a drop-down list contains the current block's referenced asset, it will be automatically selected and become non-editable, which will additionally color the element black like the rest of the non-editable text.

B.7.3 Comparison Selector

This is another type of drop-down selection that is restricted to only selecting comparisons "equal," "not equal," "greater than," or "less than"; the exact text is editable within the Schema. These elements will change color depending on their selection:

- Green when selecting the "equivalent" option
- Red when selecting the "not equivalent" option

- Gold when selecting the “greater than” or “less than” option

The drop-down lists are always paired with a text box, which only accepts numeric values, and will automatically append a unit according to the Schema.

B.7.4 Boolean (True/False) Selector

A toggle that will be Gold if set to true and Red if set to false. The exact text of this element is defined in the Schema.

B.7.5 Free-Text

A text box that accepts any input.

B.8 Rule Name

Once you have finished adding Conditions to a rule, it will need a name before it can be saved. You can either type a name in manually in the box at the top center of the Condition area, or you can click the “Auto-Generate Rule” button. You can then click the “save” button at the top of the Condition area, and the rule will move to the “Active” rules list.

B.9 Active/Inactive Repository

The active and inactive repositories provide a way to selectively “disable” rules without deleting them. Any rule in the inactive repository will not be included in the MTRS rule file when exporting a rule set. To move a rule between the repositories, click and drag the rule to the appropriate repository (Figure B-4).

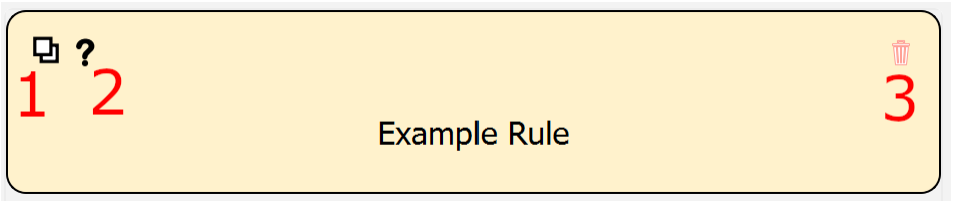


Figure B-4. Saved rule item.

Rules in the repository have the following options:

1) Copy rule
2) Ask the LLM to explain the rule
3) Delete rule
4) (Click anywhere on the rule, except on the above buttons): select the rule for editing
5) (Click and drag the rule): move the rule to the active/inactive repository

B.10 Saving/Loading

Using “File -> Save” or “File -> Save As” will allow you to save your currently active and inactive repositories to file. These files will have the .mtrs extension, and they are saved as JSON documents.

Important: These .mtrs files are **only** for the front end, to create a file that the MTRS back end can read (see the next section on Exporting).

Exporting to the MTRS Back End

Once the rules are complete, and the rules you want to be exported to the MTRS back end are in the **“active” repository**, click the “Export Rules” button, or use “File -> Export Rules.” This will export the active rules to a .tsv file that can be read by the MTRS back end.

Important: The exported .tsv is a one-way process, the MTRS Front end cannot read-in these files. It is recommended that you save your work with “File -> Save” to save in a format useful for the MTRS front end.

B.11 LLM Functionality

The MTRS additionally has some support for generating rules via an LLM. To begin the process, press the “Start Recording” button.

Note: The first time the button is pressed will require an additional second for the transcription functionality to load.

After the recording has started, dictate the rule; this dictation can be in natural language, and the LLM will translate into a format the MTRS understands. Once finished, click the “Finish Recording” button, and the dictation will be transcribed and inserted into the box above the recording button. If the transcription is not accurate, you may re-record.

Once the transcription reads to your satisfaction, press the “Send to LLM” button, and the LLM will take some time to process the natural language rule. When the rule is finished, it will be added to the end of the Active rules repository.

If there are any errors from the LLM, the MTRS Front end will display a window with the output of the LLM and a short description of the error (Figure B-5).

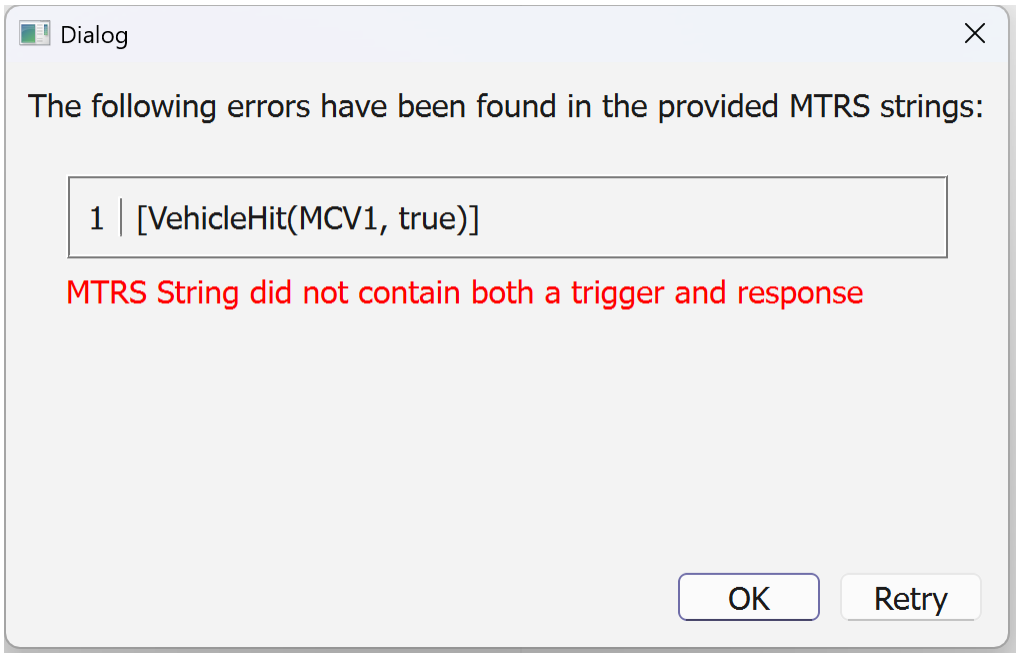


Figure B-5. LLM rule error window.

LLM Rule Explanations

To have the LLM break down the components of a rule, press the "?" button in the top left of the rule item. After the LLM is finished processing the rule, its explanation will display in a window that looks like Figure B-6.

Pressing the "text-to-speech" button in the bottom right will read the explanation out loud. The speech output can be stopped by pressing the button again.

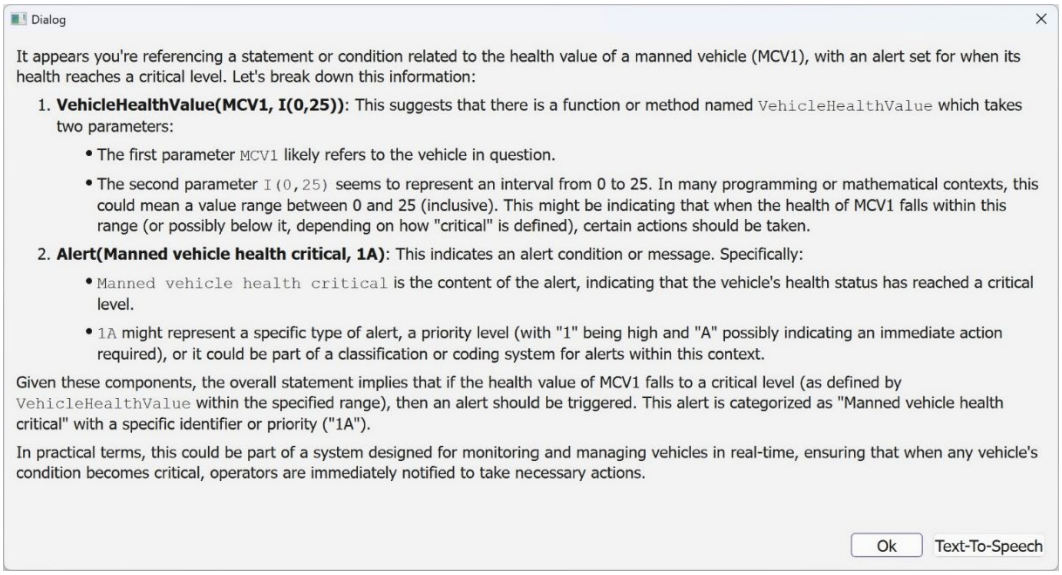


Figure B-6. LLM rule explanation.

B.12 Example Front-End Schema

Below is the MTRS front-end Schema used in the creation of this guide.

```
{
   "root-objects": [
      {
"category": "blufor-vehicle",
"display-name": "Blue Force Vehicles",
"children": [
            {
               "category": "manned-vehicle",
               "display-name": "Manned Vehicles",
               "members": [
"MCV1"
               ]
            },
            {
               "category": "robotic-vehicle",
               "display-name": "Robotic Vehicles",
               "members": [
"RCV2", "UAS3"
               ]
            }
],
"members": [
            "Any Vehicle"
]
      },
      {
"category": "blufor-crew",
"display-name": "Blue Force Crew",
"children": [
            {
               "category": "commander",
               "display-name": "Commanders",
               "members": [
"1A"
               ]
            },
            {
               "category": "driver",
               "display-name": "Drivers",
               "members": [
"1D", "2D"
               ]
            },
            {
               "category": "gunner",
               "display-name": "Gunners",
               "members": [
"1G", "2G"
               ]
            }
]
      },
      {
```

```
"category": "tasking",
"members": [
            "Driver", "Gunner"
]
       },
       {
"category": "opfor",
"display-name": "Opposing Force",
"members": [
"T-90", "BTR82"
]
       }
    ],
    "expanding": {
       "item": "Any Vehicle",
       "type": "vertical",
       "categories": [
"blufor-vehicle"
       ]
    },
    "links": [
       {
"item": "MCV1",
"matches": [
              {
                 "item":
                 "CommanderX",
                 "match": [
"1A"
                 ]
              },
              {
                 "item":
                 "DriverX",
                 "match": [
"1D"
                 ]
              },
              {
                 "item":
                 "GunnerX",
                 "match": [
"1G"
                 ]
              },
              {
                 "item":
                 "VehicleX",
                 "match": [
"MCV1"
                 ]
              }
]
       },
       {
"item": "RCV2",
"matches": [
```

```
                {
                    "item":
                    "CommanderX",
                    "match":  [
"1A"
                    ]
                } ,
                {
                    "item":
                    "DriverX",
                    "match":  [
"2D"
                    ]
                } ,
                {
                    "item":
                    "GunnerX",
                    "match":  [
"2G"
                    ]
                } ,
                {
                    "item":
                    "VehicleX",
                    "match":  [
"RCV2"
                    ]
                }
]
        } ,
        {
"item":  "UAS3",
"matches":  [
                {

                    "item":
                    "VehicleX",
                    "match":  [
"UAS3"
                    ]
                }
]
        }
    ] ,
    "expansion-priority":  {
    "vertical":  [
"Any Vehicle"
        ] ,
        "horizontal":  []
    } ,
    "link-priority":
        [ "VehicleX",
        "CommanderX",
        "DriverX",
        "GunnerX"
    ] ,
    "filters":  [
```

```
        {
"object": "MCV1", "blocklist":  [
"2G", "2D"
]
        } ,
        {
"object": "RCV2", "blocklist":  [
"1A",
             "1G", "1D"
]
        } ,
        {
"object": "UAS3", "blocklist":  [
             "1A",
             "1G",
             "1D",
             "2G",
             "2D"
]
        }
    ] ,
    "triggers":  [
        {
"groups":[  "Vehicle"  ] ,
"name": "Vehicle Health %",
"short":  "VehHPPercent",
"input": "{blufor-vehicle} health {number#health}",
"number-params":  [
             {
                "name": "health",
                "type": "integer",
                "max":  100,
                "min": 0,
                "equal-only": false,
                "unit":  "%",
                "equality-text":  {
                   "equal": "is",
                   "not-equal": "is not",
                   "greater": "above",
                   "less":   "below"
                }
             }
     ] ,
"output":  "VehicleHealthValue({blufor-vehicle},
{health})"
} ,
{
"groups":  ["Tasking"],
"name": "Crew Tasking",
"short":  "CrewTask",
"input":  "{blufor-crew} tasked to {tasking} on
{blufor-vehicle}",
"output":  "Tasking({blufor-vehicle}, {blufor-crew},
{tasking})"
} ,
{
     "groups":  ["Example"],
     "name":  "Interactive Example",
```

```
      "always-relevant": true,
      "short": "Example",
"input": "Asset Drop-down: {blufor-crew}
Number selector: {number#example-number} Boolean
"output": "ExampleCondition({blufor-crew}, {example-
number}, {example-bool}, {example-text})",
"number-params": [
            {
               "name": "example-number",
               "type": "integer",
               "max": 100,
               "min": 0,
               "equal-only": false,
               "unit": "%",
               "equality-text": {
                  "equal": "is",
                  "not-equal": "is not",
                  "greater": "above",
                  "less": "below"
               }
               }
         ],
         "bool-params": [
            {
               "name": "example-bool",
               "negate-rule": false,
               "true": "true",
               "false": "false"
            }
         ]
      }
],
   "responses": [
      {
      "groups": [
          "Alert"
         ],
          "name": "Alert Crew Member",
          "short": "Alert",
          "input": "Alert {blufor-crew} : {free-
          text#context}",
          "output": "Alert({context}, {blufor-
          crew})"
      }
   ]
}
```

Appendix C. Modular Trigger/Response System (MTRS) Schema Documentation

This document describes the **MTRS Front-End Schema,** which informs the front end user interface (UI) what assets are available, how to organize those assets into triggers and responses, and how to translate those triggers and responses to the MTRS back end.

An example schema is provided in Section C.7.

The schema is a JSON file. A blank schema is found below, and each key will be explained in detail in the following sections.

```
{
      "root-objects": [],
      "triggers": [],
      "responses": []
}
```

C.1 Root Objects

Root Objects describe assets available to the MTRS. Objects are organized into categories, with each category having members (the assets themselves) and/or children, which are subcategories and use the Root Object format as described here. **Objects must have either members or children, but they may have both.**

```
"root-objects": [
      {
         "category": "",
         "display-name": "",
         "restriction": "",
         "members": [],
         "children": []
      }
]
```

Members are strings that may have special characters and spaces. Members may be individual assets or wildcards, as described in the sections below.

C.1.1 Category: Required

This is a unique identifier used within the trigger and response blocks to refer to this asset. Spaces and special characters are not allowed, `except -`.

In addition, the following are reserved keywords and must not be used: `- bool - number - free-text - gcm`.

C.1.2 Display Name

This is a string used as a replacement for the category string in the UI. Spaces and special characters (aside from new lines) are allowed. If this option is not defined, the category is used as is.

C.1.3 Restriction

This constrains this asset to only appear in either trigger or response conditions.

Must be one of `- trigger - response`.

C.2 Expansions and Links (Wildcards)

These are for simplifying the process of creating many rules. If an environment has access to many vehicles, a user can use "Any Vehicle" as a wildcard and define which assets should be included. Additional information on the usage of wildcards can be found in the walk-through below.

C.2.1 Expansion Definitions

```
"expanding":  [
       {
          "item":  "",
          "type":  "",
          "categories":  [],
          "other-entries":  []
       }
]
```

- **Item – Required**

The name of the asset (as defined in root-objects) that should be treated as an expanding wildcard

- **Type – Required**

Vertical or horizontal:

- **Vertical Expansions**

Vertical expansion will create a new rule for each entry, i.e., after final processing, `Rule("Any Vehicle")` will be sent to the MTRS back end as follows:

```
1) Rule(MCV1)
2) Rule(RCV2)
3) Rule(UAS3)
```

- **Horizontal Expansions**

`Horizontal` expansion will logically `AND` each entry into a single rule, i.e., after final processing, `Rule("All Vehicles")` is sent to the MTRS back end as: `Rule(MCV1) AND Rule(RCV2) AND Rule(UAS3).`

- **Categories – Required**

The category this wildcard should expand into; *Note*: expansions will never expand into themselves or other expansions

- **Other-Entries**

In the event that no specific category covers all needed options, this array can be used to add additional entries. When both `categories` and `other-entries` are present, the wildcard is expanded into the `categories`, and then the `other-entries`.

C.2.2 Link Definitions

```
"links":  [
      {
         "item":  "",

         "matches":[
            {
               "item":  "",
               "match":  []
            }
         ]
      }
]
```

Link wildcards function by matching appropriate assets together. For example, if you have a rule that reads as `(Trigger(MCV1) Response(DriverX)`, then the goal is to replace DriverX with MCV1's driver (1D), which would look as follows:

```
{
      "item":  "MCV1",
      "matches":  [
         "item": "DriverX",
         "match":  [
      "1D"
         ]
      ]
}
```

Another way of describing the block above could be as follows: > "For the vehicle `MCV1`, any time `DriverX` is encountered, we are referring to the crew member 1D, and the `DriverX` text should be replaced accordingly.

Note: `DriverX` may be any text and is used here for illustrative purposes. It is recommended that all wildcard text be distinct from regular asset names to avoid confusion.

C.3 Link Priority

```
"link-priority": []
```

This list defines the order in which wildcard linking will be performed, from top to bottom. Each entry is a string that must match one of the Link definitions.

C.4 Expansion Priority

```
"expansion-priority": {
    "vertical": [],
    "horizontal": []
}
```

Each list of expansions defines the order in which each wildcard should be expanded. Each item must be a string that matches a previously defined entry in the expansion definitions.

C.5 Filters

```
"filters": [
    {
        "object": "",
        "blocklist": []
    }
]
```

If certain assets cannot interact—for example, a crew member who should not be paired with certain vehicles—filters can be created that will mark any rules that contain filtered pairings as invalid.

C.6 Triggers/Responses

Triggers and responses (collectively called "Conditions") can be thought of as "Mad Libs" style "fill-in-the-blank" building blocks. These take a natural-language description of an event (for triggers), or an action for the system to take (for responses), and provide a curated list of assets to insert in each placeholder box. Triggers and Responses both use the same format:

```
{
    "name": "",
    "groups": [],
    "short": "",
    "input": "",
    "output": "",
    "always-relevant": false,
    "number-params": [],
```

```
        "bool-params":  []
}
```

C.6.1 Name: Required

The name of this condition must be unique between all other conditions, and it will be displayed in the UI when selecting a new condition.

C.6.2 Groups: Required

An array of the groups into which this condition will be sorted. It can be any string.

C.6.3 Short: Required

A "shortened" name for this condition, and it is used when automatically generating a rule name.

C.6.4 Input: Required

A natural-language description of the trigger or system response. To create a placeholder for a user to fill in with an asset, insert the asset's category, surrounded by braces {}. This placeholder can be named by adding the name after a hashtag #. Spaces and special characters are not allowed within placeholders.

Other options include the following: multiple asset categories can be joined in one placeholder with the pipe | character; the placeholder can be given a different name with the hashtag # character.

See the below sections "number-params" and "bool-params" for more information about inserting numeric or Boolean placeholders.

C.6.5 Output: Required

The formal function used by the MTRS back end into which the input should be converted. Like inputs, placeholders are defined with the braces {}, but they **must** use the name of the placeholder as defined in the input. Input and output placeholders do not need to share the same order.

C.6.6 Always-Relevant

This option defaults to `false`.

In the MTRS front-end UI, conditions are only shown if a relevant asset has been selected; for example, if a crew member such as a driver has been selected, conditions that only rely on vehicle assets will be hidden. This behavior can be overridden by setting this value to true.

C.6.7 Number-params

When building an input, if it is desired to enable the ability to select some numeric value, insert `{number#name}` as a placeholder within the input string. Here, the name is a string without spaces or special characters. Each numeric placeholder **must** have an entry in the number-params list, which looks like the following:

```
{
      "name": "",
      "equal-only": true/false,
      "type": "",
      "unit": "",
      "min": 0,
      "max": 0,
      "equality-text": {
         "equal": "",
         "not-equal": "",
         "greater": "",
         "less": ""
      }
}
```

C.6.7.1 Name: Required

This string **must** match the name given to the placeholder as per the description above.

C.6.7.2 Equal-Only: Required

If true, it will only display a box for a number. If false, it will additionally include a drop-down to select an inequality (see below section on equality-text), which will be required if equal-only is true

C.6.7.3 Type: Required

Must be either “integer” or “float.” If set to integer, a user may only select whole numbers. If set to float, the number selection will enable decimal values (up to two decimal places).

C.6.7.4 Unit

This string can contain spaces and special characters, and it will be appended after the number as a unit. Default is no unit.

C.6.7.5 Min/Max

These set the minimum and maximum values the user can set. Default is no min or max.

C.6.7.6 Equality-Text

The values set the text to show when a user is selecting inequalities. If equal-only is set to false, these values are all required.

C.6.8 Bool-Params

```
{
      "name": "",
      "negate-rule":
      true/false, "true": "",
      "false": ""
}
```

C.6.8.1 Name: Required

This string must match the name given to the placeholder, as per the description above.

C.6.8.2 Negate-Rule: Required

If true, this will prepend NOT to the final rule, if the user selects sets the value to false. If false, it will require an accompanying placeholder in the output string, and it will set that placeholder to true/false as per user selection.

C.6.8.3 True/False: Required

String values that will display to the user when the Boolean placeholder is set to the corresponding value.

C.6.8.4 Output: Required

The back-end function this condition is building. Placeholders can be added, again, by using braces {}. For outputs, these placeholders should correspond to the name of a placeholder in the input.

C.6.9 Basic Trigger Example

For a potential trigger for a vehicle having its mobility damaged, the condition configuration could look like the following:

```
{
      "name": "Blufor Vehicle Mobility Damaged",
      "groups": ["Damage"],
      "short":  "BlufMobDam",
      "input": "{vehicle#blufor} mobility damaged",
      "output": "VehicleMobilityDamage({blufor})
}
```

C.7 Schema Example and Walkthrough

This walk-through will focus on building a Schema with the following assets:

- BLUFOR
  - 1 manned vehicle (MCV1)
    - Crewed by 1 commander (1A), 1 driver (1D), and 1 gunner (1G)
  - 1 remote vehicle (RCV2)
    - Operated by 1 driver (2D) and 1 gunner (2G)
  - 1 UAS (UAS3)
- OPFOR
  - T-90
  - BTR82

C.7.1 Root-Objects

To begin, we define the above assets in the root-objects list. For the blufor vehicles, add a new object to the list with the following content:

```
{
      "category": "blufor-vehicle",
      "display-name": "Blue Force Vehicles",
      "children":  []
}
```

For better organization, the three vehicles can be organized into two categories: *Manned Vehicles* and *Robotic Vehicles*, which will continue the blufor category like the following:

```
{
      "category": "blufor-vehicle",
      "display-name": "Blue Force Vehicles",
      "children": [
          {
              "category": "manned-vehicle",
              "display-name": "Manned Vehicles",
              "members": [
                  "MCV1"
              ]
          },
          {
              "category": "robotic-vehicle",
              "display-name": "Robotic Vehicles",
              "members": [
                  "RCV2", "UAS3"
              ]
          }
      ]
}
```

Blufor crew members can be organized in a similar way:

```
{
      "category": "blufor-crew",
      "display-name": "Blue Force Crew",
      "children": [
          {
              "category": "commander",
              "display-name": "Commanders",
              "members": [
                  "1A"
              ]
          },
          {
              "category": "driver",
              "display-name": "Drivers",
              "members": [
                  "1D", "2D"
              ]
          },
          {
              "category": "gunner",
              "display-name": "Gunners",
              "members": [
                  "1G", "2G"
              ]
          }
      ]
}
```

Finally, the Opfor assets can be defined in a simpler manner, with only a single category.

```
{
```

```
        "category": "opfor",
        "display-name": "Opposing Force",
        "members": [

            "T-90",
            "BTR82"
        ]
}
```

All together, the defined root-objects appear as follows:

```
{
        "root-objects": [
            {
                "category": "blufor-vehicle",
                "display-name": "Blue Force Vehicles",
                "children": [
                    {
                        "category": "manned-vehicle",
                        "display-name": "Manned Vehicles",
                        "members": [
                            "MCV1"
                        ]
                    },
                    {
                        "category": "robotic-vehicle",
                        "display-name": "Robotic Vehicles",
                        "members": [
                            "RCV2",
                            "UAS3"
                        ]
                    }
                ]
            },
            {
                "category": "blufor-crew",
                "display-name": "Blue Force Crew",
                "children": [
                    {
                        "category": "commander",
                        "display-name": "Commanders",
                        "members": [
                            "1A"
                        ]
                    },
                    {
                        "category": "driver",
                        "display-name": "Drivers",
                        "members": [
                            "1D",
                            "2D"
                        ]
                    },
                    {
                        "category": "gunner",
                        "display-name": "Gunners",
                        "members": [
                            "1G",
                            "2G"
```

```
                    ]
                }

            ]
        } ,
        {
            "category": "opfor",
            "display-name": "Opposing Force",
            "members": [
                "T-90",
                "BTR82"
            ]
        }
    ]
}
```

C.7.2 Wildcards

Now, we want to be able to quickly build rules using the vehicles in this scenario, and we do not want to have to create one rule each for the MCV, RCV, and UAS. To speed this process up, we can create a vertically expanding rule that will automatically create one rule per vehicle when we select “Any Vehicle.”

```
"expanding": {
    "item": "Any Vehicle",
    "type": "vertical",
    "categories": [
        "blufor-vehicle"
    ]
}
```

Then, to further ease the creation of rules, we can create linking wildcards to allow the front end to fill in other components of rules, such as vehicle crew.

```
"links": [
    {
        "item": "MCV1",
        "matches": [
            {
                "item": "CommanderX",
                "match": [
                    "1A"
                ]
            } ,
            {
                "item": "DriverX",
                "match": [
                    "1D"
                ]
            } ,
            {
                "item": "GunnerX",
                "match": [
                    "1G"
                ]
```

```
            },
            {
                "item": "VehicleX",
                "match": [
                    "MCV1"
                ]
            }
        ]
    },
    {
        "item": "RCV2",
        "matches": [
            {
                "item": "CommanderX",
                "match": [
                    "1A"
                ]
            },
            {
                "item": "DriverX",
                "match": [
                    "2D"
                ]
            },
            {
                "item": "GunnerX",
                "match": [
                    "2G"
                ]
            },
            {
                "item": "VehicleX",
                "match": [
                    "RCV2"
                ]
            }
        ]
    },
    {
        "item": "UAS3",
        "matches": [
            {
                "item": "VehicleX",
                "match": [
                    "UAS3"
                ]
            }
        ]
    }
]
```

C.7.3 Wildcard Priority

When deciding what order to expand wildcards, it is best to take a "wide to narrow" approach; that is, to start with any wildcards that expand to the largest number of assets. For the single expanding wildcard, the expansion-priority object would look like the following:

```
    "expansion-priority": {
    "vertical": [
          "Any Vehicle"
       ],
       "horizontal": []
}
```

There is not a catch-all guide for link priorities, but some wildcards can inform others. For example, if a rule contains two wildcards, e.g., `VehicleX` and `DriverX`, the priority should be `VehicleX --> DriverX`, because the MTRS likely will not know which driver to use until the correct vehicle is inserted. To that end, the example link priority would be as follows:

```
"link-priority": [
      "VehicleX",
      "CommanderX",
      "DriverX",
      "GunnerX"
]
```

C.7.4 Filters

In this hypothetical scenario, crew members of the Blufor vehicles are disallowed from operating each other's vehicles.

To prevent accidentally assigning crews to each other's vehicles, we will set up some filters here.

```
"filters": [
      {
         "object": "MCV1",
         "blocklist": [
               "2G",
               "2D"
         ]
      },
      {
         "object": "RCV2",
         "blocklist": [
            "1A",
            "1G",
            "1D"
         ]
      },
      {
         "object": "UAS3",
         "blocklist": [
```

```
            "1A",
            "1G",
            "1D",
            "2G",
            "2D"
        ]
    }
]
```

C.7.5 Writing Rules for the Schema

Finally, we can begin writing the blank triggers and responses (the rules). To start, we will write a trigger that will activate when a vehicle's health reaches a selectable percentage:

```
{
    "groups":[ "Vehicle" ],
    "name": "Vehicle Health %",
    "short": "VehHPPercent",
    "input": "{blufor-vehicle} health
    {number#health}",
    "number-params": [
        {
            "name": "health",
            "type": "integer",
            "max": 100,
            "min": 0,
            "equal-only": false,
            "unit": "%",
            "equality-text": {
                "equal": "is",
                "not-equal": "is not",
                "greater": "above",
                "less":  "below"
            }
        }
    ],
    "output": "VehicleHealthValue({blufor-vehicle},
    {health})"
}
```

Next, we will write a response that sends an alert to a crew member:

```
{
    "groups": [
        "Alert"
    ],
    "name": "Alert Crew Member",
    "short": "Alert",
    "input": "Alert {crew} : {free-text#context}",
    "output": "Alert({context}, {crew})"
}
```

This leads to our completed MTRS schema example.

C.8 Complete MTRS Schema Example

```
{
      "root-objects": [
         {
            "category": "blufor-vehicle",
            "display-name": "Blue Force Vehicles",
            "children": [
               {
                  "category": "manned-vehicle",
                  "display-name": "Manned Vehicles",
                  "members": [
"MCV1"
                  ]
               },
               {
                  "category": "robotic-vehicle",
                  "display-name": "Robotic Vehicles",
                  "members": [
"RCV2", "UAS3"
                  ]
               }
            ]
         },
         {
            "category": "blufor-crew", "display-name":
            "Blue Force Crew",
            "children": [
               {
                  "category": "commander", "display-name":
                  "Commanders", "members": [
"1A"
                  ]
               },
               {
                  "category": "driver", "display-name":
                  "Drivers", "members": [
"1D", "2D"
                  ]
               },
               {
                  "category": "gunner", "display-name":
                  "Gunners", "members": [
"1G", "2G"
                  ]
               }
            ]
         },
         {
            "category": "opfor",
            "display-name": "Opposing Force", "members": [
"T-90", "BTR82"
            ]
         }
      ],
      "expanding": {
         "item": "Any Vehicle", "type": "vertical",
```

```
            "categories": [
                "blufor-vehicle"
            ]
        },
        "links": [
            {
                "item": "MCV1",
                "matches": [
                    {
                        "item": "CommanderX", "match": [
"1A"
                        ]
                    },
                    {
                        "item": "DriverX", "match": [
"1D"
                        ]
                    },
                    {
                        "item": "GunnerX", "match": [
"1G"
                        ]
                    },
                    {
                        "item": "VehicleX", "match": [
"MCV1"
                        ]
                    }
                ]
            },
            {
                "item": "RCV2",
                "matches": [
                    {
                        "item": "CommanderX", "match": [
"1A"
                        ]
                    },
                    {
                        "item": "DriverX", "match": [
"2D"
                        ]
                    },
                    {
                        "item": "GunnerX", "match": [
"2G"
                        ]
                    },
                    {
                        "item": "VehicleX", "match": [
"RCV2"
                        ]
                    }
                ]
            },
```

```
            {
                "item": "UAS3",
                "matches": [
                    {
                        "item": "VehicleX", "match": [
"UAS3"
                        ]
                    }
                ]
            }
        ],
        "expansion-priority": { "vertical": [
                "Any Vehicle"
            ],
            "horizontal": []
        },
        "link-priority": [ "VehicleX", "CommanderX",
            "DriverX", "GunnerX"
        ],
        "filters": [
            {
                "object": "MCV1", "blocklist": [
"2G", "2D"
                ]
            },
            {
                "object": "RCV2", "blocklist": [
"1A",
                    "1G", "1D"
                ]
            },
            {
                "object": "UAS3", "blocklist": [
                    "1A",
                    "1G",
                    "1D",
                    "2G", "2D"
                ]
            }
        ],
        "triggers": [
            {
                "groups":[ "Vehicle" ],
                "name": "Vehicle Health %",
                "short": "VehHPPercent",
                "input": "{blufor-vehicle} health
                {number#health}",
                "number-params": [
                    {
                        "name": "health",
                        "type": "integer",
                        "max": 100,
                        "min": 0,
                        "equal-only": false,
                        "unit": "%",
                        "equality-text": {
                            "equal": "is",
```

```
                    "not-equal": "is not",
                    "greater": "above",
                    "less":  "below"
                }
            }
        ],
        "output": "VehicleHealthValue({blufor-
        vehicle}, {health})"
    }
],
"responses": [
    {
        "groups": [
            "Alert"
        ],
        "name": "Alert Crew Member",
        "short": "Alert",
        "input": "Alert {crew} : {free-
        text#context}",
        "output": "Alert({context}, {crew})"
    }
]
}
```

Appendix D. MTRS Project Overview Documentation (Readme.md)

Modular Cue Response System

This repository contains a modular system for defining and responding to complex cues in a data stream. It is designed to read rules, match them against incoming data from Apache Kafka topics, and execute appropriate responses. The min_example folder demonstrates a "minimal" use case focused on vehicle tracking and alerting based on location and speed.

The system is built around a few core components:

- A **rule engine** that processes triggers and executes responses.
- A **Kafka interface** to consume data streams and publish responses.
- A set of **definitions** for the data relationships (predicates).
- An **entry point** to configure and run the system.

Main Components

These are all located in qdcr/min_example.

- **minimal.py**: This is the main entry point for running the application. It handles command-line arguments, sets up logging, initializes the Kafka producer and consumer, and contains the main processing loop.
- **min_relations.py**: This file defines the semantics for the triggers. It specifies the valid relations, their arguments, and their data types. For instance, the MovementPositionLat relation connects a vehicle (a string) with a latitude (a numerical interval).
- **min_rules1.tsv**: This is a tab-separated file containing the actual rules the system will use. Each line represents a rule, consisting of a trigger condition, a corresponding response, and a unique ID. The rules in this file are as follows:

MovementPositionLat(vehicle1, I(38.8, 38.9)) and MovementPositionLon(vehicle1, I(77.0, 77.1) Alert(Veh1WhiteHouse, Central) veh1WhiteHouse MovementPositionSpeed(vehicle1, I(100, inf)) or MovementPositionSpeed(vehicle2, I(100, inf) Alert(fastMoving, Central) fastMoving

The first checks if vehicle1 has a latitude and longitude in a give rectangle; if so, an Alert is sent. The second rule sends an Alert if either vehicle1 or vehicle2 is going above 100.

How It Works

The system operates by continuously monitoring specified Kafka topics for incoming messages.

1) **Rule Loading**: At startup, the system reads the rules from the .tsv file. It parses the trigger conditions and the expected responses.
2) **Kafka Consumption**: The application listens to Kafka topics for new messages (e.g., vehicle status updates).
3) **Message Processing**: Incoming messages are processed to match against the trigger conditions defined in the rules.
4) **Response Execution**: If a message satisfies the conditions for a rule's trigger, the system executes the corresponding response, such as sending an alert.

Trigger and Response Semantics

- **Triggers** are complex conditions based on the relations defined in min_relations.py. They can be combined using logical operators such as and and or. For example, a trigger might fire if a vehicle's latitude and longitude are within a certain geographic box.
- **Responses** are the actions the system takes when a trigger's conditions are met. In the provided example, the response is to generate an Alert.

Getting Started

Prerequisites

- Python 3
- An Apache Kafka broker running on your network. The default configuration assumes the broker is available at localhost:9092.

Installation

1) Clone the repository.
2) Install the required Python packages.

pip install -r requirements.txt

Running the Example

The main script minimal.py can be run from the command line. It accepts several arguments to configure its behavior.

python minimal.py --rules_file min_rules1.tsv --kafka_broker <your_broker_address>

Command-Line Arguments:

- --logfile: Specifies a file to write logs to.
- --loglevel: Sets the logging level (INFO, DEBUG, etc.).
- --nolog: Disables file logging.

- --nostdout: Disables logging to the console.
- --rules_file: Path to the rule definition file (e.g., min_rules1.tsv).
- --global_rule_lock: The cooldown period in seconds before a rule can be triggered again.
- --kafka_broker: The address of the Kafka broker.
- --refresh_interval: The delay in seconds between polling for new messages.

For example, to run with the provided rules and connect to a Kafka broker at localhost, you would use:

python minimal.py --rules_file min_rules1.tsv --kafka_broker localhost

The system will then start listening for messages on the relevant Kafka topics, and it will execute responses as the rules are triggered.

List of Symbols, Abbreviations, and Acronyms

AAR	After-Action Review
AI	Artificial Intelligence
AiTR	Aided Target Recognition
ASR	Automatic Speech Recognition
BLUFOR	Blue Force
CV	Computer Vision
EUD	End-User Development
FY	Fiscal Year
HMI	Human–Machine Integrated
ID	Identification
INFORMS	Information for Mixed Squads
JSON	JavaScript Object Notation
LLaMA	Large Language Model Meta AI
LLM	Large Language Model
MCV	Manned Control Vehicle
METT-TC	Mission, Enemy, Terrain, Troops, Time, and Civil Considerations
ML	Machine Learning
MTRS	Modular Trigger Response System
NGCV	Next-Generation Combat Vehicle
OPFOR	Opposition/Opposing Force
RCV	Robotic Combat Vehicle
REST	Representational State Transfer
SA	Situation Awareness
SIMEX	Simulation Experiment
TAP	Trigger-Action Programming

UAS	Unmanned Aerial System
UAV	Unmanned Aerial Vehicle
UI	User Interface

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CB TECH LIB