Noname manuscript No.
(will be inserted by the editor)

Building High Assurance Human-Centric Decision Systems

Constance L. Heitmeyer · Marc Pickett · Elizabeth I. Leonard · Indrakshi Ray · David W. Aha · J. Gregory Trafton · Myla M. Archer

Received: date / Accepted: date

Abstract Many future decision support systems will be human-centric, i.e., require substantial human oversight and control. Because these systems often provide critical services, high assurance will be needed that they satisfy their requirements. How to develop human-centric decision systems that are high assurance is unknown: while significant research has been conducted in areas such as adaptive agents, cognitive science, and formal methods, how to apply and integrate the design principles and disparate models in each area has not been studied. This paper proposes a process for developing human-centric decision systems where AI (artificial intelligence) methods—namely, cognitive models to predict human behavior and agents to assist the human—are used to improve system performance, and software engineering methods—namely, formal modeling and analysis—to obtain high assurance that the system behaves as intended. The paper describes a new method for synthesizing a formal model from Event Sequence Charts, a variant of Message Sequence Charts, and a Mode Diagram, which specifies sets of system modes and mode transitions. After reviewing a technique for synthesizing user models from human participant studies data, the paper presents the results of a new pilot study investigating the optimal level of agent assistance for different users; the agent design was evaluated using synthesized user models. Finally, the paper reviews a dynamic model for predicting human overload in complex human-centric systems. To illustrate the development process and our new techniques, we describe a human-centric decision system controlling unmanned air vehicles.

Keywords high assurance · formal models · formal methods · agents · adaptive agents · cognitive models · imitation learning · formal model synthesis · user model synthesis · decision systems · human-centric decision systems · scenarios · software requirements · system requirements

C. Heitmeyer · E. Leonard · D. Aha · J. G. Trafton · M. Archer
Naval Research Laboratory, Washington, DC 20375
E-mail: {constance.heitmeyer,elizabeth.leonard,david.aha,greg.trafton,myla.archer}@nrl.navy.mil

M. Pickett
NRC/NRL Postdoctoral Fellow, Naval Research Laboratory, Washington, DC 20375
E-mail: marc.pickett.ctr@nrl.navy.mil

I. Ray
Colorado State University, Fort Collins, CO 80523
E-mail: iray@cs.colostate.edu

1 Introduction

Many future decision systems will be *human-centric*—i.e., require substantial human oversight and control. Because these systems often provide critical services, high assurance will be needed that they satisfy their requirements. Systems controlling autonomous vehicles constitute one important and growing class of human-centric decision systems which require high assurance. Currently, the largest deployed class of systems which control autonomous vehicles manage UAVs (unmanned air vehicles): the U.S. military alone is estimated to deploy over 7,000 UAVs, compared to less than 50 a decade ago [8]. Many of these systems, which perform a range of challenging tasks including surveillance and targeting, are not entirely autonomous but remotely controlled by humans. In future years, human-centric systems managing autonomous vehicles are expected to be widely deployed in both military and non-military applications. For example, plans exist to use autonomous vehicles in law enforcement, where UAVs may be equipped with cameras and scientific instruments for surveillance and information gathering *and* with weapons, such as rubber bullets, Tasers, and tear gas. These non-military systems, e.g., for law enforcement and public safety, will be natural transitions from the military's decision systems.

How to design and build high assurance human-centric decision systems is largely unknown: while significant research has been published in areas such as intelligent agents, cognitive science, and formal methods, how to relate and integrate the design principles and disparate models in each area has not been studied. In combining the research results, difficult questions arise, e.g., what pair-wise model interactions are beneficial? Can design principles in one area be combined with those in another? Designing and building high assurance human-centric decision systems also poses major challenges. Because the human user of these systems performs many complex tasks, he/she will at times become overloaded. A major challenge is how to address human overload. A second major challenge is how to obtain high assurance that these systems behave as intended.

A promising approach to human overload is to use AI (artificial intelligence) methods—in particular, a cognitive model and an agent. The cognitive model's role is to predict human overload, while the agent's role is, upon notification by the cognitive model of human overload, to alert the human to a critical task he is currently ignoring or to take control of one or more of the human's tasks. This system design raises major questions. For example, how to design the system autonomy—e.g., which tasks to assign to humans and which to the "system," when to switch from human to system control of a task, and vice versa—needs further study [2]. A promising approach to the high assurance problem is to apply software engineering methods—namely, formal modeling and analysis. However, a huge problem is how to obtain the formal system requirements model. Difficult to obtain in general, formal models of requirements are especially hard to obtain for human-centric decision systems given their complexity. Moreover, even if the problem of obtaining a formal requirements model is overcome, major questions remain. For example, how does the formal model represent the requirements of a system composed of a cognitive model and an agent?

This article, an extension of a paper presented at the Second International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2013), is organized as follows. To illustrate the complexity of human-centric decision systems, Section 2 introduces an example of a human-centric decision system controlling a team of UAVs. To address the challenges that arise in building human-centric decision systems, it proposes a system development process in which a prototype is built which relies on a cognitive model and an adaptive agent, the system requirements are derived from the prototype and expressed as scenarios, a formal system model is synthesized from the scenarios, the model is validated and verified, and finally a system is implemented based on the formal model. Sections 3–5 describe the results of our research to support this development process in three areas—formal methods, cognitive science, and adaptive agents. To provide high assurance that a human-centric decision system behaves as intended, Section 3 introduces a new technique for synthesizing formal system models from scenarios. To predict when a human performing a complex set of tasks is overloaded, Section 4 reviews a cognitive model which predicts in real-time when a UAV operator is overloaded. To provide a practical and economical approach to selecting the best agent designs, Section 5 proposes a new technique for synthesizing user models from human participant studies data.

This article contains two new research contributions. The first, presented in Section 3, describes a new method for synthesizing a formal system model from scenarios. This method, which uses a novel representation of scenarios called a Moded Scenarios Description, consists of Event Sequence Charts (ESCs), a variant of Message Sequence Charts [25], and a Mode Diagram specifying sets of system modes and mode transitions. The link between the ESCs and the Mode Diagram is provided by *numeric labels*, integers which label the event sequences in the ESCs and the modes and transitions in the Mode Diagram. Section 3 presents an example which illustrates how critical behavior in a human-centric decision system may be expressed using a Moded Scenarios Description as well as a formal model of Moded Scenarios and algorithms for translating a Moded Scenarios Description into a formal state machine model of selected system requirements. The second contribution, presented in Section 5, describes new results in our research on the role of synthesized user models in designing adaptive agents. We present the results of a new pilot study investigating the optimal level of agent assistance for different users. Synthesized user models are used to evaluate an agent which has a parameter that effectively adjusts how proactive the agent is in assisting a user. We demonstrate that the optimal value for the parameter varies by user model; higher-performing user models had lower scores when they used overly proactive agents than when they used moderately proactive agents. Such an agent could be tightly integrated with the cognitive model by using the cognitive model's predictions to determine when to help a user.

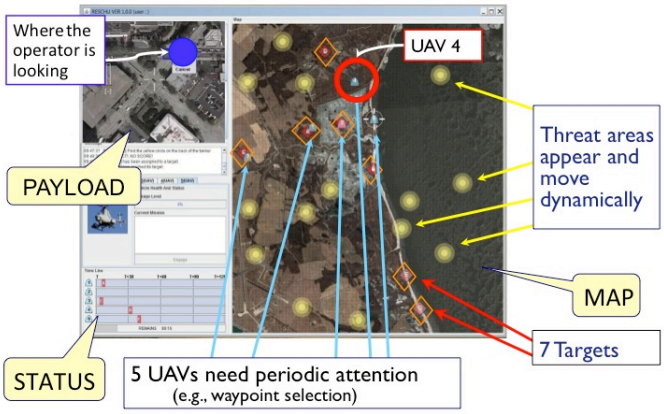

Fig. 1 Operator display of MIT's RESCHU simulator [6].

2 Human-Centric Decision System: Overview

2.1 Example of a Human-Centric Decision System: RESCHU

Our studies are being conducted in the context of the Research Environment for Supervisory Control of Heterogeneous Unmanned Vehicles (RESCHU) [6], a MIT-developed simulator of a decision system in which one human operator controls a team of UAVs. In RESCHU, operators assign and move UAVs to specific target areas, reroute UAVs to avoid threats, and order UAVs to engage targets. The operator can alter the path of a UAV towards its target by manipulating waypoints. Simultaneously, operators perform other tasks (e.g., visual acquisition, surveillance) in scenarios involving urban coastal or inland settings. RESCHU's operator interface, illustrated in Fig. 1, has three windows: The Map window displays UAVs, targets, and threats. The Status window provides information, such as UAV damage from threats, estimated time for UAVs to reach targets, etc. The Payload window displays status information for other mission tasks. As in other human-centric decision systems, a serious problem in RESCHU is *operator overload*—the operator has too many concurrent demands and is unable to handle all in a timely manner.

2.2 System Development Process

A promising approach to achieving high assurance for human-centric decision systems is Model-Based Development (MBD). In MBD, one or more models of the required system behavior are built, validated (e.g., via simulation) to capture the intended behavior, verified to satisfy required properties, and ultimately used to build the system implementation. Model properties to be verified include *completeness* (no missing cases), *consistency* (no non-determinism), and application properties, such as safety properties.

While the use of MBD in software practice is growing, a major problem is the lack of good formal requirements models. In many cases, system and software

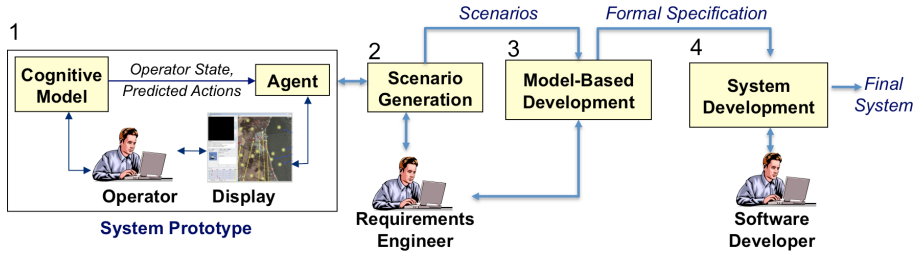

Fig. 2 Development process for a high assurance human-centric decision system.

requirements models do not exist at all. Even when they exist, these models are usually expressed ambiguously in languages without an explicit semantics *and* at a low level of abstraction. Ambiguity makes the models hard to analyze formally while the low level of abstraction leads to unneeded implementation bias and also makes the models hard to understand, validate, and change. To address these problems, researchers have introduced techniques (see, e.g., [39,38]) for synthesizing formal models from scenarios. Informally, scenarios describe how the system interacts with humans and the system environment to provide the required system services. Because many practitioners already use scenarios to elicit and define requirements, synthesizing formal models from scenarios is highly promising.

Fig. 2 shows a four-step process for developing high assurance human-centric decision systems, an extension of a process introduced in [5,23], which uses MBD. This is an idealization of the actual process which has more iteration and feedback and may not always proceed in a top-down fashion. In step 1, a prototype is built of the system's conceptual behavior. As shown in Fig. 2, a human operator of the system interacts with a visual display to perform a required set of tasks. Because the tasks are complex, an agent is available to assist an overloaded operator and a cognitive model to predict operator overload. In step 2, a requirements engineer elicits information about the system requirements from the prototype, determines the *system modes* (externally visible abstractions of the system state), and expresses the requirements as a set of scenarios and system modes. Informally, the system will behave differently in different modes; e.g., if the system is in mode A when a new input arrives, it may respond differently than it would in mode B. In the scenarios, implementation bias, e.g., in RESCHU, how the display represents an endangered UAV, can be avoided by using appropriate abstractions. The requirements engineer also formulates and expresses in precise natural language the required system properties. For RESCHU, an example property is "If endangered (i.e., too close to a threat), then a UAV is under operator control or agent control." In step 3, the scenarios and systems modes are automatically synthesized into a formal system model, and the model is checked for completeness and consistency and validated (e.g., via simulation) to capture the intended behavior. Moreover, required system properties, such as the example property above, are translated into logical formulae, and the formal model is verified to satisfy these properties, using an appropriate tool such as a model checker. Finally, in step 4, the model, along with information such as the platform characteristics, the characteristics and interfaces of I/O devices, etc., is the basis for developing source code, some generated automatically.

2.3 New Techniques: Overview

The following three sections describe new software engineering and AI techniques which support the system development process introduced in Section 2.2. In developing and evaluating these new techniques, we built a prototype system by extending RESCHU with a cognitive model to predict operator overload. In future work, an agent will be added to the prototype to assist the operator in performing the assigned tasks. We expect the technique described in Section 3 for synthesizing formal system models from scenarios to support steps 2–4 of our development process: Code generated from a validated, verified formal system model should provide high assurance that the system implementation satisfies its requirements. The techniques for evaluating agents and for predicting operator overload described in Sections 4 and 5 are expected to lead to good designs of human-centric decision systems and will therefore prove useful in developing the prototype system called for in step 1. Moreover, we expect the synthesized user models described in Section 5 will also be useful in step 2, e.g., for identifying assumptions about the operator's behavior, and in step 3 as input user data useful for validating and formally verifying the formal model synthesized in step 2.

3 Synthesis of Formal System Models from Scenarios

3.1 Background: SCR Tabular Notation and Toolset

In 1979-80, Heninger, Parnas, and other software engineering researchers proposed a tabular notation called SCR (Software Cost Reduction) for specifying software requirements [24,3]. SCR's tabular notation has two important benefits. First, software developers find models expressed in the format easy to understand. Second, the tabular notation scales; the large requirements models of practical systems can be concisely represented in tables. In [22], a formal state machine semantics for models expressed in the SCR tabular notation was presented. This semantics includes an important construct of SCR, *system modes* (also called *modes*). Modes provide a system-level abstraction for partitioning the system states into equivalence classes, one class per mode. An important feature of modes is that they are already explicit in many critical software systems (see, e.g., [3,18,1]). Based on the SCR formal semantics and the notion of modes, a large suite of tools called the SCR toolset has been developed. These tools include a consistency checker for detecting well-formedness errors (type errors, missing cases) in the model specification [22]; a simulator for symbolically executing the model to validate that it captures the intended behavior [17]; and an invariant generator for automatically generating state invariants from the model [26,29]. Also integrated into the toolset are model checkers for finding violations of desired model properties, such as safety and security properties [19], and theorem provers to verify such properties [27,21]. Tools have also been developed for automatically generating test cases [13] and for synthesizing source code from the model [30,33].

A serious problem is that, although they readily understand requirements models in the tabular notation, software developers have difficulty developing these and other formal requirements models. However, our experience is that, given a requirements model in the tabular notation, practitioners can understand, modify, and extend the model. The challenge is to produce the initial model. This section describes Moded Scenarios, our solution to this problem. A Moded Scenarios Description consists of Event Sequence Charts, which look like Message

Sequence Charts (MSCs), a popular approach among practitioners for specifying system requirements, and a Mode Diagram, which uses system modes to provide a system-level abstraction for combining the ESCs. This section presents an example showing how Moded Scenarios, i.e., ESCs, a Mode Diagram, and a Scenario Constraint, can be used specify selected system requirements for a hazard avoidance task. It also introduces a formal model that represents the information in ESCs and Mode Diagrams, a method for transforming a Moded Scenarios Description into a formal SCR requirements model, and algorithms for computing the update functions of the model.

3.2 Specifying Scenarios Using Event Sequence Charts and Mode Diagrams

To specify the requirements of a software system, many practitioners use scenarios. A popular notation for specifying scenarios is that of Message Sequence Charts (MSCs) [25]. Inspired by MSCs and their popularity among practitioners, we have developed a variant of MSCs called Event Sequence Charts (ESCs), which have a natural state machine semantics, are easy to change, and are designed to scale better than the traditional basic and hierarchical MSCs. Each ESC contains a set of *entities* and a list of *event sequences*. The entities include the system and a set of environmental entities, the latter consisting of *monitored entities*—entities which the system monitors—and *controlled entities*—entities which the system controls. Each monitored entity is associated with one or more monitored variables and each controlled entity with one or more controlled variables. Each event sequence contains a single *monitored event*, a change in value of a monitored variable, followed by a set, possibly empty, of changes in the values of controlled variables. A monitored event may also cause changes in the values of *term variables*, auxiliary variables designed to make the ESCs more concise and more understandable.

Our approach uses ESCs, a Mode Diagram, and a Scenario Constraint to specify system requirements. A Mode Diagram contains sets of modes and mode transitions. To specify the relationship between ESCs and a Mode Diagram, our approach uses *numeric labels*, positive integers which label each event sequence in an ESC and the modes and mode transitions in the Mode Diagram. To illustrate our approach, we present two ESCs, a Mode Diagram, and a simple Scenario Constraint which specify (some of) the required system behavior in a hazard avoidance task. In RESCHU, a UAV's path may cross a hazard area. For an example, see the top of Fig. 1, where UAV 4 is dangerously close to a hazard. To avoid the hazard, either the operator or an agent assisting the operator must modify the UAV's path.

3.2.1 Two Examples of Event Sequence Charts

Though inspired by MSCs [25], MSCs and ESCs have significant differences. While MSCs have been used to specify both system requirements *and* designs, the purpose of ESCs is to specify system and software requirements only. Unlike MSCs which often describe the interactions of many system components, each ESC has only a single system entity and many environmental entities. Further, an event sequence in an ESC includes not only a monitored event (change in a monitored entity) but all of that event's effects, captured in changes to controlled and term variables. Thus a single event in an ESC usually corresponds to a sequence of two or more messages in a MSC. While visually similar, ESCs and MSCs are also semantically different. In an ESC, an event and its effects occur in a single step. In an MSC, an event and its effects occur sequentially in several steps.

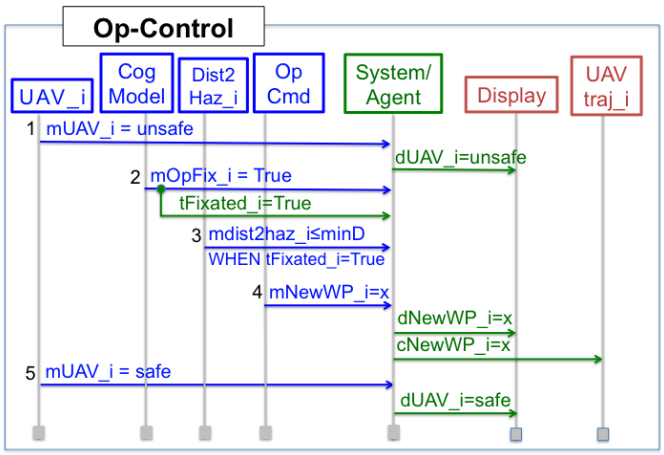

Fig. 3 Event sequence chart **Op-Control** representing a scenario which specifies the required system behavior when the operator adds a waypoint to a UAV's path to avoid a hazard.

Figs. 3 and 4 contain two ESCs, which together specify the system's required behavior in the hazard avoidance task. The scenario in Fig. 3, called **Op-Control**, describes the situation when the operator is in charge. It contains five event sequences, each assigned a numeric label and interpreted as follows:

1. Upon learning that a UAV is unsafe, the system modifies the display to warn the operator (e.g., by changing the color of the icon which represents the UAV) that the UAV needs attention.
2. The cognitive model learns, e.g., from eye tracker data, that the operator is *fixated* on (paying attention to) the UAV and notifies the system. In response, the system sets the term variable `tFixated_i` to `True`.
3. The system is notified that the UAV is in danger, i.e., the distance between the UAV's location and the hazard is at or below a threshold `minD`. Because the operator is fixated on the UAV, the system relies on the operator to act to protect the UAV.
4. The operator takes action, i.e., instructs the system to add a waypoint to the UAV's path to avoid the hazard. In response, the system updates the display to show the UAV's new path once the waypoint is added.
5. The system is notified that the UAV is safe. In response, it updates the display to show that the UAV is no longer in danger.

The scenario **Agent-Control** (see Fig. 4) specifies the system requirements when the operator is busy with other tasks, and the agent must act to protect the UAV. The ESC specifying this scenario is interpreted as follows:

1. Same behavior as in the Op-Control scenario.
6. The cognitive model informs the system that the operator is not fixated on the endangered UAV, causing the system to set `tFixated_i` to `False`.
7. As in the first scenario, the system is notified that the distance between the UAV and the hazard is at or below `minD`. However, because the operator is not fixated on the UAV, the agent acts to protect the UAV by adding a waypoint to its path. The system then updates the display to show the UAV's new path.
8. Same behavior as event sequence 5 in the Op-Control scenario.

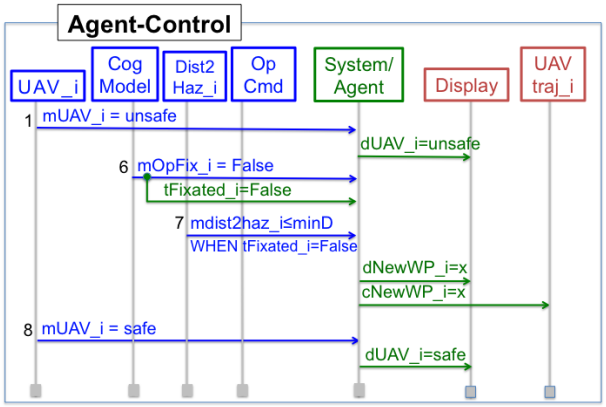

Fig. 4 Event sequence chart **Op-Control** representing a scenario which specifies the required system behavior when the agent adds a waypoint to a UAV's path to avoid a hazard.

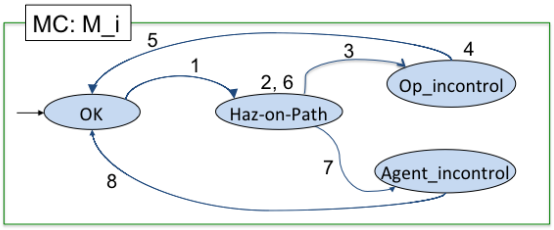

Fig. 5 Mode Diagram showing the four system modes in a mode class M_i, the allowed mode transitions, and numbers of the monitored events which trigger them.

3.2.2 Example of a Mode Diagram

The mode diagram in Fig. 5 describes a *mode class* called `M_i`; four modes, possible values of `M_i`; and five mode transitions. This diagram, together with the ESCs in Figs. 3 and 4, and the Scenario Constraint C (see below), specifies the required system behavior in the hazard avoidance task. As shown in Fig. 5, in a Mode Diagram, both transitions and modes have *numeric labels*. Each transition is required to have a single numeric label. This label identifies the event sequence containing the monitored event which triggered the transition. For example, in Fig. 5, the transition labeled 1 from mode `OK` to mode `Haz-on-Path` is triggered by the unconditioned monitored event "`UAV_i` = `unsafe`" in event sequence 1. This event sequence appears in the ESCs shown in both Figs. 3 and 4. In Fig. 5, the transition labeled 7 from mode `Haz-on-Path` to mode `Agent_incontrol` is triggered by the conditioned monitored event in event sequence 7 in Fig. 4, namely, "`mdist2haz` $\leq$ `minD WHEN tFixated_i` = `False`." In a Mode Diagram, each mode has an associated set, perhaps empty, of numeric labels. These numbers, separated by commas, indicate that one or more monitored events may occur in the mode but that none triggers a mode transition. For example, in Fig. 5, the "2, 6" label on the mode `Haz-on-Path` refers to the corresponding monitored events in event sequences 2 and 6 in Figs. 3 and 4, respectively.

3.2.3 Scenario Constraint: Examples

A Scenario Constraint can either restrict or add to the required system behavior specified in the ESCs and the Mode Diagram. For the hazard avoidance task, an example of a Scenario Constraint that restricts the system behavior is [`mOpFix_i` = `True` ⇔ `tFixated_i` = `True`] `AND` [`INIT`(`mOpFix_i` = `False`)], where `INIT` is a predicate that defines the initial value of a monitored, controlled, or term variable specified in one or more of the ESCs. This constraint states that the term variable `tFixated_i` always has the same value as the monitored variable `mOpFix_i` and that the initial value of the monitored variable `mOpFix_i` is `False`. An example of a Scenario Constraint which adds required system behavior for the hazard avoidance task is `M_i` = `Op_incontrol` `AND` `mOpFix_i` = `True` `AND` `mOpFix_i`′ = `False` → `M_i`′ = `Agent_incontrol`, where an unprimed variable represents the current value of a variable, and a primed variable represents a variable's value when a new monitored event occurs. This constraint states that if the operator stops paying attention when the UAV is close to a hazard, the system switches responsibility for moving the UAV to the adaptive agent. This example shows how a constraint may change the behavior specified in both the ESCs and the Mode Diagram.

3.3 Formal Model of Moded Scenarios Description

This section introduces the notions of entities, types, scenario state variables, events, and conditions. These are used to define ESCs, a Mode Diagram, a scenario state, and a Scenario Constraint. The ESCs, Mode Diagram, and Scenario Constraint are used in turn to define a *Moded Scenarios Description*. To link together the Event Sequence Charts and the Mode Diagram, we assume a set of *numeric labels*, positive integers which label event sequences, modes, and mode transitions. Appendix A contains a list of the notation used in this section and Section 3.4.

3.3.1 Event Sequence Charts (ESCs)

We assume the existence of the following sets.

- X is a set of entities consisting of the system and a number of environmental entities. X is partitioned into the set X_M of monitored entities, the set X_C of controlled entities, and the singleton set X_S containing the system.
- TS is a union of types $\mathcal{T}$, where each type is a nonempty set of values. The set of positive integers $\mathcal{T}_{Z^+} = \{1, 2, \ldots\}$ and the set $\mathcal{T}_B = \{\texttt{True}, \texttt{False}\}$ of Boolean truth values are examples of types.
- RF^{sc} is a set of *scenario state variables*. RF^{sc} is partitioned into a set RF_M of monitored variables, a set RF_T of terms, and a set RF_C of controlled variables. A function $\mathrm{V}^{sc} : \mathrm{RF}^{sc} \rightarrow$ TS maps each variable $r \in \mathrm{RF}^{sc}$ to a value in its type set $\mathrm{TY}^{sc}(r) \subset$ TS. Another function $F_X : \mathrm{RF}_M \cup \mathrm{RF}_C \rightarrow \mathrm{X}$ maps each monitored or controlled variable r to an associated entity x in X. Thus if $F_X(r)$ is in X_M, then r is a monitored variable; if $F_X(r)$ is in X_C, then r is a controlled variable.

  Example: In Figs. 3 and 4, the set of entities is X = {`UAV_i`, `CogModel`, `Dist2Haz_i`, `OpCmd`, `System/Agent`, `Display`, `UAVtraj_i`}, the set of monitored entities is X_M = {`UAV_i`, `OpCmd`, `CogModel`, `Dist2Haz_i`}, the set of controlled entities is X_C = {`Display`, `UAVtraj_i`}, and

the singleton set is $X_S = \{\texttt{System/Agent}\}$. The set of scenario state variables is $RF^{sc} = \{\texttt{mUAV\_i}, \texttt{mOpFix\_i}, \texttt{mdist2haz\_i}, \texttt{mNewWP\_i}, \texttt{tFixated\_i}, \texttt{dNewWP\_i}, \texttt{cNewWP\_i}, \texttt{dUAV\_i}\}$, the set of monitored variables is $RF_M = \{\texttt{mUAV\_i}, \texttt{mOpFix\_i}, \texttt{mdist2haz\_i}, \texttt{mNewWP\_i}\}$, the set of term variables is $RF_G = \{\texttt{tFixated\_i}\}$, and the set of controlled variables is $RF_C = \{\texttt{dNewWP\_i}, \texttt{cNewWP\_i}, \texttt{dUAV\_i}\}$. $V^{sc}(\texttt{dUAV\_i})$ is in $\{\texttt{safe}, \texttt{unsafe}\}$, and $V^{sc}(\texttt{mOpFix\_i})$ is in $\{\texttt{True}, \texttt{False}\}$.

Events and Conditions. Events and simple conditions are triples of the form (r, o, v) where r is a scenario state variable in RF^{sc}, o is a relational operator $(=, >, ...)$, and v is a value in TS. A monitored event $m = (r, o, v)$ is an event in which r is in RF_M; a term event $t = (r, o, v)$ is an event in which r is in RF_T, and a controlled event $c = (r, o, v)$ is an event in which r is in RF_C. In controlled events, o has the value "=".

Example: In event sequence 1 in Fig. 3, the triple $m = (r, o, v)$, where $r = \texttt{mUAV\_i}$, o has the value "=", and $v = \texttt{unsafe}$, represents the monitored event "`mUAV_i` is `unsafe`." In the `WHEN` clause of event sequence 3 in Fig. 3, the triple $d = (r, o, v)$, where $r = \texttt{tFixated\_i}$, o has value "=", and $v = \texttt{True}$, represents the simple condition "`tFixated_i` is `True`."

A condition d is either `True`, a simple condition, or an expression containing two or more simple conditions linked together by the logical operators $\wedge$ and $\vee$ in the standard way. An unconditioned event is an event with no associated condition (besides `True`). A conditioned event is an event associated with a condition other than `True`.

Example: In Fig. 3, the monitored event in event sequence 1, "`mUAV_i` = `unsafe`," is an unconditioned monitored event, while the monitored event in event sequence 3, "`mdist2haz_i` $\leq$ `minD WHEN tFixated_i` = `True`," is a conditioned monitored event.

Event Sequence Chart (ESC) and Chart Labels. Given these sets and definitions, we define an ESC Y and an associated set K_Y of numeric labels called *chart labels*.

- An Event Sequence Chart Y is a set of event sequences $\{y_1, y_2, \ldots, y_n\}$. Each event sequence y_i in Y is a triple (i, A_i, B_i), where i in $\mathcal{T}_{Z^+}$ is the *numeric label* of the sequence; A_i is a *monitored event expression*; and B_i is a set of controlled events. The monitored event expression A_i is represented as a triple (m_i, d_i, G_i), where m_i is a monitored event, d_i is a condition, and G_i is a set of term events triggered by m_i when d_i holds. The sets G_i and B_i may be empty. In two related event sequence charts Y_1 and Y_2, y_i in Y_1 and y_j in Y_2, $i = j$, must imply $A_i = A_j$ and $B_i = B_j$. That is, if two event sequences in two related ESCs have the same numeric label, their monitored events and sets of controlled and term events must be identical.
- Associated with each ESC Y is a set K_Y of *chart labels*. The set $K_Y \subset I^+$ is defined by $K_Y = \{\ i \mid y_i = (i, A_i, B_i) \text{ in } Y\}$.

Example: In Fig. 3, event sequence 1 is represented as $y_1 = \{1, A_1, B_1\}$ with $A_1 = (m_1, d_1, G_1)$, $m_1 = (\texttt{mUAV\_i}, =, \texttt{unsafe})$, $d_1 = \texttt{True}$, $G_1 = \emptyset$, and $B_1 = \{(\texttt{dUAV\_i}, =, \texttt{unsafe})\}$. Similarly, in event sequences $y_2 = \{2, A_2, B_2\}$ and $y_3 = \{3, A_3, B_3\}$, $A_2 = (m_2, d_2, G_2)$, where $m_2 = (\texttt{mOpFix\_i}, =, \texttt{True})$, $d_2 = \texttt{True}$, $G_2 = \{(\texttt{tFixated\_i}, =, \texttt{True})\}$, and $B_2 = \emptyset$; and $A_3 = (m_3, d_3, G_3)$, where $m_3 = (\texttt{mdist2haz\_i}, \leq, \texttt{minD})$, $d_3 = \{(\texttt{tFixated\_i}, =, \texttt{True})\}$, and $G_3 = B_3 = \emptyset$. For the ESC in Fig. 4, the set of chart labels is $\{1, 6, 7, 8\}$.

3.3.2 Mode Diagram

A Mode Diagram describes a simple state machine called a "mode machine" by defining a set of *system modes* (also called simply *modes*), an initial mode, and a set of mode transitions. Each transition has exactly one numeric label. Each mode has an associated set, perhaps empty, of numeric labels. These labels correspond to the numeric labels of the event sequences in the ESCs that can occur while remaining in the mode. A Mode Diagram MD and its associated set of numeric labels K_{MD} are formally defined as follows.

- A Mode Diagram is a 5-tuple MD = $(\mathcal{M}, \mathcal{M}_N, \mu_0, \mathcal{M}_T, \mathcal{M}_D)$, where $\mathcal{M}$ is a set of modes; $\mathcal{M}_N$ is the *mode class*, a variable with type set $\mathcal{M}$ that names the current mode; $\mu_0 \in \mathcal{M}$ is the initial mode; $\mathcal{M}_T$ is a set of mode transitions, each transition labeled with a unique numeric label; and $\mathcal{M}_D$ is a set which associates each mode with a set of numeric labels. The set $\mathcal{M}_T$ of mode transitions is represented as a set of triples (μ_i, k, μ_j), where μ_i and μ_j, $i \neq j$, are modes in $\mathcal{M}$, and k is the *numeric label* of the transition. The set $\mathcal{M}_D$ is represented as a set of ordered pairs (μ_j, H_j), where, for all modes μ_j in $\mathcal{M}$, H_j is the set, possibly empty, of numeric labels associated with mode μ_j. The set $\mathcal{M}_T$ contains information needed to define a mode transition function. The set $\mathcal{M}_D$ contains added information needed to define the values of controlled variables in terms of modes.
- Associated with a Mode Diagram MD=$(\mathcal{M}, \mathcal{M}_N, \mu_0, \mathcal{M}_T, \mathcal{M}_D)$ is a set of numeric labels called *diagram labels* K_{MD}, where $K_{MD} \subset I^+$ is the union $K_{MD} = K_T \cup K_M$ of the set K_T of *transition labels* and the set K_M of *mode labels*. The set K_T is defined by $K_T = \{\ k \mid (\mu_i, k, \mu_j) \in \mathcal{M}_T\}$. The set K_M is defined by $K_M = \{\ h \mid (\mu, H) \in \mathcal{M}_D, h \in H\}$. The sets of transition labels and mode labels cannot overlap; i.e., $K_T \cap K_M = \emptyset$.

  Example: In Fig. 5, the set of modes is {`OK`, `Haz-on-Path`, `Op_incontrol`, `Agent_incontrol`}, the mode class $\mathcal{M}_N$ that names the current mode is `M_i`, and the initial mode μ_0 is `OK`. The set $\mathcal{M}_T$ of mode transitions contains five triples, one for each transition in Fig. 5. The transition from `OK` to `Haz-on-Path` is represented by the triple (`OK`, 1, `Haz-on-Path`), where 1 is the transition label. The set $\mathcal{M}_D$ of mode-label pairs is $\mathcal{M}_D = \{$(`OK`, $\emptyset$), (`Haz-on-Path`, $\{2, 6\}$), (`Op_incontrol`, $\{4\}$), (`Agent_incontrol`, $\emptyset$)$\}$. The set of transition labels is $K_T = \{1, 3, 5, 7, 8\}$, the set of mode labels is $K_M = \{2, 4, 6\}$, and the set of diagram labels is $K_{MD} = \{1, 2, \ldots, 8\}$. Note that $K_T \cap K_M = \emptyset$.

Scenario State. Suppose that $\mathcal{Y} = \{Y_1, Y_2, \ldots, Y_m\}$ is a set of ESCs and MD is a Mode Diagram MD = $(\mathcal{M}, \mathcal{M}_N, \mu_0, \mathcal{M}_T, \mathcal{M}_D)$. A *scenario state* $\hat{s}$ is a function $\hat{s} : \mathrm{RF}^{sc} \cup \{\mathcal{M}_N\} \rightarrow \mathrm{TS} \cup \mathcal{M}$ that maps each scenario state variable in RF^{sc} to a value in its type set $\mathcal{T} \subset \mathrm{TS}$ and the single mode class $\mathcal{M}_N$ to a mode in $\mathcal{M}$. More precisely, if r is a variable, for all $r \in \mathrm{RF}^{\mathrm{sc}} : \hat{\mathrm{s}}(\mathrm{r}) \in \mathrm{TY}^{\mathrm{sc}}(\mathrm{r})$, and for $r = \mathcal{M}_N : \hat{s}(r) \in \mathcal{M}$. $\mathrm{TY}^{sc}(r) = \mathcal{T}$, where $\mathcal{T} \subset \mathrm{TS}$ is the subset of TS whose members are type-correct values for r. Thus a scenario state is uniquely determined by an assignment of type-correct values to every variable $r \in \mathrm{RF}^{\mathrm{sc}} \cup \{\mathcal{M}_N\}$.

3.3.3 Scenario Constraint

A *Scenario Constraint* is a conjunction of one-state and two-state properties, which are defined, respectively, by predicates on single scenario states and on pairs of scenario states [19]. Because each scenario state is represented as a mapping from a

set of variables—the scenario state variables and the mode class—to values, useful predicates can typically be defined by formulas over the variables of one or two scenario states.[1]

3.3.4 Moded Scenarios

A Moded Scenarios description $\mathcal{W}$ is defined by a triple $\mathcal{W} = (\mathcal{Y}, \mathrm{MD}, \mathrm{C})$, where $\mathcal{Y}$ is a set of event sequence charts, MD is a Mode Diagram, and C is a Scenario Constraint. We define the set $K_{\mathcal{Y}}$ of *ESC labels* of $\mathcal{Y}$ as the union of the sets of chart labels associated with the event sequence charts in $\mathcal{Y}$, that is, $K_{\mathcal{Y}} = \cup_{Y \in \mathcal{Y}} K_Y$. If K_{MD} is the set of diagram labels associated with the mode diagram MD, then we require $K_{\mathcal{Y}} = K_{MD}$, that is, every numeric label associated with an event sequence in an ESC has a corresponding numeric label associated with a mode or mode transition in a Mode Diagram, and vice versa.

Example: In Figs. 3–4, the set $K_{\mathcal{Y}}$ of ESC labels is $\{1,2,3,4,5\} \cup \{1,6,7,8\} = \{1,2,\ldots,8\}$. In Figs. 3–5, $K_{\mathcal{Y}} = K_{MD} = \{1,2,\ldots,8\}$ as required.

3.4 Synthesizing a Formal Model from Moded Scenarios

The objective is to transform a Moded Scenarios description of the form $\mathcal{W} = (\mathcal{Y}, \mathrm{MD}, \mathrm{C})$, where $\mathcal{Y}$ is a set of ESCs, MD is a Mode Diagram, and C is a Scenario Constraint, into a formal state machine model. The state machine model is represented as a system $\Sigma = (S, \Theta, \rho)$, where S is a set of states, Θ is an initial state predicate, and ρ is a transform function specifying how the current state and a monitored event are mapped to a new state. Our method has five steps:

1. *Construct the set of state variables and the set of values.* Using information in the ESCs and the Mode Diagram, construct the new set VS of values and the new set of state variables RF. Based on these sets, define a new function V which maps a state variable to a value. These two sets and the function V provide the basis for constructing the system state, conditions and events, and the set of states S in system Σ using the definitions in [22].
2. *Construct the initial state predicate.* Based on the definition of the initial mode in the Mode Diagram and information about the initial state in the Constraint, define Θ, the initial state predicate.
3. *Construct the system transform.* Based on information in the Mode Diagram and the ESCs, construct the transform function ρ. The function ρ is computed using a set of update functions which specify how the values of the dependent variables—the mode class, the controlled variables, and the term variables—change in response to a monitored event.
4. *Simplify and extend the model.* Based on the Scenario Constraint C, modify the update functions.
5. *Analyze and validate the model.* Apply analysis techniques and tools to detect defects in the specification of the model, such as syntax errors and nondeterminism. Once such defects have been removed, other tools can be applied, such as simulators and verifiers to further improve the quality of the model.

[1] An example of a predicate that may be impossible to capture in a formula is the reachability predicate where there are infinitely many possible scenario states.

3.4.1 Construct the Sets of State Variables and Values

From the ESCs $\mathcal{Y}$ and the Mode Diagram MD, we construct the following sets.

- VS = TS $\cup \mathcal{M}$ is the set of values. VS is the union of the set TS of values defined in the ESCs and the set $\mathcal{M}$ of modes defined in the Mode Diagram MD.
- RF = $\text{RF}^{sc} \cup \{\mathcal{M}_N\}$ is the set of state variables. Thus RF is the union of the set RF^{sc} of scenario state variables—the monitored, controlled, and term variables—defined in the ESCs in $\mathcal{Y}$ and the singleton set containing the mode class $\mathcal{M}_N$. A new function V : RF $\rightarrow \mathcal{T} \subset$ VS extends the function V^{sc}. This function V maps each variable in RF to a value in the type set $\mathcal{T}$. If the state variable r is a mode class $\mathcal{M}_N$, then V maps r to a mode in $\mathcal{M}$.

  Example: For the example presented in Figs. 3–5, V(`dUAV_i`) is in {`safe`, `unsafe`}, and V(`M_i`) is in {`OK`, `Haz-on-Path`, `Op_incontrol`, `Agent_incontrol`}.

Using the sets VS of values and RF of state variables, and function V, we define the system state, conditions, events, and the system Σ using the definitions in [22].

System State. A *system state* s is a function that maps each state variable r in RF to a value in its type set. More precisely, for all $r \in$ RF : s(r) $\in$ TY(r), where $\text{TY}(r) = \mathcal{T}$ is the subset of VS whose members are type-correct values for r.

Conditions. A *condition* is a single-state predicate, a function whose range type is Boolean. A *simple condition* is $true$, $false$, or of the form $\gamma_1 \odot \gamma_2$, where $\odot$ is a relational operator, and γ_1 and γ_2 are any well-formed expressions composed of constants, state variables, and single-state functions in the standard way.

Events. Denoted by "@T", *events* are two-state predicates that describe a change in the value of at least one variable. A *monitored event*, denoted @T($r = v$), describes the change in monitored variable r in RF_M to value v in TY(r). A *basic event* is denoted @T(c), where c is a condition. A *simple conditioned event* is of the form @T(c) WHEN d, where @T(c) is a basic event and d is a simple condition or a conjunction of simple conditions. It is defined by

$$@\text{T}(c) \text{ WHEN } d \stackrel{\text{def}}{=} \neg c \wedge c' \wedge d, \tag{1}$$

where the unprimed version of c denotes c in the old state, the primed version denotes c in the new state, and the condition d is evaluated in the old state.

System. A system Σ is a triple $\Sigma = (S, \Theta, \rho)$, where S is a set of states, $\Theta \subseteq S$ is the set of initial states, and $\rho \subset S \times S$ is a set of state transitions.

3.4.2 Construct the Initial State Predicate

Suppose the Mode Diagram MD is defined by MD = $(\mathcal{M}, \mathcal{M}_N, \mu_0, \mathcal{M}_T, \mathcal{M}_D)$ and the Scenario Constraint C by $C = c_1 \wedge c_2 \wedge \ldots \wedge c_m$, where each c_i is a one-state or two-state predicate. Suppose further that $\Sigma = (S, \Theta, \rho)$. Let C^I be the set of predicates that assign an initial value to a variable, i.e., $C^I = \{c_i \mid c_i \texttt{ is a conjunct in } C, c_i = \texttt{INIT}(r = v),\ r \in \text{RF}, v \in \text{VS}\}$, and let $\text{RV}^I = \{(r, v) \mid \texttt{INIT}(r = v) \in C^I\}$. If $s \in S$ is a system state, then $\Theta(s)$ is defined by

$$\Theta(s) = \texttt{True} \Leftrightarrow \forall r \in \mathrm{RF} : [\ r = \mathcal{M}_N\ \wedge\ s(r) = \mu_0]\ \vee\ [(r, s(r)) \in \mathrm{RV}^I]\ \vee \\ [\ r \neq \mathcal{M}_N\ \wedge\ \nexists\ v : (r, v) \in \mathrm{RV}^I]$$

Thus Θ assigns the initial value μ_0 to the mode class $\mathcal{M}_N$, and the initial value $s(r)$ to r if the Constraint C contains $\texttt{INIT}(r, s(r))$. All variables other than the mode class which are not assigned initial values by C may have any initial value.

3.4.3 Construct the Transform Function

The transform function ρ is defined by the composition of a set of update functions. There is an update function Υ_r for each dependent (i.e., non-monitored) variable $r \in RF$, specifying how the value of r changes from current state s to the new state s'. To make the definition of the function complete, we require that for all variables r, if r' is not explicitly assigned a value in the new state, then $r' = r$, i.e., in the new state, the value of r does not change.

The update function Υ_r can be represented in the following form:

$$(*) \quad r' = \begin{cases} v_1 \text{ if } E_1 \\ \ldots \\ v_n \text{ if } E_n \\ r \quad \text{otherwise} \end{cases}$$

where each E_i is an expression defined on the variables in states s and s'. We can assume the v_i are distinct because if $v_i = v_j$ for $i \neq j$ then we could replace the two cases "v_i if E_i" and "v_j if E_j" by the single case "v_i if $E_i \vee E_j$".

This section presents two algorithms, Algorithm 1 and Algorithm 2, which can be used to generate the update functions for mode classes, terms, and controlled variables from the information contained in a Moded Scenarios Description.

Update functions for mode classes. Algorithm 1 computes the update function $\Upsilon_{\mathcal{M}_N}$ for the mode class $\mathcal{M}_N$ using information in the Moded Scenarios $\mathcal{W} = (\mathcal{Y}, \mathrm{MD}, \mathrm{C})$ with $\mathrm{MD} = (\mathcal{M}, \mathcal{M}_\mathrm{N}, \mu_0, \mathcal{M}_\mathrm{T}, \mathcal{M}_\mathrm{D})$. As preconditions, the algorithm requires 1) the set $K_\mathcal{Y}$ of chart labels associated with the set $\mathcal{Y}$ of ESCs to be identical to the set K_{MD} of diagram labels associated with the Mode Diagram MD and 2) if two ESCs in $\mathcal{Y}$ have event sequences with the same chart label, the event sequences must be identical. The algorithm produces a set of 4-tuples (μ_i, m_k, d_k, μ_j), where μ_i is a mode, m_k is a basic monitored event, d_k is a simple condition, and μ_j is a mode. The interpretation of the 4-tuple is that if basic event m_k occurs when the system is in mode μ_i and condition d_k is true, then the new mode is μ_j, i.e., $\mathcal{M}'_N = \mu_j$. To begin, the algorithm initializes $\Upsilon_{\mathcal{M}_N}$ (line 1). For each mode transition (line 2), the algorithm checks each event sequence in each ESC (line 3) for a chart label that matches the transition label k (line 4). If it finds a matching label, the algorithm adds to $\Upsilon_{\mathcal{M}_N}$ the transition from mode μ_i to mode μ_j conditioned on the event m_k occurring when d_k holds (line 5).

Because event sequences in related ESCs with the same numeric label are required to have identical monitored, controlled, and term events, Algorithm 1 can be implemented more efficiently. The `for` loop on lines 3-5 can exit once an event

Algorithm 1: Generating the Update Function for a Mode Class

INPUT: Moded Scenarios $\mathcal{W} = (\mathcal{Y}, \text{MD} = (\mathcal{M}, \mathcal{M}_\text{N}, \mu_0, \mathcal{M}_\text{T}, \mathcal{M}_\text{D}), \text{C})$
OUTPUT: Mode Transition Function $\Upsilon_{\mathcal{M}_N}$
PRECONDITION: $(K_\mathcal{Y} = K_{MD}) \ \wedge \ (K_T \cup K_M = \emptyset)$
PRECONDITION: $\forall Y_i, Y_j \in \mathcal{Y} \ ((y_i = (i, A_i, B_i) \in Y_i \wedge y_j = (j, A_j, B_j) \in Y_j \wedge i = j) \implies (A_i = Aj \wedge B_i = B_j))$

$\Upsilon_{\mathcal{M}_N} = \{\}$;
foreach $(\mu_i, k, \mu_j) \in \mathcal{M}_T$ **do**
  foreach $Y \in \mathcal{Y}$ **do**
    if $(k, A_k = (m_k, d_k, G_k), B_k) \in Y$ **then**
      $\Upsilon_{\mathcal{M}_N} = \Upsilon_{\mathcal{M}_N} \cup \{(\mu_i, m_k, d_k, \mu_j)\}$;

sequence with diagram label k is found rather than finding all event sequences with transition label k in all $Y \in \mathcal{Y}$.

Example: Applying Algorithm 1 to the Mode Diagram for mode class `M_i` in Fig. 5 and the ESCs in Figs. 3–4 generates the following five 4-tuples for `M_i`.

$$
\begin{aligned}
\Upsilon_{\texttt{M\_i}} = \{ & (\ \texttt{OK}, (\texttt{mUAV\_i}, =, \texttt{unsafe}), \texttt{True}, \texttt{Haz-on-Path}), \\
& (\ \texttt{Haz-on-Path}, (\texttt{mdist2haz\_i}, \leq, \texttt{minD}), (\texttt{tFixated\_i}, =, \texttt{True}), \texttt{Op\_incontrol}), \\
& (\ \texttt{Haz-on-Path}, (\texttt{mdist2haz\_i}, \leq, \texttt{minD}), (\texttt{tFixated\_i}, =, \texttt{False}), \texttt{Agent\_incontrol}), \\
& (\ \texttt{Op\_incontrol}, (\texttt{mUAV\_i}, =, \texttt{safe}), \texttt{True}, \texttt{OK}), \\
& (\ \texttt{Agent\_incontrol}, (\texttt{mUAV\_i}, =, \texttt{safe}), \texttt{True}, \texttt{OK})\}.
\end{aligned}
$$

From the set of 4-tuples, the update function can be computed and expressed in the form of $(*)$ as shown below or equivalently in a special tabular format (see Table 1) called in SCR a mode transition table [22]. Table 1 defines a two-state function describing how the system mode changes as a function of the current mode and a new monitored event.

$$
\texttt{M\_i}' = \begin{cases}
\texttt{Haz-on-Path} & \text{if } \texttt{M\_i} = \texttt{OK} \wedge \texttt{@T(mUAV\_i = unsafe)} \\
\texttt{Op\_incontrol} & \text{if } \texttt{M\_i} = \texttt{Haz-on-Path} \wedge \texttt{@T(mdist2haz\_i} \leq \texttt{minD)} \wedge \\
& \quad \texttt{tFixated\_i = True} \\
\texttt{Agent\_incontrol} & \text{if } \texttt{M\_i} = \texttt{Haz-on-Path} \wedge \texttt{@T(mdist2haz\_i} \leq \texttt{minD)} \wedge \\
& \quad \texttt{tFixated\_i = False} \\
\texttt{OK} & \text{if } \texttt{M\_i} \in \{\texttt{Op\_incontrol, Agent\_incontrol}\} \wedge \\
& \quad \texttt{@T(mUAV\_i = safe)} \\
\texttt{M\_i} & \text{otherwise}
\end{cases}
$$

Old Mode M_i	Event	New Mode M_i′
OK	@T(mUAV_i = unsafe)	Haz-on-Path
Haz-on-Path	@T(mdist2haz_i ≤ minD) WHEN tFixated_i = True	Op_incontrol
Haz-on-Path	@T(mdist2haz_i ≤ minD) WHEN tFixated_i = False	Agent_incontrol
Op_incontrol, Agent_incontrol	@T(mUAV_i = safe)	OK

Table 1 Mode Transition Table for the Human-Centric Decision System

Update functions for term and controlled variables. Algorithm 2 generates the update functions for the term variables and controlled variables. The inputs to the algorithm are the Moded Scenarios Description $\mathcal{W} = (\mathcal{Y}, \text{MD} = (\mathcal{M}, \mathcal{M}_\text{N}, \mu_0, \mathcal{M}_\text{T}, \mathcal{M}_\text{D}), \text{C})$, the set of controlled variables RF_C, and the set of term variables RF_G. As in Algorithm 1, preconditions are that the set $K_\mathcal{Y}$ of chart labels associated with the ESCs in $\mathcal{Y}$ are identical to the set of diagram labels associated with the Mode Diagram MD and that if two event sequence charts in $\mathcal{Y}$ share a chart label, the label must be attached to identical event sequences. For each variable r in $RF_C \cup RF_G$ the algorithm produces a set of 4-tuples (m_k, μ_k, d_k, v), each representing the assignment $r = v$ that occurs if event m_k occurs when $\mathcal{M}_N = \mu_k \wedge d_k$ holds. The algorithm begins by initializing the $\Upsilon_\mathtt{a}$ (lines 1 - 2). For each transition $(\mu_i, k, \mu_j) \in \mathcal{M}_T$, the algorithm checks each event sequence chart Y and if the event sequence indexed by k occurs in Y (line 5), then for each controlled event $(r, o, v) \in B_k$ and each term event $(r, o, v) \in G_k$ the algorithm adds to r's update function an assignment $r = v$ conditioned on the monitored event m_k occurring and the condition $\mathcal{M}_N = \mu_i \wedge d_k$ holding (lines 7 and 9). For each ordered pair $(\mu_i, H) \in \mathcal{M}_D$, the algorithm checks whether each member k in set H is in each ESC Y. If so, then for each controlled event $(r, o, v) \in B_k$ and each term event $(r, o, v) \in G_k$, the algorithm adds to r's update function an assignment $r = v$ conditioned on the monitored event m_k occurring and the condition $\mathcal{M}_N = \mu_i \wedge d_k$ holding (lines15 and 17). Like Algorithm 1, the `for` loops on lines 4-9 and 12-17 can be optimized to exit once an event sequence with label k is found rather than finding all event sequences with label k in all $Y \in \mathcal{Y}$.

Example: Applying Algorithm 2 to the ESCs and Mode Diagram in Figs. 3–5 generates four sets of 4-tuples, one for each dependent variable: the term variable `tFixated_i` and the three controlled variables `dUAV_i`, `dNewWP_i`, and `cNewWP_i`:

- $\Upsilon_{\texttt{tFixated\_i}} = \{$ (`mOpFix_i,=,true`),`Haz-on-Path` ,`True`, `True`), (`(mOpFix_i,=,false)`,`Haz-on-Path`, `True`, `False`) $\}$
- $\Upsilon_{\texttt{dUAV\_i}} = \{$ (`(mUAV_i,=,unsafe)`, `OK`, `True`, `unsafe`), (`(mUAV_i,= safe)`, `Op_incontrol`, `True`, `safe`), (`(mUAV_i,= safe)`, `Agent_incontrol`, `True`, `safe`)$\}$
- $\Upsilon_{\texttt{dNewWp\_i}} = \{$ (`(mNewWp_i, =, x)`, `Op_incontrol`, `True`, `x`), (`(mdist2haz_i,`$\leq$`,minD)`, `Haz-on-Path`, `(tFixated_i,=,False)`, `x`)$\}$
- $\Upsilon_{\texttt{cNewWp\_i}} = \{$ (`(mNewWp_i ,=, x)`, `Op_incontrol`, `True`, `x`), (`(mdist2haz_i'`,$\leq$`,minD)`, `Haz-on-Path`, `(tFixated_i,=,False)`, `x`) $\}$

The update functions generated for the four dependent variables are presented below. See Tables 2–5 for tabular representations of these functions, each expressed as an SCR event table [22].

$$\texttt{tFixated\_i}' = \begin{cases} \texttt{True} & \text{if } \texttt{@T(mOpFix\_i= True)} \wedge \texttt{M\_i} = \texttt{Haz-on-Path} \\ \texttt{False} & \text{if } \texttt{@T(mOpFix\_i= False)} \wedge \texttt{M\_i} = \texttt{Haz-on-Path} \\ \texttt{tFixated\_i} & \text{otherwise} \end{cases}$$

$$\texttt{dUAV\_i}' = \begin{cases} \texttt{unsafe} & \text{if } \texttt{@T(mUAV\_i= unsafe)} \wedge \texttt{M\_i} = \texttt{OK} \\ \texttt{safe} & \text{if } \texttt{@T(mUAV\_i = safe)} \wedge \texttt{M\_i} \in \{\texttt{Op\_incontrol}, \texttt{Agent\_incontrol}\} \\ \texttt{dUAV\_i} & \text{otherwise} \end{cases}$$

Algorithm 2: Update Functions for Terms and Controlled Variables

INPUT: Moded Scenarios $\mathcal{W} = (\mathcal{Y}, \text{MD} = (\mathcal{M}, \mathcal{M}_\text{N}, \mu_0, \mathcal{M}_\text{T}, \mathcal{M}_\text{D}), \text{C})$, Controlled Variables RF_C, Term Variables RF_G
OUTPUT: Set of Update Functions $\{\Upsilon_a | a \in RF_C \cup RF_G\}$
PRECONDITION: $(K_\mathcal{Y} = K_{MD}) \ \wedge \ (K_T \cup K_M = \emptyset)$
PRECONDITION: $\forall Y_i, Y_j \in \mathcal{Y} \ ((y_i = (i, A_i, B_i) \in Y_i \wedge y_j = (j, A_j, B_j) \in Y_j \wedge i = j) \implies (A_i = Aj \wedge B_i = B_j))$

```
foreach a ∈ RF_C ∪ RF_G do
    Υ_a = {};
foreach (μ_i, k, μ_j) ∈ M_T do
    foreach Y ∈ Y do
        if (k, A_k = (m_k, d_k, G_k), B_k) ∈ Y then
            foreach (r, o, v) ∈ B_K do
                Υ_r = Υ_r ∪ {(m_k, μ_i, d_k, v)};
            foreach (r, o, v) ∈ G_K do
                Υ_r = Υ_r ∪ {(m_k, μ_i, d_k, v)};
foreach (μ_i, H) ∈ M_D do
    foreach k ∈ H do
        foreach Y ∈ Y do
            if (k, A_k = (m_k, d_k, G_k), B_k) ∈ Y then
                foreach (r, o, v) ∈ B_K do
                    Υ_r = Υ_r ∪ {(m_k, μ_i, d_k, v)};
                foreach (r, o, v) ∈ G_K do
                    Υ_r = Υ_r ∪ {(m_k, μ_i, d_k, v)};
```

Mode M_i	Event	Event
Haz-on-Path	@T(mOpFix_i = True)	@T(mOpFix_i = False)
tFixated_i′=	True	False

Table 2 Event Table for tFixated_i

$$\texttt{dNewWp\_i}' = \begin{cases} \texttt{x} & \text{if } (\texttt{@T(mNewWp\_i} = \texttt{x}) \wedge \texttt{M\_i} = \texttt{Op\_incontrol}) \vee \\ & \quad (\texttt{@T(mdist2haz\_i} \leq \texttt{minD)} \wedge \texttt{M\_i} = \texttt{Haz-on-Path} \wedge \\ & \quad\ \texttt{tFixated\_i} = \texttt{False}) \\ \texttt{dNewWp\_i} & \text{otherwise} \end{cases}$$

$$\texttt{cNewWp\_i}' = \begin{cases} \texttt{x} & \text{if } (\texttt{@T(mNewWp\_i} = \texttt{x}) \wedge \texttt{M\_i} = \texttt{Op\_incontrol}) \vee \\ & \quad (\texttt{@T(mdist2haz\_i} \leq \texttt{minD)} \wedge \texttt{M\_i} = \texttt{Haz-on-Path} \wedge \\ & \quad\ \texttt{tFixated\_i} = \texttt{False}) \\ \texttt{cNewWp\_i} & \text{otherwise} \end{cases}$$

To demonstrate the semantics of the SCR tabular format, we consider Table 3, a compact representation of the update function $\Upsilon_{\texttt{dUAV\_i}}$ for the controlled variable dUAV_i. The last row of the table gives the possible value assignments for dUAV_i, with one column for each possible value. In the other rows of the table, the leftmost column contains a value for the mode variable M_i and the remaining columns contain events. The table defines a set of updates to variable dUAV_i as follows.

Mode M_i	Event	Event
OK	@T(mUAV_i= unsafe)	False
Op_incontrol, Agent_incontrol	False	@T(mUAV_i= unsafe)
dUAV_i′ =	unsafe	safe

Table 3 Event Table for dUAV_i

Mode M_i	Event
Op_inControl	@T(mNewWp_i = x)
Haz-on-Path	@T(mdist2haz_i ≤ minD) WHEN tFixated_i = False
dNewWp_i′ =	x

Table 4 Event Table for dNewWp_i

Mode M_i	Event
Op_inControl	@T(mNewWp_i = x)
Haz-on-Path	@T(mdist2haz_i ≤ minD) WHEN tFixated_i = False
cNewWp_i′ =	x

Table 5 Event Table for cNewWp_i

For each row j and column k if M_i has the value in row j and the event in cell (j, k) occurs, then the variable dUAV_i is assigned the value in column k of the last row of the table. The entry False in a cell (j, k) indicates that there is no event that can occur when M_i has the value in row j which would cause dUAV_i to be assigned the value in column k of the last row of the table.

3.4.4 Simplify and Extend the Model

As stated in Section 3.2.3, given a Moded Scenarios Description, each conjunct in a Scenario Constraint may either restrict the required system behavior or add new behavior. Consider the conjunct [mOpFix_i = True ⇔ tFixated_i = True]. This formula states that the monitored variable mOpFix_i and the term variable tFixated_i always have the same value. Unlike the two-state function computing the value of mOpFix_i (shown by the event table in Table 2), a more abstract function may be defined that is a one-state, rather than a two-state, function and whose computed value does not depend upon the mode. Hence, the function defining tFixated_i can be more concisely expressed by the SCR condition table shown in Table 6.

Consider next the conjuncts in the Scenario Constraint of the form INIT($r = v$). As described in Section 3.4.2, if, for example, INIT(mOpFix_i = False) appears in a conjunct, then (mOpFix_i, False) is in Θ, which means that the operator in the hazard avoidance task is not initially fixated on UAV_i.

As stated in Section 3.2.3, the conjunct [M_i = Op_incontrol AND mOpFix_i = True AND mOpFix_i′ = False → M_i′ = Agent_incontrol] adds new required system behavior in the hazard avoidance task, i.e., if the operator is responsible for moving the UAV to avoid a hazard but becomes distracted, the system passes control to the adaptive agent. Clearly, the Scenario Constraint is a more efficient way to add this new behavior than changing or extending the ESCs and Mode Diagram in Figs. 3–5. Although adding a new transition from Op_incontrol to Agent_incontrol in the Mode Diagram is easy, modifying the existing ESCs or adding a new ESC to reflect this added behavior is problematic because ESCs, like MSCs, are naturally

	Condition	Condition
	mOpFix_i = True	mOpFix_i = False
tFixated_i′=	True	False

Table 6 Condition Table for tFixated_i resulting from constraint mOpFix_i = True ⇔ tFixated_i = True

Old Mode M_i	Event	New Mode M_i′
OK	@T(mUAV_i = unsafe)	Haz-on-Path
Haz-on-Path	@T(mdist2haz_i ≤ minD) WHEN tFixated_i = True	Op_incontrol
Haz-on-Path	@T(mdist2haz_i ≤ minD) WHEN tFixated_i = False	Agent_incontrol
Op_incontrol, Agent_incontrol	@T(mUAV_i = safe)	OK
Op_incontrol	@T(mOpFix_i = False)	Agent_incontrol

Table 7 Mode Transition Table for M_i updated by constraint (last row)

sequential. Adding a new ESC would repeat most of the system behavior in Figs. 3 and 4 (1, 2, 3, 8) but add a new event sequence between 3 and 8, call it 9, in which the system is notified that the operator is distracted, and the system in response (still in 9) moves the waypoint and updates the display. In contrast, extending the tabular representation to specify this added behavior is trivial. We simply add a new fifth row to Table 1. This added row specifies a transition from Op_incontrol to Agent_incontrol, which is triggered by the monitored event mOpFix_i' = False. Table 7 shows the new mode transition table which reflects this change.

3.4.5 Analyzing the State Machine Model

As other researchers have noted (see, e.g., [12,39]), models synthesized from scenarios will have defects, such as missing initial states, syntax errors, and missing cases. To detect such defects, a tool such as the SCR Consistency Checker (CC) [22, 17] can be applied. The CC finds these and many other "well-formedness" errors automatically. Other classes of errors the CC detects include undefined and unused variables, duplicate variables, circular dependencies, modes that are unreachable, and unwanted non-determinism. When it detects a problem, the CC displays the table with the defect and highlights the specific table entry which contains the defect. For missing cases and non-determinism, called Coverage and Disjointness errors, the CC, after displaying the table which contains the error, presents a counter-example, useful to the developer for diagnosing and fixing the problem.

To remove defects in the state machine model, a software developer has two choices. First, the developer may fix the state machine model indirectly by making changes to the Moded Scenarios Description from which the state machine model was synthesized, i.e., by modifying the set of Event Sequence Charts, the Mode Diagram, and/or the Scenario Constraint. Second, the developer may make changes in the tabular specification of the state machine model directly. As noted in Section 3.1, in our experience, developers who have difficulty creating a tabular specification directly are often able to understand, modify, and extend a tabular specification that is presented to them. Thus, the second alternative is not unrealistic. Further, a practitioner may prefer the second alternative because the effects of changes are more direct and may be easier to predict.

Once the synthesized model is free of well-formedness errors, other tools can be applied—a simulator to ensure that the model captures the intended behavior and verification tools to prove that the model satisfies critical application properties, such as safety and security properties. Eventually, the model may be used as a basis for synthesizing source code, a least for parts of the model, such as the system's control logic and simple functions.

3.5 Related Work

During the last 15 years, there has been a large volume of research published on scenarios and synthesis of formal models from scenarios. This research falls into three categories: 1) Techniques for increasing the expressiveness of scenarios, for making them more formal, and for combining them, both at the same abstraction level and at different abstraction levels; 2) methods for scenario-based synthesis of formal models; and 3) techniques for tool-based analysis of the synthesized models.

Regarding expressiveness, while in ESCs, one or more variables are associated with each entity, variables are not explicit in MSCs. To address this gap, researchers have combined MSCs with other formalisms that include state variables (see, e.g., [39, 11, 12, 37]). *Fluents*—propositions expressed by an initial value and sets of initiating and terminating events—are one such formalism [15]. A fluent may be viewed as a state variable with an initial value and an update function. Using fluents to guard the occurrence of an event, as in [12], is similar to our use of conditioned monitored events. In [12], fluents are used inside High-level Message Sequence Charts (HMSCs) to indicate which MSCs are enabled and thus determine high level behavior. Used this way, fluents are similar to modes. Also, properties expressed in Fluent Temporal Logic (FTL) may describe preconditions on messages [37], and are thus similar to our conditioned monitored events. To constrain UML sequence diagrams, which are similar to MSCs, Whittle and Schumann [39] use state variables to specify pre- and postconditions on events in the Object Constraint Language (OCL). The OCL preconditions play a role similar to our conditions; the OCL postconditions are similar to the effects of our events. Other work on extending MSCs for expressiveness includes specification of negative scenarios [11] and the use of properties to specify system behaviors that are allowed but not necessarily required [37].

Regarding the combination of scenarios, some researchers express relationships between MSCs by attaching *state labels* to a component at appropriate points (see, e.g., [38]). Using the same label in multiple scenarios indicates that the component is in the same state in these scenarios and that at this execution point the component's subsequent behavior can be that of any MSC at the point immediately following the label. In contrast, we relate ESCs using the chart labels on the event sequences in the ESCs and the diagram labels in the Mode Diagram. Rather than representing a single identical state (as state labels do in MSCs), modes represent an equivalence class of states, and thus, do *not* indicate points where execution can switch from one ESC to another.

Regarding scenario-based synthesis of formal models, many researchers have synthesized state machine models from MSCs (e.g., [38, 11, 12, 37]). In [38], Labeled Transition Systems (LTSs) are synthesized from MSCs, using HSMCs and state labels to describe the relationships between the MSCs. LTSs are synthesized in [11] from positive and negative scenarios captured in MSCs. In [12] an LTS is synthesized from HMSCs with the addition of fluents used as guards on the exe-

cution of MSCs in the HMSCs hierarchy. Unfortunately, the state space generated from an LTS specification may be very large. In an industrial case study [38], for example, 658 LTS states were generated, which makes the model incomprehensible to the human user. In contrast, our models are expressed as a set of update functions and represented in a tabular format, producing a state machine model that is more easily understood at the local level. Further, the modes are a system-level abstraction which makes the model easy to understand at the global level. In [39], statechart models, rather than LTSs, are synthesized from UML sequence diagrams and OCL constraints, producing a more human- readable model.

Regarding formal model analysis, researchers have proposed many techniques for analyzing the synthesized models. For example, the fluent guards in the synthesized guarded LTS model can be checked for completeness, disjointness, and reachability [12]. These checks are similar to the checks performed on our formal model by the SCR Consistency checker [22]. In [11], fluents are used in combination with the synthesized LTS model to generate state invariants that hold in the model. Similarly, in [12], FTL properties are analyzed by model checking the synthesized LTS model and are used in combination with the synthesized model to generate state invariants that hold in the model. As stated above, the SCR toolset can also generate state invariants from a formal model [26, 29]. Alur and Yannakakis [4] describe how individual MSCs, MSC-graphs consisting of multiple MSCs; and HMSCs can be linearized into an automaton and verified using model checking. As described in Section 3.1, the SCR toolset provides all of these techniques, i.e., consistency checking, invariant generation, model checking, as well as theorem proving, in a single toolset.

3.6 Future Work

We are currently developing a tool for synthesizing a formal requirements model from a Moded Scenarios Description using the method described in Section 3.4. The tool will compute the update functions by applying the two algorithms described in Section 3.4.3. it will then translate the state variables, their initial values, and the update functions obtained from the Moded Scenarios Description into a partial specification of an SCR model. After the developer has produced a more complete model, by adding information, such as variable types and missing initial values, the developer can apply the CC to find additional defects in the model.

In addition to the two algorithms described in Section 3.4.3, a third algorithm is needed that automatically builds (some of) the type sets of the state variables and associates them with the appropriate variables. For variables that are clearly Boolean (e.g., because they are assigned values of `True` or `False`), this is trivial. In the hazard avoidance task, for example, the monitored variable `mOpFix_i` is clearly Boolean. Constructing the type sets of enumerated types should also be relatively straightforward. In our example, both the monitored variable `mUAV_i` and the controlled variable `dUAV_i` have values in $\mathcal{T}_{\texttt{UAV}} = \{\texttt{safe}, \texttt{unsafe}\}$, although adding more ESCs, and hence more behavior, could cause more values to be added to $\mathcal{T}_{\texttt{UAV}}$. How to specify the type set of variables with numerical values is an open question. In our example, we can assume from the context that some of the variables, e.g., `mdist2haz_i`, have numerical values. However, whether these values are integers, reals, or limited to some interval is difficult to determine. Moreover, the waypoint, `cNewWP_i`, might be expressed by a triple (x, y, z), where x, y, and z

are latitude, longitude, and altitude. Clearly, the developer would need to provide type sets in these cases.

Another issue to be explored is models with more than a single mode class and how to compose them. In the hazard avoidance task, the operator manages a team of many UAVs, rather than a single UAV. By creating n versions of the system model synthesized from the ESCs and the Mode Diagram in Figs. 3–5, the developer can create n models, each with a mode class $\mathtt{M_i}$, $i = 1, 2, \ldots, n$. Composing these together will produce a system that, with the agent, protects n UAVs from hazards, rather than a single one. When the individual UAVs in a team are independent of one another, the composition is essentially parallel, and can be done by simply combining the tables of all of the n models. When models of systems that interact with one another are composed, the composition may need to add constraints that limit the interaction. For example, particular modes of two or more systems may be incompatible; for example, a UAV that is avoiding a hazard should not initiate a targeting task. How best to capture this incompatibility (i.e., "feature interaction") using scenarios and tabular requirements specifications is an open question.

4 Cognitive Model

The goal of our cognitive modeling research is to develop a model that can predict when a human operator performing a set of tasks will become overloaded. Such a model can help identify situations in which an agent could assist the operator by taking over one or more operator tasks.

As robots have become increasingly autonomous, one human sometimes supervises multiple robots. This single-human-multiple-robot (SHMR) paradigm, while labor-saving, raises human factors concerns about the capabilities and limitations of the human operator who is multitasking in this situation.

Crandall, Cummings, and Mitchell [9,10] have introduced "fan-out" models to estimate the maximum number of robots a single operator can supervise in a given SHMR context. These estimates are based on *neglect time* (NT), the time a robot may be neglected before its performance falls below a predetermined threshold; *interaction time* (IT), the time a operator needs to interact with a robot to restore its performance to an above threshold level; *wait time attention* (WTA), the time required for the operator to notice that the robot requires maintenance; and *wait time queue* (WTQ), the delay in interacting with the robot when other robots require maintenance at the same time. Fan-out models have been shown to predict the overall performance of operators on SHMR platforms with different numbers of robots to supervise.

One limitation of fan-out models is that they do not predict performance during the course of the SHMR session. Even when the number of robots supervised is within the constraints specified by a fan-out model, there will be times when the operator is overloaded and as a result subject to error. This becomes clear when we consider that the components of the fan-out model (IT, NT, WTA, WTQ) fluctuate from their typical values during the course of a session and at times will conspire to increase the likelihood of error, whether through the increase in the time needed to maintain a vehicle (IT) or the increase in the wait times, WTA and WTQ, as a result of several vehicles requiring attention at the same time.

To address this limitation, we have developed a dynamic model to predict operator overload during the course of a SHMR session. This dynamic operator overload (DOO) model [32,14,7] uses predictor variables similar to the fan-out variables. The model was initially developed for the supervision of multiple unmanned air vehicles (UAVs) by one operator. The SHMR platform on which the model was tested is the RESCHU system introduced in Section 2.

In our implementation of RESCHU, each UAV is automatically assigned to a target on which to deliver ordnance and automatically launched on a straight-line flight path towards its target. This path appears in the Map window (see Fig. 1). In RESCHU, the operator has two goals: to engage as many vehicles as possible and to prevent damage to the UAVs. Our main concern was preventing UAV damage.

In the DOO model, failure to prevent damage was the indicator of operator overload. In the generation of the model, the unit of analysis was a *path-intersects hazard* (PIH) event, which started from the moment a UAV entered on a collision course with a hazard (i.e., a threat) and ended either when the UAV traversed the hazard, and consequently incurred damage, or else when the operator changed the UAV's trajectory to prevent damage from occurring. Multiple PIH events can overlap in time. The model was created using logistic regression analysis, with NT operationalized as the expected duration of the PIH assuming damage occurs and WTA by the amount of time until the operator first fixates on the relevant hazard. The amount of time required to execute an evasive action on the relevant vehicle (IT) was not included since it is not relevant when damage is avoided. An initial model included WTQ, operationalized as the amount of time spent on actions focused on nonrelevant vehicles (i.e., other than the PIH's vehicle), including hazard evasions or target engagements. While this model was fairly strong, a superior model was generated by replacing WTQ with WTF, the number of eye fixations on objects (UAVs, targets, hazards) not relevant to the PIH event. WTF and WTA are based on eye fixations assessed by an eye tracker. Thus, the predictor variables in the DOO model were NT, WTA, and WTF.

The DOO model provided a strong fit to the data in the baseline experiment from which it was generated. One measure of fit, d', was 2.7, which is quite high.[2] The model was evaluated in a replication of the baseline experiment (d'=2.4). The model's generalizability was assessed by factorial comparisons [36] of the initial RESCHU platform to platform variations, including where engagement was time-constrained and thus harder (d'=2.2), where engagement was automated and thus easier (d'=2.6), where UAVs moved faster (d'=1.7) or slower (d'=2.3) than in the baseline, and where the operator supervised heterogeneous vehicles (UAVs, HALEs,[3] and UUVs) (d'=2.4), rather than only UAVs. In all cases, except the high-speed UAVs, generalization was excellent.

Additionally, the DOO model has been incorporated into the RESCHU platform as the basis for alerting the operator to PIHs as soon as damage is predicted. Here the model assesses the likelihood of damage repeatedly during the course of each PIH event, rather than after the fact, and as soon as it predicts damage is probable, it alerts the user to the threat. The model-based alert system has been tested in several experiments and has been found to reduce instances of damage by as much as half. The model-based alerts typically occur after the elapsing of

[2] A measure of fit at or above 2.0 is considered very good.

[3] High Altitude Long Endurance UAVs.

approximately 20% of the time between identifying a UAV's proximity to a hazard and occurrence of damage (i.e., 20% of NT), thus providing a timely warning.

Thus, the DOO model has practical utility in helping operators cope with overload situations in the course of SHMR supervisory control tasks. It also has theoretical value in highlighting the roles of attention and planning in multitasking contexts.

4.1 Limitations

The current cognitive model focuses on predicting when an operator becomes overloaded. The cognitive model focuses on minimizing damage to UAVs from threats. While the theory is applicable to any situation where the operator must deal with multiple UAVs, it has not yet been applied to other situations. Ongoing work is examining how to use the theoretical model to predict operator overload during secondary tasks like engaging a target.

5 Agents

In a human-centric decision system, when the user is overloaded it is beneficial to have an agent that can be invoked to assist the user in performing tasks. Our goal is to evaluate a large range of agent designs for a given system (e.g., RESCHU) to determine which designs best assist the user. However, evaluating a large number of agent designs using human participants is infeasible given that even small participant studies require substantial time and resources.

Traditionally, developers evaluate the performance of an agent by conducting human participant studies, iterating the process many times as the agent is refined. The cost and time required for such studies can lead to slow iterations. In contrast, our approach uses synthesized *user models* instead of actual people for some iterations. User models are given the same observations that a human participant would be given and must produce specific actions. Simulated user studies with these models are inexpensive, can be performed quickly, and require no human participant board approval. Thus, they provide an efficient way to identify problems with an agent design.

Like others, we frame synthesis of user models as an *imitation learning* task [34], where our goal is for the models to learn to operate by imitating human participants. Our work differs from much of the earlier work in imitation learning because, in our case, building an observation-action model without a significant amount of state abstraction is infeasible. The original developers of RESCHU have learned a user model for RESCHU [10], but their model, unlike our user models, is descriptive and cannot be used to generate actual operation in RESCHU. Furthermore, their model does not describe individual users, but an aggregate of users.

Table 8 provides a high level description of our five-step method for learning user models and using them to evaluate agent designs. The method and an initial case study applying the method to RESCHU are described briefly here. Full details of the application of the method to the RESCHU case study can be found below in Section 5.1. Our method begins (Step 1) with a set of traces of human behavior (i.e., sequences of observed human actions and environmental updates to the display); in our case study, these traces were obtained from data collected in previous studies with RESCHU involving human participants. The user actions in a trace are captured at a high level of abstraction (e.g., "delete waypoint" instead of "delete the waypoint at location `x` for `UAV_i`").

Table 8 Method for synthesizing and applying user models

1. Gather traces of human behavior in an initial participant study.
2. Synthesize user models from traces.
   - Extract feature vectors from traces.
   - Construct an expert operator.
   - Reduce capabilities of expert operator to match user feature vectors.
3. Evaluate whether the models accurately emulate humans.
   - Learn a user classifier.
   - Extract traces from the user models.
   - Attempt to fool the classifier with the models' traces.
   - If the model fails to fool the classifier, go to Step 2.
4. Iterate agent design using the user models:
   - Build/modify an agent.
   - Use the models to test the agent.
5. Test agent performance with human participants.

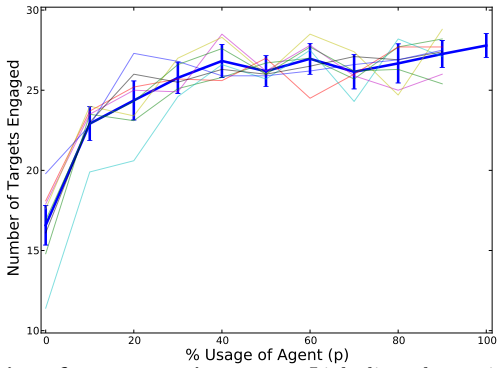

Fig. 6 User models' performance using agent. Light lines denote individual user models' performances; bold denotes average performance.

In Step 2, we synthesize a model for each human user in the data set by first extracting a set of features from each trace for that user and then using the resulting set of feature vectors together with a manually specified *expert operator* (an implementation that behaves like a near optimal user) to produce a model of the user. In our case study, we generated a set of 36 features, including: tallies over each of the high level RESCHU actions, tallies over the ordered pairs or "bigrams" of the high level actions (where the bigram tally `A`→ `B` is the number of times `B` was the first action after `A`), and the average distance between the UAVs. Each feature was assigned a weight, indicating how useful the feature is in distinguishing between users, using a variant of the RELIEF feature weighting algorithm [28].

In Step 3, the behavior of the synthesized user models is compared to that of the actual humans. To do this, we trained a classifier given the original user traces and applied it to classify traces generated from the synthesized user models (i.e., the classifier identifies which user it thinks produced the trace). For example, in our application of this process, when the classifier misidentified the user, we used that information to improve the synthesized models in the next iteration of Step 2, e.g., by suggesting a different set of features.

Once sufficiently accurate, the user models may be applied to evaluate agent designs (Step 4). In our case study, we evaluated an agent, derived from the expert

operator, which suggests an action at every step of the test. We learned eight user models and tested their performance in RESCHU with the agent, varying p, the probability that the user model used the agent's suggestion at any time step, from 0 to 100%. The performance metric is the number of targets acquired. Fig. 6 shows the results. When $p=0$, each model performs as if there is no agent. When $p=100\%$, the models behave identically to the expert operator. While the increased use of the agent improves performance for all user models, the models with the lowest non-assisted performance benefit the most from the agent. For this application, it would be feasible to set $p=100\%$ because the number of suggestions given by the agent averages 1 every 3.3 seconds (and the agent rarely gives more than one suggestion per second). However, in a human-centric decision system, user participation is essential, and too much automation is inadvisable because it can lead to user disengagement.

Because the user models only approximate human behavior, the performance of the agent design that is finally selected should be validated in a study involving actual human participants (Step 5). Evaluation of the agent design for the RESCHU case study has not yet been performed; it is future work.

5.1 Case Study Implementation Details

This sub-section provides additional details on the application of the first four steps of our process in the RESCHU case study.

Step 1. Gathering Initial Traces We collected 10 traces from each of 8 human users. Each trace recorded the user's actions and the observations available to him or her while operating RESCHU for a 5-minute period. There are six different high level actions the user can perform. Five of them are available at any time: change the assignments of UAVs to targets (`change goal`), insert a waypoint (`insert WP`) , delete a waypoint (`delete WP`), move a waypoint (`move WP`), and do nothing (`NOP`). The sixth action, engage a target (`engage target`), is available only when a UAV has arrived at its assigned target. Though we retained no identifying information about the users, each user is referred to by an index from 1 to 8.

Step 2. Synthesizing User Models Because the number of possible observations a user may receive in RESCHU is large ($\sim 10^{124}$), building an observation-action model, such as those used in [34] and [35], was infeasible without a significant amount of state abstraction. Instead, as described above we extracted feature vectors from the 80 traces and used the RELIEF feature weighting algorithm [28] to help judge which features are most distinguishing for our 8 users (i.e., we want features that have high variability between users, but low variability within vectors generated by a single user). We chose RELIEF over other feature weighting algorithms because of its popularity and ease of implementation.

From each trace, we generated a set of 36 features. These include tallies over the 6 high level actions, tallies over the 25 "bigrams" for the 5 non-NOP actions (where `A`$\rightarrow$ `B` is the number of times `B` was the first non-NOP action after `A`), the average proximity to a hazard a UAV reached before a user rerouted the UAV, the average distance between (or "spread") of the UAVs, the average time a UAV was idle (how long between when the UAV arrived at its target and when the

Table 9 The highest weighted features and their discrimination scores, where high level actions are shown in `this font`.

Feature	Weight
average distance to hazard before action	.71
tally: `change goal`	.49
bigram tally: `engage target`→`change goal`	.29
bigram tally: `change goal`→`engage target`	.23
tally: `engage target`	.18
bigram tally: `change goal`→`add WP`	.13
bigram tally: `change goal`→`change goal`	.12
tally: `delete WP`	.12
average distance between UAVs	.12

user engaged the target), the average number of (non-NOP) actions the user took divided by the user's performance score (as a measure of action efficiency), and the average amount of time a user waited between when a UAV's path crossed a hazard area and taking action to reroute the UAV. Note that the number of `engage target` actions is the same as the performance measure (the number of targets engaged). Table 9 shows the top weighted features. We then modified the expert operator to reflect the behavior of a specific user, as described by some of the most heavily weighted features; we then used this modified expert operator as the model for that user.

Since reaction time is a limiting factor in human-user performance for RESCHU, we were able to build a simple rule-based "expert" operator (which we referred to above) that outperformed our best human operator. The expert operator uses the following rules:

1. **if** there is a UAV waiting at a target, `engage` that target.
2. **else if** the UAV target assignments are "suboptimal" (using a greedy nearest target to UAV calculation), reassign the targets greedily (using `change goal`s).
3. **else if** a UAV's flight path crosses a hazard area, reroute the UAV around the hazard (using `add WP`).

This simple expert operator performs well, averaging a score of 27.3 targets acquired per 5-minute session, compared to an average of 21.7 targets acquired per session for the human users. The highest scoring human user averaged 24.5 targets per session.

This expert operator served as a basis for constructing individual user models. Based on the results shown in Table 9, we constrained our expert operator to mimic the two highest weighted features: the user's average distance to hazard before an evasive action and the user's average total number of `change goal` actions. We also constrained the expert operator to mimic the user's average UAV idle time, a feature that, while it was not one of the highest weighted features, was easy to implement; the expert operator can simply delay engaging a target to better match the user's average UAV idle time.

Step 3. Evaluating the User Models Since there is a good deal of interdependence among the features in a user's feature vector, we hypothesize that it would be difficult to generate a trace that produces a similar feature vector, but that operates with a markedly different style. We implemented the substeps of

Step 3 and trained a classifier to distinguish human users by their feature vectors, extracted traces from the user models, then tested whether the classifier can distinguish human users from their models.

For each of our 8 human users, we have 10 vectors of 36 real-valued features. Thus, learning a classifier is a straightforward application of multiclass supervised learning, with the classes being the 8 individual users. We trained a suite of 35 classifiers from the PyMVPA toolkit [16] using leave-one-out cross-validation (i.e., train the classifiers on nine of a user's vectors and use the tenth for validation, repeating this process ten times so that each vector is used for validation once). Included in the classifier suite were various paramaterizations of Bayes Logistic Regression, Naive Bayes, Gaussian Process Classifier, k-Nearest Neighbors, Least-Angle Regression, Sparse Multinomial Logistic Regression, Elastic Networks, and Support Vector Machines (SVMs) with various kernels. The most accurate of these classifiers for our data was an SVM with a polynomial kernel (Poly-SVM), which yields an average of 62.5% accuracy on the test set (compared to 12.5% accuracy for a random classifier).

Applying the user model, we generated 10 new traces for each of our 8 users with the aim of mimicking that user. Poly-SVM's classification accuracy was 27.5% on these traces. That is, given a trace generated by the model for user u, the classifier correctly identified the trace as coming from user u 27.5% of the time. (In comparison, a random classifier yield only 12.5% accuracy.) For this classifier, we consider a practical upper limit to be 62.5% because this is the classifier's accuracy on actual human traces.

We interpret these results to mean that the user models performed poorly at fooling the classifier, meaning that they do not emulate their human counterparts very well. In practice, we would want both the classifier and the model's accuracy to be much higher. The current results can be improved by returning to Step 2 and using the classifier's predictions to guide our selections of new combinations of features for use in synthesizing user models. We can iterate between Steps 2 and 3 to find a set of features that produces higher accuracy for both the classifier and the model. Our current work is a demonstration of principle; we plan to improve these results in our future work.

Step 4. Testing an Agent Design We implemented an agent that constantly "suggested" an action —the action the expert operator would perform— at every step of the session. The user (or the user model) then had the choice of executing a high level action, or calling the agent, which would then execute its suggested action.

We had no model for how users would use an agent (as there were no agents available to them when we collected the traces), so we parameterized the user models' interaction with the agent. That is, we introduced a parameter p that denotes the percent probability that a user model will call the agent at any particular time. As described above, we tested the agent by running the eight user models varying p from 0 to 100%, obtaining the results shown in Figure 6.

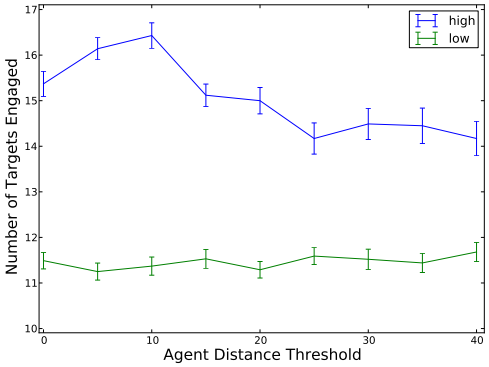

Fig. 7 User models' performance using agent with different thresholds. A high-performing user model requires less help from the agent.

5.2 Adjusting Agent Interaction for Different User Types

While our results, shown above and first described in [31], show that use of an agent can improve a user's (or user model's) performance, an overly active agent may interfere with a user's goals, causing a decrease in performance. The question of *when* an agent should provide assistance to a user must be addressed.

In this section, we describe new results using a modified agent that "stalls" individual UAVs when the UAV comes within a certain *threshold* distance t to a hazard. We found that the optimal choice of this threshold varies for different user models, with higher-performing user models generally requiring a shorter distance (and therefore less assistance) than lower-performing users.

For this experiment, we modified the agent so that its only available action is to pause a UAV (i.e., as if it is flying in a tight holding pattern). This modification helps to maintain the user's situational awareness because missions cannot be accomplished without the user's participation. The agent pauses a UAV when the UAV gets within distance t from a hazard area.

While stalling a UAV can prevent it from entering a hazard area and being damaged, it can also delay the UAV from reaching its target. To find the optimal value for t for each user model, we ran each of our eight user models using agents with t ranging from 0 (effectively causing the agent to never assist) to 40 (causing frequent assistance).

The performance results (averaged over 100 trials) for the lowest and highest performing user models are shown in Fig. 7. Note that, compared to operating without an agent, the higher-performing user model benefits most from a low distance threshold (i.e., a less proactive agent), but has reduced performance when interacting with an overly intrusive agent. We hypothesize that this is because the optimal strategy for a user model might sometimes be to fly a UAV near (but not entering) a hazard area. An overly intrusive agent would prevent this possibility. In contrast, our lower-performing user model's performance did not substantially vary with t.

The modified agent used in this experiment allows for the possibility of tight integration with cognitive models, such as those described below in Section 4. Instead of using the distance from a UAV to a hazard as the threshold for agent assistance of a user, we could instead apply a threshold based on the probability

that a user will allow a UAV to enter a hazard area, as assessed by the cognitive model.

5.3 Future Work

Future work includes fully automating the process of learning user models from traces, and testing the derived models more thoroughly. Also, we plan to extend our RESCHU case study to include many iterations of agent development (i.e., iterating Steps 2-4 of the method). One major research issue is validating user models. The hypothesis that a user model that matches a user's feature vectors will also match the user's style of operation needs to be validated. The interdependence of the user's features also needs to be assessed (e.g., we can omit the "bigram tally: `engage target` → `change goal`" feature to see if a user model that matches the other features also matches this feature). As mentioned above, future work also includes improvements in each of the five steps of our method for learning user models and using them to evaluate agent designs. In particular, we have left as future work agent design validation using human participants (Step 5). We also plan to iterate Steps 2 and 3 to improve the fidelity of our user models.

6 Conclusions

This paper has presented a Moded Scenarios Description, a new technique which allows a developer to specify the required behavior of complex systems in terms of scenarios. A Moded Scenarios Description includes Event Sequence Charts, a variant of Message Sequence Charts; a Mode Diagram; and a Scenario Constraint. It also introduced a formal model for a Moded Scenario Description, a method for transforming the description into a formal state machine model, and two algorithms that construct the next-state function of the state machine model from the description. The state machine model is represented in a scalable tabular format which has proven in practice to be both human readable and easy to change. We illustrated our new method by applying it to a hazard avoidance task for a system that manages UAVs. The system uses an adaptive agent to assist the operator.

After reviewing the process for synthesizing user models presented in [31] and [20], this paper introduced the results of new research showing how different user models for a hazard avoidance task perform under different values for the distance threshold t at which the adaptive agent is triggered in a hazard avoidance scenario. This agent design facilitates integration of the agent and a cognitive model of the user; a slight variation of the agent can threshold on predictive values returned by the cognitive model.

The cognitive model can predict in real-time when an operator is overloaded. The model is theoretically based and uses cognitive science and engineering principles to monitor operator workload. The model has been tested across multiple experiments and has been shown to reduce damage to UAVs [7,14]. The cognitive model is important not only because it is able to predict operator workload and prevent UAV damage, but also because it can be integrated into other systems where understanding whether an operator is overloaded is important. Integrating the theoretical model of workload, the agent approach to automation, and high assurance methods has the potential to significantly improve the overall performance of systems and to provide high assurance that these systems behave as intended.

Acknowledgements We gratefully acknowledge the contributions of Leonard Breslow to the research on cognitive models described in Section 4.

References

1. Ardupilot. http://www.diydrones.com/notes/ArduPilot.
2. The role of autonomy in DoD systems. Technical report, Defense Science Board, Office of the Under Secretary of Defense for Acquisition, Technology and Logistics, Washington, DC, July 2012.
3. T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore. Software requirements for the A-7E aircraft. Technical Report NRL-9194, Naval Research Laboratory, Washngton, D.C., 1992.
4. R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proc. 10th International Conference on Concurrency Theory (CONCUR)*, pages 114–129, Eindhoven, The Netherlands, 1999.
5. R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements method. In *Proc. 19th Digital Avionics Systems Conference (DASC)*, Philadelphia, Pennsylvania, Oct. 2000.
6. Y. Boussemart and M. Cummings. Behavioral recognition and prediction of an operator supervising multiple heterogeneous unmanned vehicles. In *Proc. 1st International Conference on Humans Operating Unmanned Systems (HUMOUS)*, Brest, France, Sept. 2008.
7. L. A. Breslow, D. Gartenberg, J. M. McCurry, and J. G. Trafton. Dynamic fan out: predicting real-time overloading of an operator supervising multiple UAVs. Under review.
8. E. Bumiller and T. Shanker. War evolves with drones, some tiny as bugs. *New York Times*, June 2011.
9. J. W. Crandall, M. A. Goodrich, J. D. R. Olsen, and C. W. Nielsen. Validating human-robot systems in multi-tasking environments. *IEEE Transactions on Systems, Man, and Cybernetics*, 35(4):438–449, July 2005.
10. M. L. Cummings and P. J. Mitchell. Predicting controller capacity in supervisory control of multiple UAVs. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, 38(2):451–460, Mar. 2008.
11. C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transiactions on Software Engineering*, 31(12):1056–1073, Dec. 2005.
12. C. Damas, B. Lambeau, F. Roucoux, and A. van Lamsweerde. Analyzing critical process models through behavior model synthesis. In *Proc. 31st International Conference on Software Engineering (ICSE)*, pages 241–251, Vancouver, Canada, May 2009.
13. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 146–162, Toulouse, France, Sept. 1999.
14. D. Gartenberg, L. Breslow, J. Park, J. McCurry, and J. Trafton. Adaptive automation and cue invocation: The effect of cue timing on operator error. In *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 3121–3130, Paris, France, Apr. 2013.
15. D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 257–266. ACM, 2003.
16. M. Hanke, Y. O. Halchenko, P. B. Sederberg, E. Olivetti, I. Fründ, J. W. Rieger, C. S. Herrmann, J. V. Haxby, S. J. Hanson, and S. Pollmann. PyMVPA: a Python toolbox for multivariate pattern analysis of fMRI data. *Neuroinformatics*, 7(1):37–53, Mar. 2009.
17. C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *Computer Systems: Science and Engineering*, 20(1), Jan. 2005.
18. C. Heitmeyer and R. Jeffords. Applying a formal requirements method to three NASA systems: Lessons learned. In *Proc. IEEE Aerospace Conference*, page 84, Big Sky, Montana, Mar. 2007.
19. C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, Nov. 1998.

20. C. Heitmeyer, M. Pickett, L. Breslow, D. Aha, J. G. Trafton, and E. Leonard. High assurance human-centric decision systems. In *Proc. 2nd International NSF-Sponsored Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, May 2013.
21. C. L. Heitmeyer, M. M. Archer, E. I. Leonard, and J. D. McLean. Applying formal methods to a certifiably secure software system. *IEEE Transactions on Software Engineering*, 34(1):82–98, Jan. 2008.
22. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
23. C. L. Heitmeyer, S. Shukla, M. M. Archer, and E. I. Leonard. On model-based software development. In J. Munch and K. Schmid, editors, *Perspectives on the Future of Software Engineering*, pages 49–60. Springer, 2013.
24. K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, Jan. 1980.
25. ITU. Message sequence charts. Recommendation Z.120, Intern. Telecomm. Union, Telecomm. Standardization Section, 1996.
26. R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. 6th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 56–69, Lake Buena Vista, Florida, Nov. 1998.
27. R. D. Jeffords and C. L. Heitmeyer. A strategy for efficiently verifying requirements. *SIGSOFT Softw. Eng. Notes*, 28:28–37, Sept. 2003.
28. K. Kira and L. A. Rendell. A practical approach to feature selection. In *Proc. 9th International Workshop on Machine Learning (ML)*, pages 249–256, Aberdeen, Scotland, July 1992.
29. E. I. Leonard, M. Archer, C. L. Heitmeyer, and R. D. Jeffords. Direct generation of invariants for reactive models. In *Proc. 10th ACM/IEEE Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 119–130, 2012.
30. E. I. Leonard and C. L. Heitmeyer. Program synthesis from formal requirements specifications using APTS. *Higher-Order and Symbolic Computation*, 16(1-2):63–92, Mar. - Jun. 2003.
31. M. Pickett, D. W. Aha, and J. G. Trafton. Acquiring user models to test automated assistants. In *Proc. 26th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pages 112–117, May 2013.
32. R. Ratwani and J. G. Trafton. A real-time eye tracking system for predicting postcompletion errors. *Human Computer Interaction*, 26(3):205–245, 2011.
33. T. Rothamel, C. Heitmeyer, E. Leonard, and A. Liu. Generating optimized code from SCR specifications. In *Proceedings, ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2006)*, June 2006.
34. C. Sammut, S. Hurst, D. Kedzier, and D. Michie. Learning to fly. In D. H. Sleeman and P. Edwards, editors, *Proc. 9th International Workshop on Machine Learning (ML)*, pages 385–393, Aberdeen, Scotland, July 1992. Morgan Kaufmann.
35. D. Šuc, I. Bratko, and C. Sammut. Learning to fly simple and robust. In *Proc. 15th European Conference on Machine Learning (ECML)*, pages 407–418, Pisa, Italy, Sept. 2004.
36. J. A. Swets. *Signal detection theory and ROC analysis in psychology and diagnostics: Collected Papers*. Lawrence Erlbaum Associates, 1996.
37. S. Uchitel, G. Brunet, and M. Chechik. Synthesis of partial behaviour model synthesis from properties and scenarios. *IEEE Transactions on Software Engineering*, 35(3):384–406, Mar. 2009.
38. S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, Feb. 2003.
39. J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, pages 314–323, Limerick, Ireland, 2000.

A Notation

symbol	usage
A	a monitored event expression
B	a set of controlled events
C	a Scenario Constraint
C^I	a set of initial value predicates of the form $\texttt{INIT}(r = v)$
F	a map of variables to associated entities
G	a set of term events
H	a set of numeric labels
K	a set of numeric labels
X	a set of entities
Y	an event sequence chart
Z	the integers
Z^+	the positive integers
a	a scenario state variable
o	a binary relational operator such as $=$ or $\leq$
r	a scenario state variable
s	a state; equivalently, a map from state variables to values
$\hat{s}$	a scenario state; equivalently, a map from scenario state variables to values
MD	a mode diagram
RF	the set of scenario state variables
RV	a set of variable/value pairs
TS	a union of type sets; possible values of scenario state variables
TY	the map of variables to their associated type
VS	TS $\cup$ $\mathcal{M}$; possible values of state variables
$\mathcal{M}$	a set of modes
$\mathcal{M}_N$	a mode class; a special variable whose set of possible values is a set of modes
$\mathcal{M}_T$	a set of mode transitions
$\mathcal{M}_D$	a set of mode/numeric label pairs; in effect, a map from modes to sets of numeric labels
$\mathcal{T}$	a set of values; equivalently, a type set, or a type
$\mathcal{Y}$	a set of event sequence charts
$\mathcal{W}$	a Moded Scenarios Description
Θ	the initial state predicate of a state machine
μ	a mode
μ_0	the initial mode
ρ	the transition relation of a state machine
Υ	an update function

Table 10 Notation used in this paper. Unless explicitly indicated otherwise in this table, a subscript attached to a symbol is used to associate the object it denotes with an object denoted by the subscript symbol.