Obtaining Trust in Autonomous Systems: Tools for Formal Model Synthesis and Validation

Constance L. Heitmeyer and Elizabeth I. Leonard
Naval Research Laboratory, Washington, DC 20375
Email: {constance.heitmeyer, elizabeth.leonard}@nrl.navy.mil

***Abstract*—An important and growing class of cyber physical systems are autonomous vehicles (AVs). While the U.S. military has used AVs, such as unmanned air vehicles, for many years, civilian use of these vehicles, e.g., by the FBI, has been steadily increasing. Plans to equip AVs with cameras, scientific instruments, and weapons, such as tear gas and pepper spray, have led to growing mistrust of AVs and calls for greater human control and oversight. This paper describes two kinds of trust needed in systems involving AVs and how a formal model and model-based simulation can help obtain such trust. It introduces two new tools to be integrated into FOrmal Requirements Modeling and AnaLysis (FORMAL), an upgrade of NRL's Software Cost Reduction (SCR) toolset; the new tools support formal modeling and symbolic execution (via simulation) of cyber physical systems. The first tool synthesizes a formal model from scenarios. The synthesized model forms a basis for formal verification, and for model validation using simulation. The second tool, which combines the existing SCR simulator with the eBotworks 3D autonomy simulator, overcomes the SCR simulator's major limitations—its support for only discrete computation and its inability to simulate continuous movement. To evaluate the new tools, the paper describes synthesis of a formal model of a Navy unmanned ground vehicle currently under development and simulation of its behavior.**

I. Introduction

A cyber physical system combines a cyber component (e.g., a computer) with a physical environment. The cyber component controls the physical environment using sensors that observe the physical environment and actuators that actuate the controls [1]. Developing cyber physical systems is challenging because the computer and physical processes, treated as separate in the past, must be connected. An important and growing class of cyber physical systems are autonomous vehicles (AVs) and systems managing AVs. The U.S. military currently deploys AVs to perform tasks such as surveillance, reconnaissance, and targeting. Most of these AVs are not entirely autonomous but remotely controlled by humans. According to [2], AVs and systems managing them have already had a major impact on military operations worldwide.

AVs are also being deployed increasingly in civilian applications such as law enforcement and public safety. The FBI and U.S. Border Protection Agency, among others, have equipped unmanned air vehicles with cameras and scientific instruments to conduct surveillance and gather information [3]. These applications and recent plans to equip unmanned air vehicles with nonlethal weapons, e.g., tear gas and pepper spray, have raised serious privacy and other concerns [4] and led to calls for greater human control and oversight.

A major problem for systems controlling AVs is lack of human trust in the system's autonomy software. According to a 2012 Defense Science Board report on autonomy's future role in U.S. military systems, commanders and operators lack "trust that the autonomous functions of a given system will operate as intended..." [2]. The result is that most AVs used by the military are teleoperated. Researchers at Raytheon have distinguished two kinds of trust needed in these systems: *system trust*, human confidence that the system behaves as intended, and *operational trust*, confidence that the system will help a human perform the assigned tasks [5]. To evaluate trust in AVs, researchers have recently designed measures of human trust in the autonomy software, see, e.g., [6]; when the trust level, modified by either a human or an algorithm, falls below some threshold, the system can take special actions.

One approach to obtaining human trust in autonomy is to apply formal methods. Building a formal model of the autonomy software and verifying that it satisfies critical (e.g., service and safety) properties can lead to system trust, i.e., high assurance that an autonomous system satisfies its requirements. Providing users with a powerful simulation capability that symbolically executes the autonomy software based on the formal model can lead to operational trust; an operator can run the simulator to validate that the autonomous software will help him/her perform the assigned tasks.

To date, most applications of formal methods to AVs and systems managing them have largely fallen into two classes. First, many researchers have formally modeled a robot's required behavior in, e.g., LTL, and used the model to automatically synthesize the robot's control software; see, for example, [7]. There has also been extensive use of formal methods, especially model checking, to verify the the human-machine interaction (HMI) in current autonomous systems [8]. As stated in [2], the HMI in current autonomous systems is "frequently handicapped by poor design" and requires more research. To address the HMI problem, we have developed an approach, based on cognitive science, adaptive agents, and formal methods, in which a cognitive model predicts human behavior (e.g., operator overload), an adaptive agent helps an operator perform the assigned tasks, and a formal model provides evidence that the system behaves as intended [9]. To demonstrate our approach, we applied it to a system managing a team of unmanned air vehicles [9].

At least two significant barriers exist in applying formal methods to systems involving AVs. First, while modeling

978-1-4673-7043-1 2015

DOI 10.1109/FormaliSE.2015.16

and formal verification can lead to system trust—i.e., high confidence that a system satisfies its requirements—a major problem is how to obtain the formal system model. Difficult to obtain in general, in software practice, formal models of systems managing AVs are virtually non-existent. Moreover, while simulation can help a user develop operational trust of a given system, most simulators based on formal methods are not designed to simulate cyber physical systems. Like the simulator in the SCR toolset [10], they have serious limitations—they cannot easily simulate continuous vehicle movement and support only discrete computation and two-dimensional (2D) displays. To overcome the first barrier, we have developed a new tool which synthesizes a formal system model from user scenarios. To overcome the second, the existing SCR simulator has been integrated with a 3D application-specific simulator that realistically models the continuous nature of the system's physical environment.

As background, Section II reviews the SCR tabular notation, the SCR toolset, and a Moded Scenarios Description [9], a new graphical scenario language which uses Event Sequence Charts and a set of modes to specify the required system behavior. Sections III and IV contain the paper's major contributions—they describe 1) a new model synthesis tool which creates a formal model from a Moded Scenarios Description, and 2) a new simulator which combines the simulator in the SCR toolset with a 3D simulator for autonomy software. To illustrate the new tools, we describe how we evaluated the tools by applying them to an UGV currently under development by the U.S. Navy. The new integrated simulator exposed a serious safety problem in our UGV model, a problem that would be much harder to detect using the SCR simulator alone. Section V contains concluding remarks, including how limitations of our current tools will be addressed in future work.

II. Background

A. SCR tabular notation and toolset

In 1980, the Software Cost Reduction (SCR) tabular notation was introduced to specify software requirements precisely and unambiguously [11]. The SCR notation has two important benefits. First, software developers find models expressed in table format easy to understand. Second, the tabular notation scales—the large requirements models of practical systems, see, e.g., [11]–[13], can be concisely represented in tables. In 1996, a formal state machine semantics for SCR models was proposed [14]. This semantics defines an important SCR construct called *system modes*, 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 cyber physical systems; see, e.g., [13].

Based on the SCR formal semantics and the notion of modes, a large suite of tools called FOrmal Requirements Modeling and AnaLysis (FORMAL), an upgrade of the SCR toolset [10], is under development. This toolset includes a consistency checker for automatically detecting well-formedness errors (e.g., type errors, non-determinism, missing cases) in the model specification [14]; an invariant generator for automatically generating invariants from SCR specifications [15]; and a simulator for validating the formal model [10]. An important feature of the SCR simulator is its ability to check properties at each simulation step [10], [12]. Also integrated into the toolset are model checkers [12] and theorem provers [16], [17] for verifying properties of SCR models, and tools for automatic test generation [18] and source code synthesis [19].

B. Moded Scenarios Description

Although software developers may understand formal requirements models in the tabular format, they have difficulty creating these and other formal models. However, in our experience, when developers are presented with a model represented by tables, they can readily understand, modify, and extend the model. The challenge is to produce the initial model. Because many developers already use scenarios to specify requirements, our solution to this problem is to formalize scenarios and to synthesize a formal requirements model from these scenarios. To that end, we have developed a new technique for specifying scenarios called a Moded Scenarios Description [9], which contains a set of Event Sequence Charts (ESCs), a Mode Diagram, and a Scenario Constraint.[1] The ESCs look like MSCs [22], a popular notation many software practitioners use to specify requirements. A Mode Diagram uses system modes to provide a system-level abstraction for combining the ESCs. A Scenario Constraint restricts or adds to the required system behavior specified by the ESCs and the Mode Diagram; for example, it can define the initial values of state variables.

ESCs have a natural state machine semantics and are easy to change. Each ESC contains *entities* and a list of *event sequences*. The entities include the system and a set of environmental entities called *monitored* and *controlled entities*, each associated with one or more state variables. Each event sequence contains a *monitored event*, a value change in a monitored variable, followed by a set of value changes in terms (auxiliary variables) or controlled variables. A Mode Diagram contains modes and mode transitions. *Numeric labels* link modes and mode transitions in the Mode Diagram with event sequences in the ESCs.

Although ESCs and MSCs look very similar, they 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 single monitored event but all effects of the monitored event, each represented as a change in value of either a controlled or term variable. Thus a single event

[1] A large volume of research has been published on formalizing scenarios and scenario-based formal model synthesis; see, e.g., [20], [21]. Two unique features of our technique are the use of modes to link scenarios and of the Scenario Constraint to define assumptions, assertions, state invariants, etc. See [9] for a detailed discussion of related research.

in an ESC usually corresponds to a sequence of two or more messages in a MSC. ESCs and MSCs are also semantically different. In an ESC, an event and its immediate effects occur in a single step. In an MSC, an event and its effects may occur in several steps. Because ESCs are the basis for synthesizing SCR models which have been shown to scale to practical systems (see, e.g., [11]–[13]), we expect models synthesized from ESCs to scale better and to be easier to understand than models, such as Labeled Transition Diagrams, which have been synthesized from basic and hierarchical MSCs.

To provide a formal semantics for a Moded Scenarios Description, [9] defines entities, types, scenario state variables, events, and conditions. These are used in turn to define ESCs, a Mode Diagram, and a Scenario Constraint, the three components of a Moded Scenarios Description.

III. Formal Model Synthesis Tool

We have developed the following five-step method for synthesizing a formal SCR requirements model of a system $\Sigma = (S, \Theta, \rho)$ from a Moded Scenarios Description.

1) *Construct sets of state variables and values.* From information in the ESCs and Mode Diagram, construct the sets of state variables and values. Based on these sets, define a function mapping each state variable to its value set. The sets of variables and values together with this function provide the basis for constructing the system state, conditions, events, and the state set S of system Σ using the definitions in [14].
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 Scenario Constraint, define the initial state predicate Θ.
3) *Construct the system transform.* From information in the Mode Diagram and ESCs, construct the transform function ρ, the composition of a set of update functions. Each update function specifies how values of dependent variables—the mode class, controlled variables, and terms—change in response to a monitored event. Compute the update functions using the algorithms in [9].
4) *Define constants and specify assumptions, assertions, and other model constraints.* Based on the Scenario Constraint, construct the sets of constants, assumptions, and assertions, and any additional constraints, such as state invariants (represented in SCR as condition tables).
5) *Analyze and validate the model.* Apply analysis techniques and tools to detect defects in the specification of the model, such as type errors and non-determinism. Once such defects have been removed, other tools, such as simulators and verifiers, can be applied to further improve the quality of the model.

To implement the above method, a prototype formal model synthesis tool has been developed in Java. Docking Frames [23], an open source Java Swing docking framework, was used to create the panels that comprise the tool's graphical interface and the NetBeans Visual Library to support the drawing of Mode Diagrams. As input, the tool accepts a graphical representation of a Mode Diagram and one or more ESCs, as well as the individual components of the Scenario Constraint (e.g., initial values, assertions). From these, the tool constructs the components of a formal model: a set of type definitions, a mode class and its component modes, and sets of monitored variables, terms, and controlled variables; an initial state predicate; the system transform function; and sets of constants, assumptions, and assertions. The tool converts this information into an intermediate textual format and then into the XML representation used internally by the FORMAL toolset. This synthesized formal model can be viewed, edited, and analyzed using FORMAL.

Translation of a Mode Diagram and a set of ESCs into a formal model is straightforward—each set and function the tool constructs corresponds to a set or function in the SCR model. In addition, the synthesis tool tries to infer a type for each variable. For variables assigned boolean or numeric values in the ESCs, the tool makes the obvious inferences. If it infers that a variable is neither boolean nor numeric, the tool defines the variable type as enumerated and constructs its value set (which is likely to be incomplete). If a variable is assigned incompatible types (e.g., boolean and numeric), the tool classifies the variable type as unknown.

One limitation of the synthesis tool is that the ESCs generally lack information needed to construct a complete SCR model, and thus the model may not pass all consistency checks. However, the FORMAL toolset can be used to detect such defects. For example, if the specification does not assign an initial value to a variable, the toolset notifies the user of the missing value. The FORMAL toolset performs other checks, e.g., detects non-determinism in function definitions, that identify inconsistencies in the specification. In addition to supplying missing information identified by FORMAL, the user may need to modify parts of the synthesized specifications, such as completing the set of values in an enumerated type definition.

To evaluate the synthesis tool, we used scenarios in the form of ESCs to specify the required behavior of a UGV, currently under development by the U.S. Navy, which provides mission support services, such as cargo transport and explosive ordnance disposal (EOD). These tasks are similar: the UGV travels to the pickup location, loads the cargo (explosive), proceeds to the drop-off location, unloads the cargo (explosive), and, upon mission completion, returns home. The UGV autonomously plans and travels the routes between locations.

Figures 1 and 2 show two scenarios of the required UGV behavior, each represented as an ESC. Shown at the top of each figure are the entities of interest. Each ESC has a single System/Agent entity (shown in green). To the left of the System/Agent are the monitored entities (shown in blue), each associated with a monitored variable; and to the right are the controlled entities (shown in red), each associated with a controlled variable. In the **CargoTransport** scenario shown in Figure 1, the monitored entities are `OpCmd`, `ActualLoc`, and `CargoLoaded`?; the sole controlled entity is `DesiredLoc`. Not all system entities must be present in an ESC; for example,

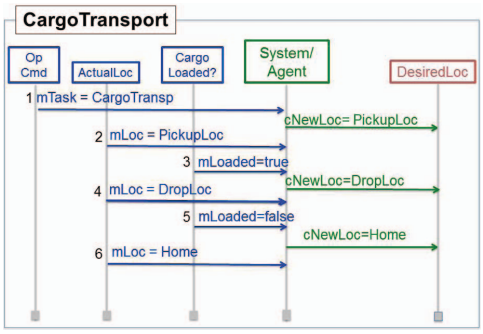

Fig. 1. ESC specifying cargo transport behavior during normal operation.

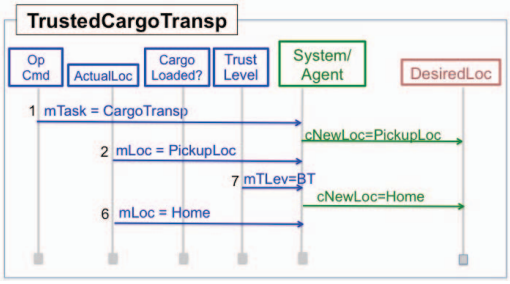

Fig. 2. ESC of cargo transport when trust is lost before cargo is loaded.

the entity `TrustLevel` is present in the **TrustedCargoTransp** scenario in Figure 2 but not in the scenario in Figure 1.

Numeric labels link the event sequences in the ESCs with the modes and transitions in the Mode Diagram. The UGV's Mode Diagram (see Figure 3) specifies the name **UGVmode** of the mode class (the set of possible modes), the four modes, all possible mode transitions, and the initial mode `Home`. In the Mode Diagram, each transition has exactly one label which links it with an event sequence in one or more ESCs. The transition is triggered when a monitored event with the same label occurs in the ESC. Each mode in the Mode Diagram may optionally have one or more labels; a label on a mode indicates that a monitored event with that label may occur, but if it occurs, the system mode does not change. In the scenarios specified by Figures 1–3, when the operator initiates the cargo transport task (event sequence 1), the mode changes from `Home` to `Load` (transition 1). For the scenario shown in Figures 1 and 3, when the UGV arrives at the pickup site (event sequence 2), the system remains in mode `Load` (labeled 2 in the Mode Diagram); and once the cargo is loaded (label 3), the mode changes to `Unload`.

Figure 1 shows the normal behavior of the UGV for cargo transport. In contrast, Figure 2 shows the system behavior when operator trust is low—if the UGV is at the pickup site, and operator trust falls below the trust threshold before the cargo is loaded, the UGV aborts the task and returns home. The complete set of scenarios for the cargo transport task includes ESCs that specify the system behavior in other cases in which trust is lost. ESCs are also needed to specify the behavior of the UGV in performing the explosive ordnance disposal task. In the normal case, the behavior of the UGV during EOD is the same as for cargo transport. When trust is

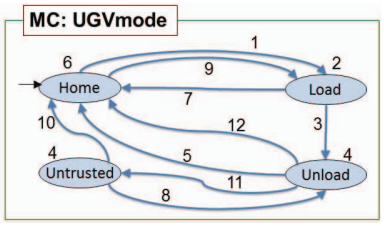

Fig. 3. Mode Diagram for UGV specification.

lost, however, the behavior of the UGV performing ordnance disposal may not always be the same as during cargo transport (see Section IV below).

The model synthesis tool is invoked in two steps. In the first step, the user uses the tool to construct graphical representations of the ESCs and a Mode Diagram and to specify the individual pieces of the Scenario Constraint (constants, variable initial values, assumptions, and assertions). Figure 4 shows the contents of the display after the user has invoked the synthesis tool to draw the Mode Diagram and three of the ESCs for the cargo transport task; the two ESCs at the top correspond to the ESCs shown in Figures 1 and 2. Initial values for some of the variables and a single assertion named `SafeEOD` have also been specified. In the second step, the user uses the tool to synthesize a formal model from the information entered into the GUI.

Figure 5 shows the synthesized UGV model as displayed by FORMAL. The top left window lists all models in open user projects. This example lists only a single model named UGV. The bottom left window lists all model components: the types, variables, condition tables, event tables, the name of the single mode class, any constants, the single mode transition table, a set of assertions (desired system properties), and a set of assumptions (constraints defined by the system environment). At the bottom of the display spanning the two rightmost columns is the output window which reports analysis results (e.g., of type and consistency checking). The top middle window displays the type dictionary: four types are shown, all enumerated. Displayed below that window is the mode class dictionary which lists all modes in the mode class. The bottom middle window contains the variable dictionary which lists the name, type, and, for terms and controlled variables, the type of table that defines it. Figure 5 lists only a single controlled variable `cNewLoc` and no term variables. The bottom right window displays the Assertions Dictionary which contains the single assertion `SafeEOD`.

In Figure 5, the top two windows on the right contain a mode transition table and an event table. The mode transition table shows how mode class `UGVmode` changes when a specified monitored variable changes. Each row in the table corresponds to a transition in the Mode Diagram of Figure 3, and the event in each row is the monitored event in the ESCs corresponding to the numeric label on the transition. For example, the first row in the table corresponds to the transition labeled 3. The event table describes how the value of controlled variable `cNewLoc` changes when a specified monitored variable changes in a given mode. The table has five

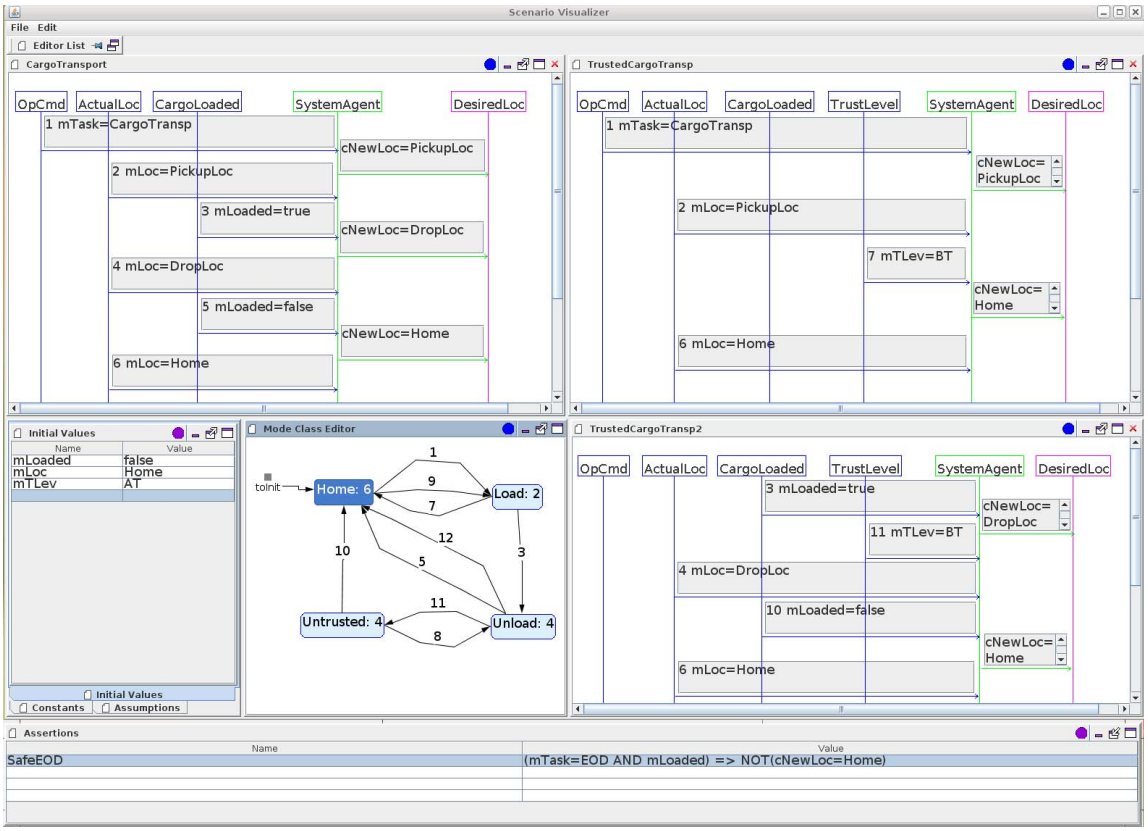

Fig. 4. Synthesis tool: User display of three ESCs, initial values of three monitored variables, the Mode Diagram, and an assertion named `SafeEOD`.

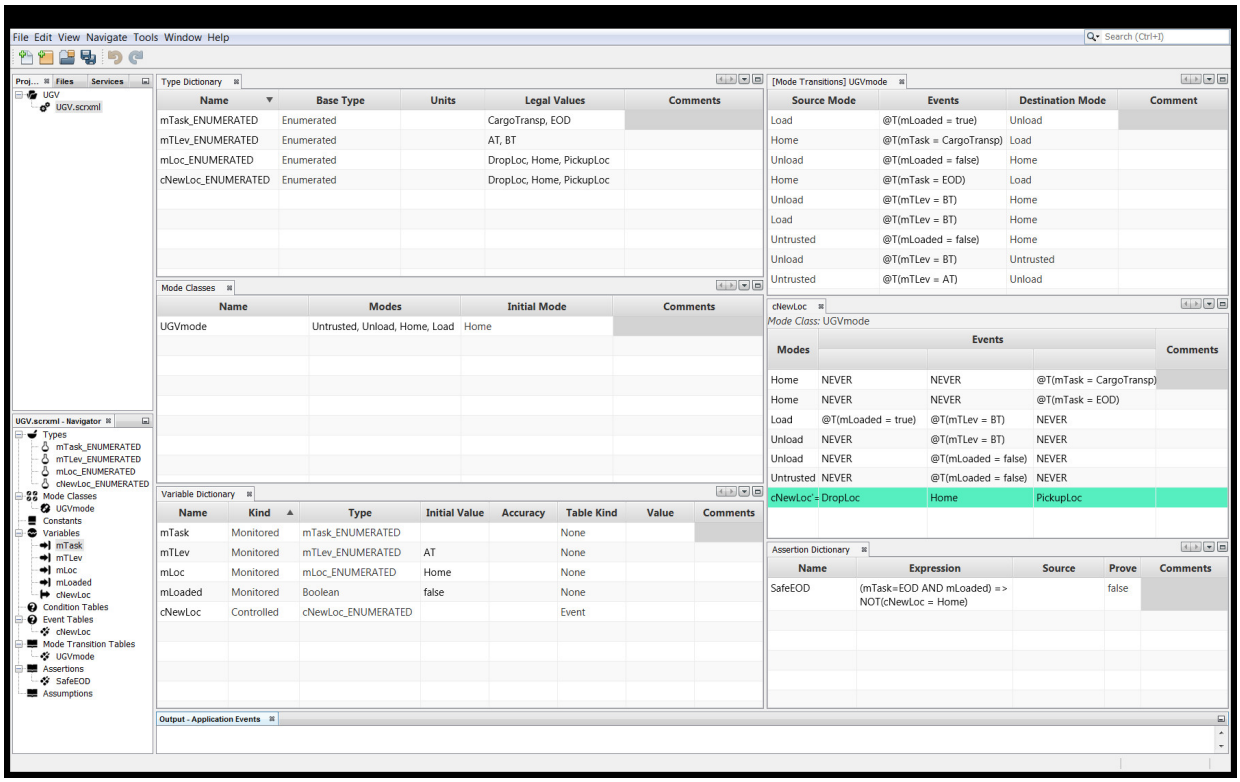

Fig. 5. FORMAL toolset: User display of the synthesized formal model.

columns; the first lists a mode, the next three columns contain monitored events, and the last is for comments. The final row of the table gives the new value assigned to `cNewLoc`. For each combination of mode in row i, non-`NEVER` event in (row i, column j), and assignment in column j, there is a corresponding event sequence in the ESCs. For example, the entry in the first row and last event column states that if the system is in mode `Home` and event `mTask` = `CargoTransp` occurs, then `cNewLoc`'s new value is `PickupLoc`. This corresponds to the event sequence labeled 1.

As expected, applying the consistency checker detects that initial values for `mTask` and `cNewLoc` are missing. Consistency checking also uncovers non-determinism in rows 5 and 8 of the mode transition table, arising from transitions 12 and 11 in the Mode Diagram. (Figure 7 below contains an ESC with an event sequence labeled 12, and Figure 4 an ESC using label 11.) A way to remove this non-determinism is to condition the event in row 8 on `mLoad` = `CargoTransp` and the event in row 5 on `mLoad` = `EOD`. To more realistically represent the UGV location, the user may replace the enumerated type with a set of variables that hold the UGV's x, y, z coordinates.

IV. Integrated 3D Simulation Tool

Once a model has been synthesized and any necessary corrections and extensions made, the user can run a simulator to validate the system behavior. To provide a powerful simulation capability for autonomous ground vehicles, the SCR simulator [10] has been integrated with eBotworks [24], a 3D simulator designed to support autonomy. Similar integrations can be performed with other domain-specific simulators to provide validation support for other classes of cyber physical systems. Like the SCR toolset, other toolsets for formal modeling and analysis include simulators with the ability to build customizable 2D GUIs (e.g., [25], [26]), but to the best of our knowledge, the integration described in this paper is the first between a requirements modeling tool and a 3D simulator.

eBotworks was designed to test and simulate autonomy software for command and control of unmanned systems, and includes autonomy packages for locomotion and path planning. Several choices of robot(s) are available, including the wheeled UGV that was used in our effort. To set up a simulation, the user constructs a world that includes landmarks, such as roads, and objects, such as packages.

In our integrated simulation tool, the SCR simulator is the system's cyber component, i.e., it plays the role of the controller used by the UGV's human operator, while the eBotworks simulator emulates the UGV and the UGV's environment. A custom GUI for the SCR UGV specification was designed that mimics how an operator might control a UGV via a tablet interface. The operator can click on buttons in the display to select a task and to indicate a loss of trust. The operator can also observe the values of other variables, such as the UGV's coordinates and whether the UGV is loaded, values which eBotworks changes during simulation. In addition to creating a custom GUI for the SCR simulator, we also created a world in eBotworks containing roads, a building, and a

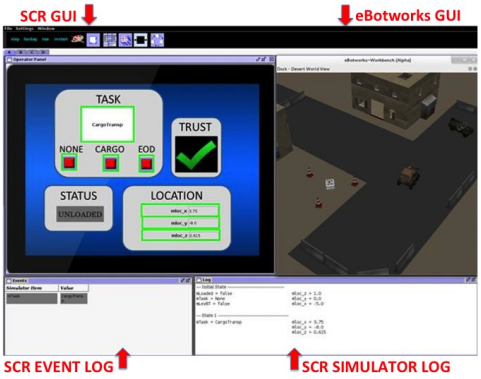

Fig. 6. Integrated Simulation: SCR + eBotworks.

single package which was either cargo or explosive ordnance. The user display of the integrated simulation tool, shown in Figure 6, shows the SCR 2D tablet GUI and the eBotworks simulation display at the top, and the monitored events log and simulator log displayed by the SCR simulator at the bottom.

Because the SCR simulator represents the UGV's controller, commands to begin a mission, e.g., `mTask` = `CargoTransp`, are initiated by the user via the SCR simulator interface and are sent to eBotworks. Once eBotworks receives a command, the UGV autonomously performs the task by taking a sequence of actions. For example, invoking the transport cargo task results in the action sequence: `goto-pickup-loc`, `load-cargo`, `goto-drop-off-loc`, `unload-cargo`. eBotworks communicates the completion of each action to the SCR simulator. Completed actions, which correspond to monitored events in SCR, are added to the SCR simulator's list of monitored events to be executed. After executing each new monitored event, the SCR simulator log is updated with newly calculated state information, and the simulator checks for and reports violations of any assumptions and assertions. For details of the integration of eBotworks with the SCR simulator, see [27].

Using the integrated simulation tool makes the UGV's behavior easier to understand than using the SCR simulator alone. In the ESC in Figure 7, the UGV performing EOD returns home when trust falls below the threshold. Because the new simulator displays continuous motion, the user can visually see that the vehicle is moving toward home with an explosive still loaded. Because this behavior violates the safety property `SafeEOD` (see the bottom of Figure 5), which states that if the task is explosive ordnance disposal and an explosive is loaded on the UGV, the UGV's new location is not its home base, the SCR simulator will alert the user of the property violation. If the safety property was not included in the model and the SCR simulator by itself was used, such dangerous behavior is much more difficult to detect; the user must recognize that the UGV is loaded, and, based on the UGV's coordinates, that the UGV is en route to its home base. This dangerous behavior corresponds to the event sequence and mode transition labeled 12 in Figures 7 and 3. Figure 8 contains a corrected ESC which eliminates this dangerous behavior. The Mode Diagram can be corrected simply by removing transition 12.

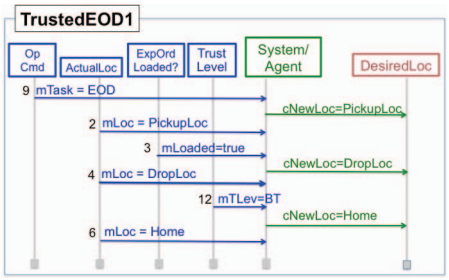

Fig. 7. ESC of EOD when operator trust is lost.

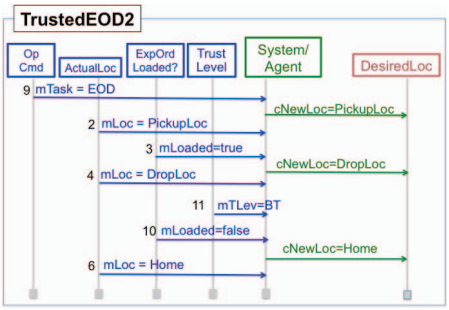

Fig. 8. Corrected ESC of EOD when trust is lost.

V. Concluding Remarks

The focus of most formal methods research on cyber physical systems has been formal verification. To exploit this research, two major gaps must be addressed: how to obtain formal models of these systems and how to validate them. This paper has introduced two tools designed to fill these gaps. Our next step in applying our method and tools is to use scenarios and our model synthesis tool to elicit and model system requirements of autonomous systems under development by the Navy, where there is significant interest in identifying safety hazards that such systems may encounter and understanding what actions the operator and the autonomous system should take to remove or mitigate such hazards. As part of this process, we plan to integrate the SCR simulator with existing application-specific simulators for those autonomous systems. Planned future research includes strengthening the type inference algorithm used in synthesizing formal models to, among other things, detect when two or more inferred enumerated types are identical and replace them by a single type; extending consistency checking of scenarios, e.g., when event sequences in different ESCs have the same label, their variable names and values must be identical; and developing an abstract interface to the SCR simulator to facilitate its easy integration with application-specific simulators.

Acknowledgment

This research is supported by the Office of Naval Research. We thank Carolyn Gasarch and Michael Thomas for developing the model synthesis tool; Valerie Chen for integrating eBotworks with the SCR simulator; and Knexis Research for providing eBotworks, in particular, Kellen Gillespie for answering questions about its installation and plugin framework. Finally, we thank Dustin Hoffman and James Swaine for their work on the new FORMAL toolset.

References

[1] A. L. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems." *Eur. J. Control*, vol. 18, no. 3, pp. 217–238, 2012.

[2] Defense Science Board, "The role of autonomy in DoD systems," Office of the Under Secretary of Defense for Acquisition, Technology and Logistics, Washington, DC, Tech. Rep., Jul. 2012.

[3] S. Sengupta, "U.S. border agency allows others to use its drones," *New York Times*, July 2013.

[4] U.S. Senate, "The future of drones in America: Law enforcement and privacy considerations." Tech. Rep. J-113-10, 2013.

[5] G. Palmer, A. Selwyn, and D. Zwillinger, "The "Trust V" - Building and measuring trust in autonomous systems," Raytheon Co., Tewksbury, MA, Tech. Rep., 2014.

[6] M. W. Floyd, M. Drinkwater, and D. W. Aha, "How much do you trust me? Learning a case-based model of inverse trust." in *22nd Internat. Conf. on Case-Based Reasoning*. Cork, Ireland: Springer, 2014.

[7] S. Chinchali *et al.*, "Towards formal synthesis of reactive controllers for dexterous robotic manipulation," in *IEEE Internat. Conf. on Robotics and Automation*. St. Paul, MN: IEEE, 2012.

[8] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, "Using formal verification to evaluate human-automation interaction: a review," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 43, no. 3, 2013.

[9] C. Heitmeyer, M. Pickett, E. Leonard, M. Archer, I. Ray, D. Aha, and J. G. Trafton, "Building high assurance human-centric decision systems," *Automated Software Engineering*, vol. 22, no. 2, 2015.

[10] C. L. Heitmeyer, M. Archer, R. Bharadwaj, and R. D. Jeffords, "Tools for constructing requirements specifications: The SCR toolset at the age of ten," *Comput. Syst. Sci. Eng.*, vol. 20, no. 1, 2005.

[11] K. L. Heninger, "Specifying software requirements for complex systems: New techniques and their application," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 1, pp. 2–13, Jan. 1980.

[12] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj, "Using abstraction and model checking to detect safety violations in requirements specifications," *IEEE Trans. Softw. Eng.*, vol. 24, no. 11, 1998.

[13] C. Heitmeyer and R. Jeffords, "Applying a formal requirements method to three NASA systems: Lessons learned," in *IEEE Aerospace Conf.*, Big Sky, MT, 2007.

[14] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Trans. on Softw. Eng. and Methodology*, vol. 5, no. 3, 1996.

[15] E. Leonard, M. Archer, C. Heitmeyer, and R. Jeffords, "Direct generation of invariants for reactive models," in *10th ACM/IEEE Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, 2012.

[16] M. Archer and C. Heitmeyer, "Human-style theorem proving using PVS," in *Theorem Proving in Higher Order Logics*, ser. LNCS. Springer, 1997, vol. 1275.

[17] R. D. Jeffords and C. L. Heitmeyer, "A strategy for efficiently verifying requirements," *SIGSOFT Softw. Eng. Notes*, vol. 28, Sep. 2003.

[18] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *7th ACM SIGSOFT Symp., Foundations of Software Eng.*, ser. LNCS, vol. 1687, Toulouse, FR, 1999.

[19] E. I. Leonard and C. L. Heitmeyer, "Program synthesis from formal requirements specifications using APTS," *Higher-Order and Symbolic Computation*, vol. 16, no. 1-2, 2003.

[20] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Trans. on Softw. Eng.*, vol. 29, no. 2, Feb. 2003.

[21] C. Damas, B. Lambeau, F. Roucoux, and A. van Lamsweerde, "Analyzing critical process models through behavior model synthesis," in *Proc. 31st Internat. Conf. on Software Eng. (ICSE)*, Vancouver, CAN, 2009.

[22] ITU, "Message Sequence Charts," Recommendation Z.120, Intern. Telecomm. Union, Standardization Sect., 1996.

[23] "Docking Frames," http://dock.javaforge.com/.

[24] Knexus, "eBotworks," http://www.knexusresearch.com/.

[25] IBM, "Statemate," http://www-03.ibm.com/software/products/en/ratistat.

[26] M. P. E. Heimdahl, M. Whalen, and J. Thompson, "NIMBUS: A tool for specification centered development," in *Proc. 11th IEEE Int'l Requirements Eng. Conf.*, 2003.

[27] E. I. Leonard, C. L. Heitmeyer, and V. Chen, "Integrating a formal requirements modeling simulator and an autonomy software simulator to validate the behavior of unmanned vehicles (WIP)," in *Proc. 2015 Spring Simulation Multi-Conference (SpringSim'15)*, 2015.