A Systematic Approach to Model Checking Human–Automation Interaction Using Task Analytic Models

Matthew L. Bolton, *Member, IEEE*, Radu I. Siminiceanu, and Ellen J. Bass, *Senior Member, IEEE*

***Abstract*—Formal methods are typically used in the analysis of complex system components that can be described as "automated" (digital circuits, devices, protocols, and software). Human–automation interaction has been linked to system failure, where problems stem from human operators interacting with an automated system via its controls and information displays. As part of the process of designing and analyzing human–automation interaction, human factors engineers use task analytic models to capture the descriptive and normative human operator behavior. In order to support the integration of task analyses into the formal verification of larger system models, we have developed the enhanced operator function model (EOFM) as an Extensible Markup Language-based, platform- and analysis-independent language for describing task analytic models. We present the formal syntax and semantics of the EOFM and an automated process for translating an instantiated EOFM into the model checking language Symbolic Analysis Laboratory. We present an evaluation of the scalability of the translation algorithm. We then present an automobile cruise control example to illustrate how an instantiated EOFM can be integrated into a larger system model that includes environmental features and the human operator's mission. The system model is verified using model checking in order to analyze a potentially hazardous situation related to the human–automation interaction.**

***Index Terms*—Formal methods, human–automation interaction, model checking, task analysis.**

I. Introduction

FAILURES in complex safety-critical systems are often not due to a single component, but rather system components interacting in unexpected ways. Formal methods researchers are developing languages, techniques, and tools capable of proving whether potentially dangerous interactions between concurrently executing system components exist. Formal methods are a set of languages and techniques for the modeling, specification, and verification of systems [1]. Model checking is an automated formal methods approach used to verify that a formal model of a system (usually of computer software or hardware) satisfies a set of desired properties (a specification) [2]. A formal model describes a system as a set of variables and transitions between variable valuations (states). Specification properties are typically represented in a temporal logic, usually linear temporal logic (LTL) or computation tree logic (see [3]), using the variables that describe the formal system model to construct propositions. Verification is the process of proving that the system meets the properties in the specification. Model checking performs this process automatically by exhaustively searching a system's state space in order to determine if these criteria hold. If there is a violation, an execution trace is produced (a counterexample). This counterexample depicts a model state (a valuation of the model's variables) corresponding to a specification violation along with a list of the incremental model states that led up to the violation.

Model checking has been successfully used to find problems in computer hardware and software applications (see [4]). However, one source of failures in complex safety-critical systems that is rarely considered in model checking analyses is the interaction between the human operator and other system components. For example, human–automation interaction has contributed to a number of system failures [5], including the crashes of American Airlines Flight 965 [6] and China Air 140 [7].

A. Task Analytic Models of Human Behavior

When designing the procedures, displays, controls, and training associated with the human-automation interaction (HAI) of an automated system, human factors engineers use task analytic methods to describe the normative human behaviors required to control the system automation [8]. The resulting task analytic models represent the mental and physical activities that operators use to achieve the goals that the system was designed to support. Such models are typically structured as a hierarchy, where activities decompose into other activities and (at the lowest level) atomic actions. Task analytic models, such as ConcurTaskTrees (CTT) [9], operator function model (OFM) [10], or User Action Notation (UAN) [11], can represent these hierarchies using discrete graph structures. In these models, strategic knowledge (condition

Manuscript received January 11, 2010; revised August 17, 2010; accepted November 26, 2010. Date of publication March 22, 2011; date of current version August 23, 2011. This work was supported in part by the National Library of Medicine under Grant T15LM009462, by the National Aeronautics and Space Administration (NASA) Cooperative Agreement NCC1002043, and by the NASA Contract NNA10DE79C. This paper was recommended by Associate Editor M. Dorneich.

M. L. Bolton is with the San Jose State University Research Foundation, National Aeronautics and Space Administration Ames Research Center, Moffet Field, CA 94043 USA (e-mail: mlb4b@virginia.edu).

R. I. Siminiceanu is with the National Institute of Aerospace, Hampton, VA 23666 USA (e-mail: radu@nianet.org).

E. J. Bass is with the Department of Systems and Information Engineering, University of Virginia, Charlottesville, VA 22904 USA (e-mail: ejb4n@virginia.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCA.2011.2109709

logic) controls when activities can execute. Modifiers control how activities or actions execute in relation to each other.

B. Formal Verification With Task Analytic Models

Because they can be represented discretely, task analytic models can be used to include human behavior in formal system models along with other system elements such as device automation, human–device interfaces, and the operational environment. Researchers have incorporated task analytic models into formal system models of human–automation interactive systems: either modeling task behavior natively in the formal notation [12]–[14] or translating task analytic representations, such as CTT [15]–[18], UAN [19], and other custom representations [20] into the formal notation. This allows analysts to verify system safety properties in light of the modeled normative human behavior and thereby prove that, assuming the representativeness of the system model, the system will always operate safely if the human operators adhere to the modeled behavior.

C. Limitations of Current Techniques

The power of these formal verification analyses is limited by the ability of the task analytic modeling notations to express normative human behavior. For example, CTT [9] and Fields' task modeling notation [20] do not support all of the temporal and cardinal relationships between activities and actions (referred to as subacts hereinafter) of other task analytic modeling notations. Considering the OFM [10], CTT [9], Goals, Operators, Methods, and Selection rules (GOMS) [21], UAN [22], and Fields' notation [20], all of the following relationships can be specified to control how activities or actions in a decomposition hierarchy execute [23].

1) One or more of the subacts must execute, one at a time, for the parent activity to finish.
2) One or more of the subacts must execute, and each subact can be executed concurrently.
3) All of the subacts must execute, one at a time.
4) All of the subacts must execute, possibly concurrently.
5) Exactly one subact must execute.
6) All of the subacts must execute, one at a time, in a specified order.
7) Zero or more of the subacts must execute, one at a time.
8) One or more of the subacts must execute, possibly concurrently.
9) All subacts must execute synchronously (at the same time).

Of the task modeling notations used in formal verification (and task analytic modeling techniques in general), only the UAN supports all of these relationships. However, unlike the other notations, the UAN assumes that it is modeling human interaction with a desktop computer and thus specifically models human actions as they relate to moving mouse cursors, clicking mouse buttons, and pressing keyboard keys. Thus, it is only applicable to a limited subset of human–automation interactive systems. Techniques that rely exclusively on formal modeling notations do not have this limitation [12]–[14]. However, these require that modelers manually implement task models using notations not intended for such a use.

Furthermore, of the task modeling techniques employed in formal verification analyses, only the CTT supports a graphical notation that can be used to represent modeled behavior visually, where visual notations are common for modeling paradigms (such as the GOMS or OFM) more typically used in HAI analyses. Such visualizations can be useful because they can facilitate the communication and comprehension of the modeled human behavior.

D. EOFM

The enhanced operator function model (EOFM) [23] was designed to be a generic task analytic modeling language that specifically addresses these issues. EOFM extends the OFM [10] which supports a visual and object-oriented means of representing task models. It uses state and derived variables to specify model behavior. It models goal-level behaviors as activities. Each activity may include conditions that describe under what conditions it can be undertaken. Activities are decomposed into lower level subactivities and, finally, actions. Operators on each decomposition specify how many subactivities or actions can execute and what the temporal relationship is between them. The EOFM standardizes the type of conditions that modify activities and supports all of the cardinalities and temporal orderings discussed earlier. The EOFM language is Extensible Markup Language (XML) based, thus making it platform independent and easy to parse. It also supports a visual notation where tasks are represented as treelike graphs [24].

E. Objectives

Because the EOFM does not exhibit the limitations of the other task analytic modeling notations discussed earlier, we would like to be able to exploit the expressive power of its notation such that the task analytic behavior models implemented in it could be considered in formal verifications of system safety properties. The research described in this paper refines the EOFM task analytic modeling language, its syntax, and its formal semantics introduced in [23] and shows how task behavior models represented using the EOFM can be incorporated into formal system models so that the impact of normative human behavior on system safety properties can be evaluated using formal verification with model checking. We first present the EOFM language, its syntax, and its formal semantics. We show how our automated EOFM-to-Symbolic-Analysis-Laboratory (SAL [25]) language translator uses this formal description to generate models that can be formally verified with the SAL symbolic model checker (SAL-SMC). Benchmarks are reported for models produced with this process. We then present an automobile driving case study illustrating how this process can be used to verify safety properties related to the use of the cruise control. Finally, we discuss the limitations of this technique and outline directions for future development.

II. EOFM

The EOFM language allows for the modeling of a human operator as an input/output system. Inputs may come from the human–device interface, environment, mission goals, and other human operators. Output variables are human actions.

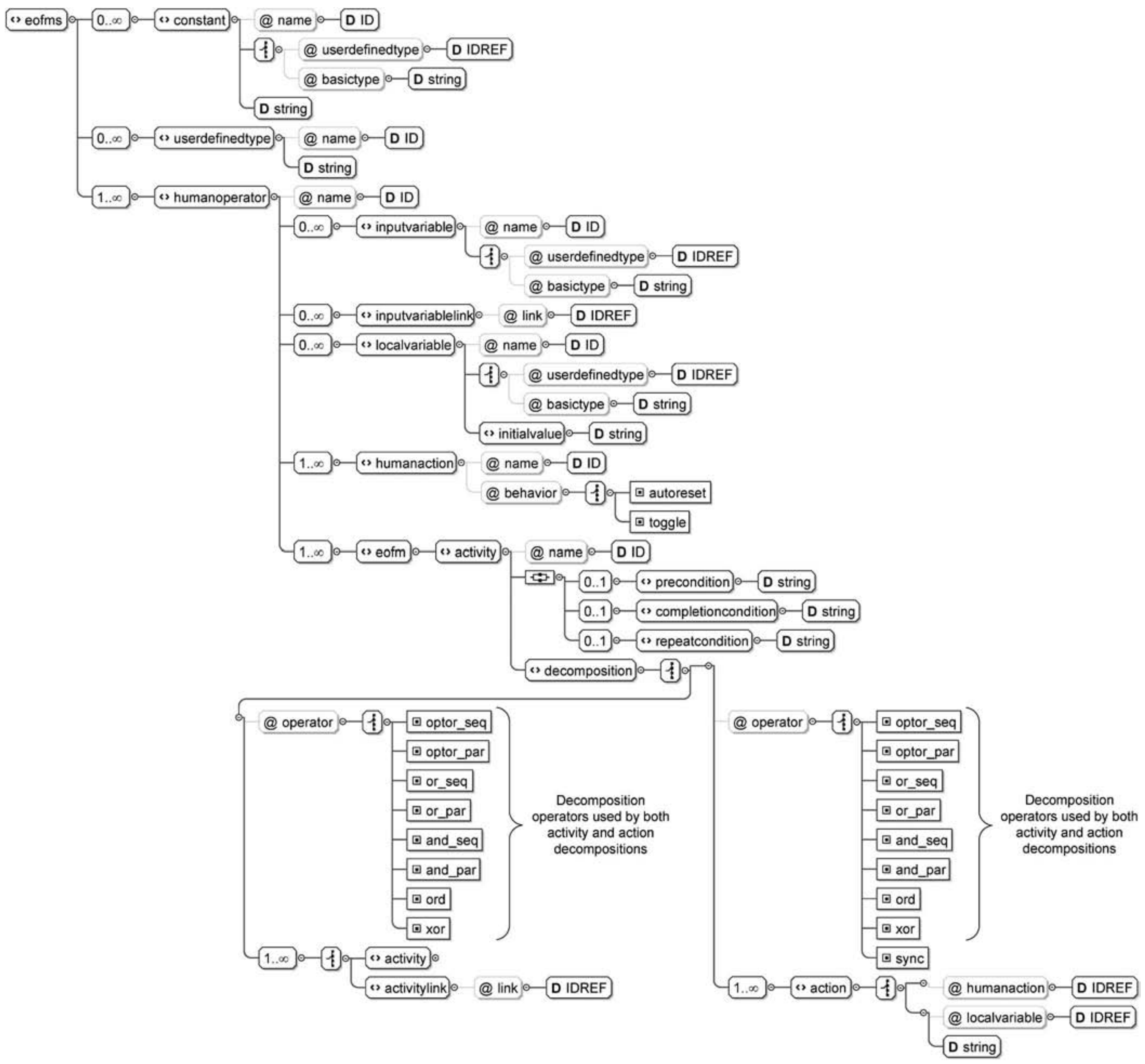

Fig. 1. Visualization [58] of the EOFM's RELAX NG language specification.

The operator's task model describes how human actions may be generated based on input and local variables (representing perceptual or cognitive processing).

Each human operator model is a set of EOFM task models that describe goal-level activities. Activities decompose into lower level activities and, eventually, atomic actions. Decompositions are controlled by decomposition operators that specify the cardinality of and temporal relationship between the sub-activities or actions. Activities can have preconditions, repeat conditions, and completion conditions (Boolean expressions written in terms of input, output, and local variables as well as constants) which specify what must be true before an activity can execute, when it can execute again, and what is true when it has completed the execution, respectively. Atomic actions are either an assignment to an output variable (indicating an action has been performed) or a local variable (representing a perceptual or cognitive action). All variables are defined in terms of constants, user-defined types, and basic types, described hereinafter.

A. Syntax

The EOFM language's XML syntax is defined using the REgular LAnguage for XML Next Generation (RELAX NG) standard [26]. The syntax (see Fig. 1) has been modified from [23] in order to support more standardized terminology and an XML structure that better represents the EOFM graphical notation.

XML documents contain a single root node whose attributes and subnodes define the document. For the EOFM specification, the root node is called *eofms*. The next level of the hierarchy has zero or more *constant* nodes, zero or more *userdefinedtype* nodes, and one or more *humanoperator* nodes. The *userdefinedtype* nodes define enumerated types useful for representing the operational environment, human–device interface, and human mission concepts. A *userdefinedtype* node is composed of a unique name attribute (by which it can be referenced) and a string of data representing the type construction (the syntax of which is application dependent). A *constant* node is defined by a unique name attribute, either a *userdefinedtype* attribute (the name attribute of a *userdefinedtype* node) or a *basictype* attribute.

The *humanoperator* nodes represent the task behavior of the different human operators. Each *humanoperator* has zero or more input variables (*inputvariable* nodes and *inputvariablelink* nodes for variables shared with other human operators), zero or more local variables (*localvariable* nodes), one or more human action output variables (*humanaction* nodes), and one or more task models (*eofm* nodes). A human action (a *humanaction* node) describes a single, observable, atomic act that a human operator can perform. A *humanaction* node is defined by a unique *name* attribute and a *behavior* attribute which can have one of two values: *autoreset* (for modeling a single discrete action such as flipping a switch) or *toggle* (for modeling an action that must be started and stopped as separate discrete events such as starting to hold down a button and then releasing it).

Input variables (*inputvariable* nodes) are composed of a unique name attribute and either a *userdefinedtype* or *basictype* attribute (defined as in the constant node). To support the definition of inputs that can be perceived concurrently by multiple human operators (for example, two human operators hearing the same alarm issued by an automated system), the *inputvariablelink* node allows a *humanoperator* node to access input variables defined in a different *humanoperator* node using the same input variable name. Local variables are represented by *localvariable* nodes, themselves defined with the same attributes as an *inputvariable* or *constant* node, with an additional subnode, *initialvalue*, which is a data string with the variable's default initial value.

The task behaviors of a human operator are defined using *eofm* nodes. One *eofm* node is defined for each goal-directed task behavior. The tasks are defined in terms of *activity* nodes. An *activity* node is represented by a unique *name* attribute, a set of optional conditions, and a *decomposition* node. Condition nodes contain a Boolean expression (in terms of variables and human actions) with a string that constrains the *activity*'s execution. The following conditions are supported:

1) *precondition*: criterion to start executing;
2) *repeatcondition*: criterion to repeat the execution;
3) *completioncondition*: criterion to complete the execution.

An activity's *decomposition* node is defined by a decomposition operator (an *operator* attribute) and a set of activities (*activity* or *activitylink* nodes) or actions (*action* nodes). The decomposition operator controls the cardinal and temporal execution relationships between the *subactivity* and *action* nodes (referred to as subacts). The EOFM language implements the following decomposition operators: *and*, *or*, *optor*, *xor*, *ord*, and *sync*. Each of these operators has two modalities: sequential (suffixed *_seq*) and parallel (suffixed *_par*) (see Table I). For the sequential mode, the subacts must be executed one at a time. In the parallel mode, the execution of the subacts may overlap in any manner. For the *xor*, *ord*, and *sync* decomposition operators, only one modality can be defined: *xor* and *ord* are always sequential, and *sync* is always parallel.

TABLE I
DECOMPOSITION OPERATORS

Operator		Modality	
Type	Semantics	Sequential	Parallel
and	All of the sub-acts must execute	*and_seq*	*and_par*
or	One or more of the sub-acts must execute	*or_seq*	*or_par*
optor	Zero or more of the sub-acts must execute	*optor_seq*	*optor_par*
xor	Exactly one sub-act must execute	*xor*	—
ord	All sub-acts must execute in the order they appear in	*ord*	—
sync	All sub-acts must be executed at the same time	—	*sync*

The *activity* nodes represent lower level subactivities and are defined identically to those higher in the hierarchy. Activity links (*activitylink* nodes) allow for the reuse of model structures by linking to existing activities via a *link* attribute which names the linked *activity* node.

The lowest level of the task model hierarchy is represented by either observable atomic human actions or internal (cognitive or perceptual) ones, all using the *action* node. For an observable human action, the name of a *humanaction* node is listed in the *humanaction* attribute. For an internal human action, the valuation of a local variable is specified by providing the name of the local variable in the *localvariable* attribute and the assigned value within the node itself.

B. EOFM Visualization

The structure of an instantiated EOFM's task behaviors can be represented visually as a treelike graph structure (examples appear in Figs. 12–14) where actions are represented by rectangular nodes and activities are represented by rounded rectangle nodes. In these representations, the conditions are connected to the activity that they modify: A *precondition* is represented by a yellow downward pointing triangle connected to the right side of the activity; a *completioncondition* is presented as a magenta upward pointing triangle connected to the left of the activity; and a *repeatcondition* is conveyed as a recursive arrow attached to the top of the activity. These standard colors are used for condition shapes to help distinguish them from each other and the other task structures. Decompositions are presented as arrows, labeled with the decomposition operator, extending below an activity that points to a large rounded rectangle containing the decomposed activities or actions.

C. EOFM Formal Semantics

We now formally describe the semantics of the EOFM language's task models: explicitly defining how and when each activity and action in a task structure executes.

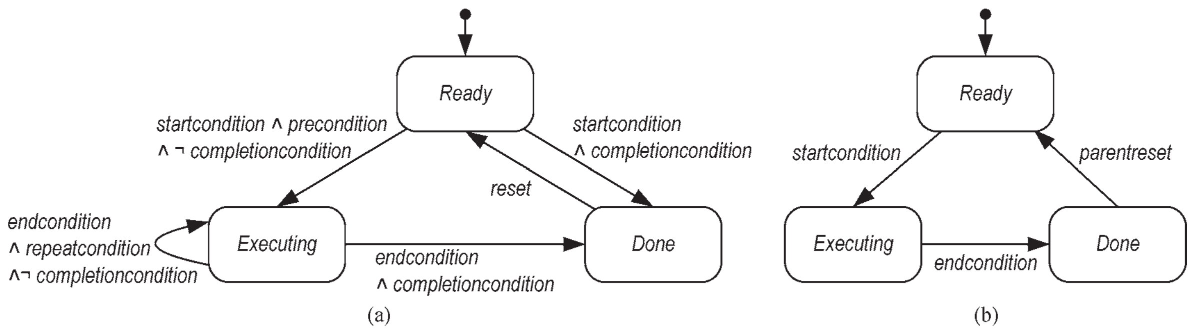
Ready
startcondition ∧ precondition ∧ ¬ completioncondition
startcondition ∧ completioncondition
reset
endcondition ∧ repeatcondition ∧¬ completioncondition
Executing
endcondition ∧ completioncondition
Done
(a)
Ready
startcondition
parentreset
Executing
endcondition
Done
(b)

Fig. 2. (a) Execution state transition diagram for a generic activity. (b) Execution state transition diagram for a generic action.

An activity's or action's execution is controlled by how it transitions between three discrete states.

1) *Ready*: The initial (inactive) state which indicates that the activity or action is waiting to execute.
2) *Executing*: The active state which indicates that the activity or action is executing.
3) *Done*: The secondary (inactive) state which indicates that the activity has finished executing.

While *preconditions*, *repeatconditions*, and *completionconditions* can be used to describe when activities and actions transition between these execution states, three additional conditions are required. These conditions support transitions based on the activity's or action's position in the task structure and the execution state of its parent, subacts (activities or actions into which the activity decomposes), and siblings (activities or actions contained within the same decomposition). These conditions are as follows:

1) *startcondition*: implicit condition that triggers the start of an activity or action defined in terms of the execution states of its parent and siblings;
2) *endcondition*: implicit condition to end the execution of an activity or action defined in terms of the execution state of its subacts;
3) *reset*: implicit condition to reset an activity (have it return to the *Ready* execution state).

For any given activity or action in a decomposition, a *startcondition* is composed of two conjuncts: one stipulating conditions on the execution state of its parent and the other stipulating conditions on the execution state of its siblings based on the parent's decomposition operator, generally formulated as

$$(parent.state = Executing) \wedge \bigwedge_{\forall siblings\, s} (s.state \neq Executing).$$

This is formulated differently in the following circumstances. If the parent's decomposition operator has a parallel modality, the second conjunct is eliminated. If the parent's decomposition operator is *ord*, the second conjunct is reformulated to impose restrictions only on the previous sibling in the decomposition order: $(prev\_sibling.state = Done)$. If it is the *xor* decomposition operator, the second conjunct is modified to enforce the condition that no other sibling can execute after one has finished:

$$\bigwedge_{\forall siblings\, s} (s.state = Ready).$$

An *endcondition* is also composed of two conjuncts both related to an activity's subacts. Since an action has no subacts, an action's *endcondition* defaults to *true*. The first conjunct asserts that the execution states of the activity's subacts satisfy the requirements stipulated by the activity's decomposition operator. The second asserts that none of the subacts are executing. This is generically expressed as follows:

$$\left(\bigoplus_{\forall subacts\, c} (c.state = Done)\right) \wedge \bigwedge_{\forall subacts\, c} (c.state \neq Executing).$$

In the first conjunct, $\bigoplus$ (a generic operator) is to be substituted with $\wedge$ if the activity has the *and_seq*, *and_par*, or *sync* decomposition operator and with $\vee$ if the activity has the *or_seq*, *or_par*, or *xor* decomposition operator. Since *optor_seq* and *optor_par* enforce no restrictions, the first conjunct is eliminated when the activity has either of these decomposition operators. When the activity has the *ord* decomposition operator, the first conjunct asserts that the last subact has executed.

The *reset* condition is *true* when an activity's or action's parent transitions from *Done* to *Ready* or from *Executing* to *Executing* when it repeats the execution. If the activity has no parent (i.e., it is at the top of the decomposition hierarchy), *reset* is *true* if that activity is in the *Done* execution state.

The *startcondition*, *endcondition*, and *reset* conditions are used with the *precondition*, *repeatcondition*, and *completioncondition* to define how an activity or action transitions between the execution states. This is presented in Fig. 2 where the states are represented as nodes (rounded rectangles) and the transitions are represented as arcs. Guards are attached to each arc.

The transition criteria for an activity [see Fig. 2(a)] are described in more detail in the following list.

1) An activity is initially in the inactive state, *Ready*. If the *startcondition* and *precondition* are satisfied and the *completioncondition* is not, then the activity can transition to the *Executing* state. However, if the *startcondition* and *completioncondition* are satisfied, the activity moves directly to *Done*.
2) In the *Executing* state, an activity will repeat execution when its *endcondition* is satisfied as long as its *repeatcondition* is true and its *completioncondition* is not. An activity transitions from *Executing* to *Done* when both the *endcondition* and *completioncondition* are satisfied.

3) An activity will remain in the *Done* state until its *reset* condition is satisfied, where it returns to the *Ready* state.

The transition criterion for an action is simpler [see Fig. 2(b)] since an action cannot have a *precondition*, *completioncondition*, or *repeatcondition*. Note that, because actions do not have any subacts, their *endcondition*s are always *true*.

III. EOFM to SAL Translation

To be utilized in a model checking verification, instantiated EOFMs must be translated into a model checking language. We use the formal semantics to translate instantiated EOFMs into the language of the SAL. The SAL is a framework for combining different tools to calculate properties of concurrent systems [25], [27]. The SAL language is designed for specifying concurrent systems in a compositional way. Constants and types are defined globally. Discrete system components are represented as modules. Each module is defined in terms of input, output, and local variables. Modules are linked by their input and output variables. Within a module, local and output variables are given initial values. All subsequent value changes occur as a result of transitions. A transition is composed of a guard and a transition assignment. The guard is a Boolean expression composed of input, output, and local variables as well as the SAL's mathematical operators. The transition assignment defines how the values of the local and output variables change when the guard is satisfied. The SAL language is supported by a tool suite containing several state-of-the-art model checkers including SAL-SMC. Auxiliary tools include a simulator, a deadlock checker, and an automated test case generator.

The EOFM-to-SAL translation is automated by our custom-built Java program which uses the Document Object Model [28] to parse an instantiated EOFM's XML code and convert it into a SAL code.

For a given instantiated EOFM, the translator defines the SAL constants and types using the *constant* and *userdefinedtype* nodes. The translator creates a separate SAL module for each *humanoperator* node. Input, local, and output variables are defined in each module corresponding to the *humanoperator* node's *inputvariable*, *localvariable*, and *humanaction* nodes, respectively. Input and local variables are defined in the SAL using the *name* and type (*basictype* or *userdefinedtype*) attributes from their markup. Local variables are initialized to their values from the markup. All output variables in the SAL module (one for each *humanaction* node) are defined with a Boolean type and initialized to *false*: A value of *true* indicates that the action is being performed.

The translator defines the following two Boolean variables in the *humanoperator* node's module to handle a coordination handshake with the human–device interface module (see [29] and [30]):

1) an input variable *interfaceReady* that is *true* when the interface is ready to receive input;
2) an output variable *actionsSubmitted* that is *true* when one or more human actions are performed.

The *actionsSubmitted* output variable is initialized to *false*.

The translator defines a SAL type, *ActivityState*, to represent the execution states of activities and actions: *Ready*, *Executing*, or *Done* (see Fig. 2). As described previously, the activity and action state transactions define the task (see Fig. 2). Each *activity* and *action* in the human operator's node structure has an associated local variable of the type *ActivityState*. For activities, in addition to the task-model-defined *precondition*s, *repeatcondition*s, and *completioncondition*s, three other conditions are required to define the transition guards: *startcondition*s, *endcondition*s, and *reset*. The translator defines the *startcondition*s based on the execution state of its parent activity, the state of the activity's siblings, and its parent's decomposition operator. The *startcondition* for an activity is *true* when the parent is *Executing*, when the activity is *Ready*, and when any of the following conditions are true.

1) The decomposition operator is *ord*, and all preceding siblings are *Done* (it is the next activity to execute).
2) The decomposition operator is *xor*, and all siblings are *Ready* (it is the only activity to execute).
3) The decomposition operator is *and_seq*, *or_seq*, *optor_seq*, or *sync*, and no other siblings are *Executing* (the activity cannot execute when its siblings execute).

The *endcondition* for an activity is true when the activity is *Executing* and when any of the following conditions are true.

1) The decomposition operator is *and_seq*, *and_par*, or *sync*, and all child activities or actions are *Done*.
2) The decomposition operator is *ord*, and the last child activity or action is *Done*.
3) The decomposition operator is *or_seq*, *or_par*, or *xor*, and all child activities or actions are not *Executing* with at least one being *Done*.
4) The decomposition operator is *optor_seq* or *optor_par*, and all child activities or actions are not *Executing*.

Because the *reset* occurs when an *activity*'s parent resets, the *reset* transition is handled differently than the others. When a parent *activity* transitions from *Executing* to *Executing*, its subacts' execution state variables (and all activities and actions lower in the hierarchy) are assigned the *Ready* state. Additionally, for each *activity* at the top of a task hierarchy, a guard is created that checks if its execution state variable is *Done*. Then, in the transition assignment, this variable is assigned a value of *Ready* along with the lower level activities and actions in order to achieve the desired *reset* behavior.

Transitions between execution states for variables associated with *action* nodes are handled differently due to the coordination handshake. For each *action*, a *startcondition* is created using execution state variables and written as a guard for the *Ready* to *Executing* transition [see Fig. 2(b)] with the additional condition that *interfaceReady* is *true*. In the transition assignment, the execution variable associated with the given *action* is set to *Executing*, the corresponding *humanaction* output variable is set to the logical negation of its value (*true* or *false*, where the change in the variable value indicates that a human action has been performed), and *actionsSubmitted* is set to *true*. Because the *endcondition* for all *action*s is always true, the *Executing* to *Done* transition is handled by a single guard and transition assignment, where the guard accounts

TABLE II
BENCHMARK EXPERIMENT RESULTS

Decomposition Operator	# Interleavings	# Subacts(n) 2 # States	2 Time (s)	8 # States	8 Time (s)	16 # States	16 Time (s)	24 # States	24 Time (s)
and_seq	$n!$	20	0.15	2,564	0.51	1,179,652	1.82	436,207,620	5.77
or_seq	$2^n - 1$	24	0.15	3,072	0.51	1,310,720	1.85	469,762,048	6.20
optor_seq	2^n	26	0.14	3,074	0.50	1,310,722	1.76	469,762,050	6.43
xor	n	16	0.11	52	0.32	100	1.07	148	3.05
ord	1	14	0.13	38	0.46	70	1.16	102	2.62
and_par	$n!$	22	0.12	13,126	0.39	86,093,446	1.75	564,859,072,966	21.48
or_par	$2^n - 1$	26	0.13	13,634	0.41	86,224,514	1.56	564,892,627,394	145.52
optor_par	2^n	28	0.13	13,636	0.39	86,224,516	1.91	564,892,627,396	76.42
sync	1	10	0.11	10	0.21	10	0.37	10	0.67

for the handshake protocol. Thus, the guard specifies that the *actionsSubmitted* is *true* and that the *interfaceReady* is *false*: verifying that the interface has received submitted *humanaction* outputs. In the assignment, *actionsSubmitted* is set to *true*, any execution state variable associated with an *action* that has the value *Executing* is set to *Done* (it is unchanged otherwise), and any *humanaction* output variables that support the *autoreset* behavior are set to *false*. The *reset* transition occurs as a result of the activity *reset* process discussed earlier.

Because all of the transitions are nondeterministic, multiple activities can be executed independent of each other when *_par* decomposition operators are used. Multiple human actions resulting from such decompositions are treated as if they occur at the same time if the associated *humanaction* output variables change during the same interval (a sequential set of model transitions) when *interfaceReady* is *true*.

IV. BENCHMARKS

A variety of tests was run to validate that the translator was generating a SAL code that adhered to the EOFM's formal semantics (see [31]). To evaluate the complexity and scalability of the EOFM task models, we generated EOFM models and investigated the translated models' state spaces and runtimes using the SAL.

The EOFM models include a single human operator who presses keys on a human–device interface. The *autoreset* behavior was used for all key press actions. The human operator had a single goal-level activity that decomposed into two or more actions via a decomposition operator.

To ensure that the verification process would search a model's entire state space, we created an LTL specification that would not produce a counterexample. This stated that the model could never have an execution state variable indicating that a specific action is executing (*action1State*) without its corresponding human action output (*PressKey1*) being true

$$\mathbf{G}\left(\neg(action1State = actExecuting \wedge PressKey1 \neq true)\right).$$

Using the SAL-SMC on a single-core 2.8-GHz machine with 8 GB of RAM running the Ubuntu Linux, we ran nine benchmarking trials. Table II shows the size of the state space and runtimes for the entire set of singleton operators. The results are consistent with the interleavings of the actions associated with the decomposition operators. For example, there is one more interleaving allowed for *optor* than for *or*.[1] Also, the number of synchronous actions executed at the same time has no impact on the size of the state space.

In terms of scalability, the symbolic model checker can handle large state spaces (over half a trillion for some of the parallel operators). However, the size of the state space, the time to check the LTL query, and the type of decomposition operator interact to impact the runtime. With the *and_par*, *or_par*, and *optor_par* decomposition operators, the state spaces are roughly of the same size; however, the runtimes vary by a factor of seven between *and_par* and *or_par*.

V. APPLICATION: CRUISE CONTROL SYSTEM

To illustrate how an instantiated EOFM can be used to find human–automation interaction-related problems, we present a formal model of driving with a simple automobile cruise control system (see Fig. 3) in which a car is traveling down a street toward a traffic light. The distance of the car from the light is represented in discrete intervals corresponding to its relative position: Very Very Far, Very Far, Far, Merging, Close, Very Close, and At Intersection. At the Merging interval, a ramp intersects with the road, allowing any traffic on the ramp to merge.

The driver of the car wants to drive safely down the road. The driver starts at an initial speed with his foot on an accelerator pedal. His goal is to drive at his desired speed while avoiding merging traffic and safely responding to traffic lights. He can drive at one of three speeds (Slow, Moderate, or Fast) by manipulating the pressure applied to the accelerator pedal. Pressing the accelerator pedal causes the car to accelerate to the next faster speed. Decreasing the pressure on the accelerator pedal decelerates the car to the next slower speed. The driver can release the accelerator pedal (which causes the car to decelerate two speeds per interval until it stops) and can press the brake (which stops the car in a single interval). The driver can enable or disable the cruise control using available buttons. Cruise control can also be disabled by pressing the brake. When enabled, the cruise control will keep the car moving forward at its current constant speed unless the driver presses

[1]The number of reachable states is multiplied by two because each interleaving produces an additional reachable state where the key is pressed and the corresponding action is first *executing* and then transitions to *done*.

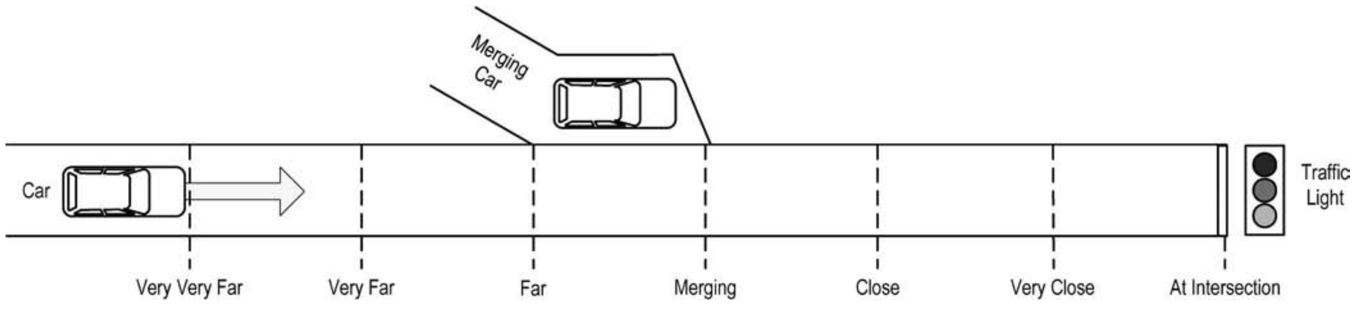

Fig. 3. Cruise control model scenario.

the accelerator pedal. In this situation, the driver controls the car's speed above the cruise speed.

The formal system model architecture [29], [32] includes the human–device interface, device automation, operational environment, and human mission. It also contains the human task behavior which is translated from an instantiated EOFM for the driving tasks: driving at the desired speed, avoiding merging traffic, and responding to the light. Formal verification is performed on this system model using the SAL-SMC to ensure that the driver will never run a red light.

At each distance interval, the driver can perform a single action, or a synchronous set of actions, in response to the information that he is receiving from the rest of the model. Once these actions have been committed, the human–device interface, device automation, and environments update: The light is allowed to change, merging traffic may arrive on the ramp, and the car advances one interval down the road if it has not stopped. We discuss each component of this system model, how each is modeled formally, and the instantiated EOFM.

A. Human–Device Interface

The human–device interface receives actions from the human operator and provides feedback from the device automation. Through the human–device interface, the driver can press the accelerator pedal (*PressAccelerator*), decrease the pressure on the accelerator pedal (*EaseOffAccelerator*), release the accelerator pedal (*ReleaseAccelerator*), press the brake (*Brake*),[2] press the enable cruise control button (*EnableCruise*), and press the disable cruise control button (*DisableCruise*). The driver receives information as inputs from the human–device interface: the position of the pedal (*Accelerator*), the speed of the car (*CarSpeed*), and whether the car has accelerated, decelerated, or remained at a constant speed (*CarAcceleration*).

The driver can directly control the state of the human–device interface's accelerator pedal (see Fig. 4).[3] The pedal has four different states (*Unpressed*, *PressedToSlow*, *PressedToModerate*, and *PressedToFast*) representing its position. These directly correspond to a car speed (*Stopped*, *Slow*, *Moderate*, or *Fast*, respectively). The initial state of the pedal (*PressedToSlow*, *PressedToModerate*, or *PressedToFast*) determines the initial speed of the car. If the human operator presses the accelerator pedal, the state of the pedal is set to the next higher state. If he decreases the pressure on the pedal, its state is set to the next lower state. The pedal transitions to the *Unpressed* state when the operator releases the pedal.

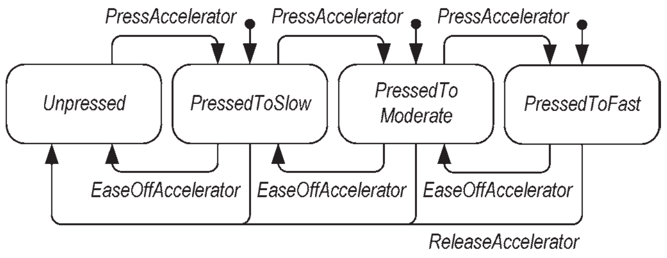

Fig. 4. State transition diagram depicting the formal model's behavior for the accelerator pedal (*Accelerator*).

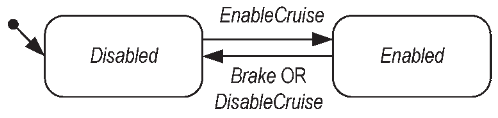

Fig. 5. State transition diagram depicting the formal model's behavior for the state of the cruise control (*Cruise*).

B. Device Automation

The device automation tracks the status of the cruise control and manages the car's speed and acceleration.

Cruise (see Fig. 5) is *Enabled* when the driver presses the enable cruise button and *Disabled* when he presses the disable cruise button or presses the brake.

Enabling the cruise control also sets the cruise speed in the device's automation (see Fig. 6) where the cruise speed directly corresponds to the speed of the car when the cruise is enabled.

The human operator input to the accelerator pedal and the state of the cruise control also impact how the device automation controls the speed of the car (see Fig. 7): a property visible to the human operator via the speedometer. The driver can increase the car's speed by pressing the accelerator pedal even when the cruise control is enabled. The application of the brake stops the car. When the driver decreases the pressure on the accelerator pedal (without applying the brake), the car's speed decreases when *Cruise* is *Disabled* or when *Cruise* is *Enabled* and the *CruiseSpeed* is below the car's current speed. When the driver removes his foot from the accelerator pedal (as indicated by the *Accelerator* being *Unpressed*), the car slows

[2]The model assumes that the human operator will never press the break and accelerator pedal at the same time.

[3]Figs. 4–11 represent the formal models of system elements as nondeterministic finite state transition diagrams [33], [34]. States are depicted as rounded rectangles. Arcs with Boolean expressions attached represent guards on transitions between states. Arrows with dots point to valid initial states. In actual practice, these models were represented in the notation of the SAL.

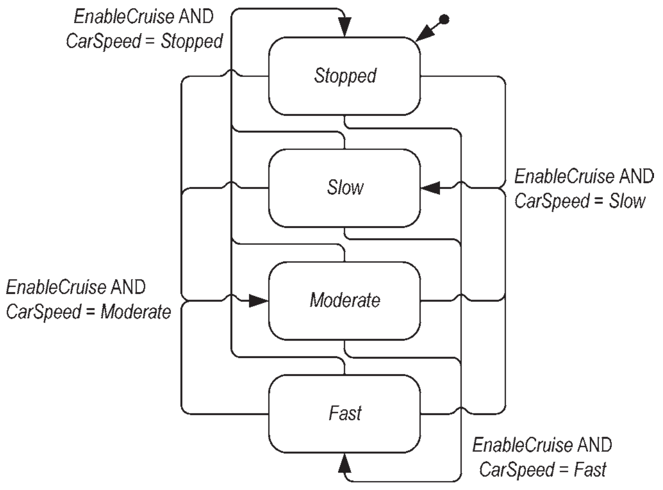
Stopped
Slow
Moderate
Fast
EnableCruise AND CarSpeed = Stopped
EnableCruise AND CarSpeed = Slow
EnableCruise AND CarSpeed = Moderate
EnableCruise AND CarSpeed = Fast

Fig. 6. State transition diagram depicting the formal model's behavior for the state of the cruise speed (*CruiseSpeed*).

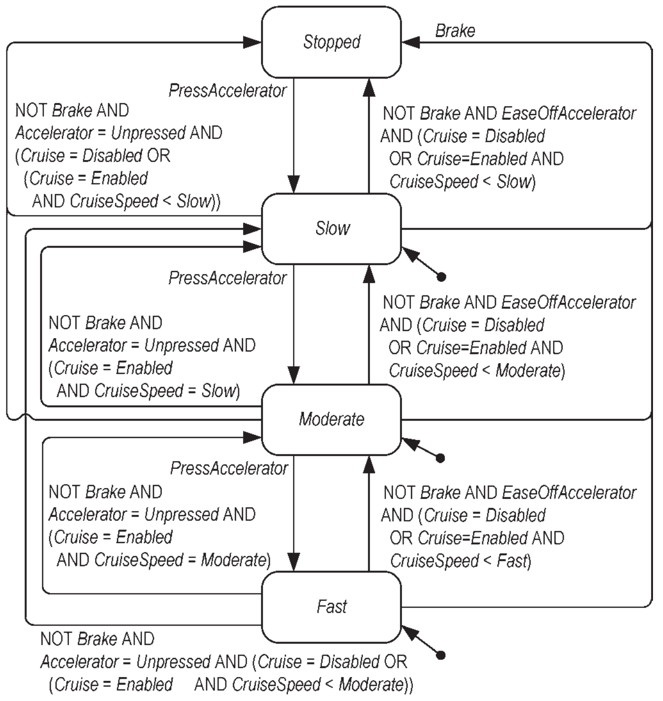
Stopped
Slow
Moderate
Fast
Brake
PressAccelerator
NOT Brake AND Accelerator = Unpressed AND (Cruise = Disabled OR (Cruise = Enabled AND CruiseSpeed < Slow))
NOT Brake AND EaseOffAccelerator AND (Cruise = Disabled OR Cruise=Enabled AND CruiseSpeed < Slow)
NOT Brake AND Accelerator = Unpressed AND (Cruise = Enabled AND CruiseSpeed = Slow)
NOT Brake AND EaseOffAccelerator AND (Cruise = Disabled OR Cruise=Enabled AND CruiseSpeed < Moderate)
NOT Brake AND Accelerator = Unpressed AND (Cruise = Enabled AND CruiseSpeed = Moderate)
NOT Brake AND EaseOffAccelerator AND (Cruise = Disabled OR Cruise=Enabled AND CruiseSpeed < Fast)
NOT Brake AND Accelerator = Unpressed AND (Cruise = Disabled OR (Cruise = Enabled AND CruiseSpeed < Moderate))

Fig. 7. State transition diagram depicting the formal model's behavior for the state of the car's speed (*CarSpeed*).

by two speed increments below the current speed unless *Cruise* is *Enabled* at an intermediate speed. In this case, the car will remain at or slow down to the *CruiseSpeed*.

The state of the car's acceleration (see Fig. 8) is tied to the changes in the state of the car's speed. If the car's speed has decreased, then the car has *Decelerated*. If the car's speed has increased, then the car has *Accelerated*. If there has been no change in the car's speed, the car has remained at a *ConstantSpeed*.

C. Operational Environment

The environment model encompasses the state of the traffic light, the state of merging traffic, and the relative position of the car to the traffic light.

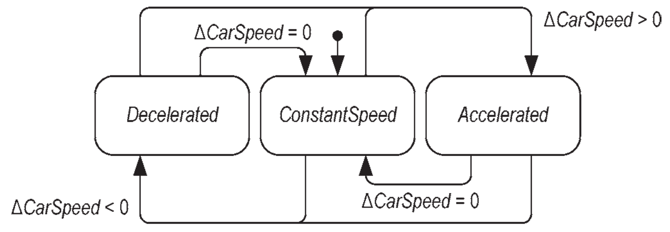

Fig. 8. State transition diagram depicting the formal model's behavior for the state of the car's acceleration (*CarAcceleration*).

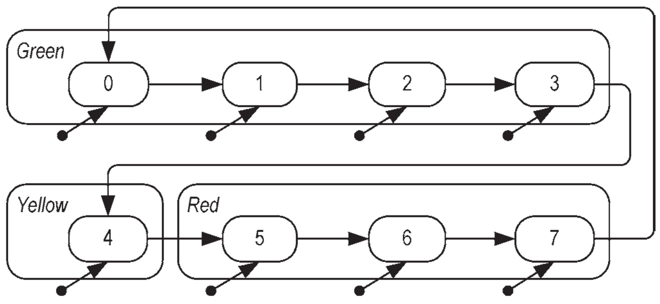

Fig. 9. State transition diagram depicting the formal model's behavior for the traffic light (*TrafficLight*).

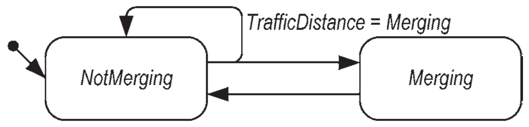

Fig. 10. State transition diagram depicting the formal model's behavior for the presence of merging traffic (*Traffic*).

The state of the traffic light changes between its three colors in response to a modulo 8 counter (see Fig. 9). The light is *Green* when the counter is between zero and three, *Yellow* when the counter is four, and *Red* when the counter is between five and seven. At every step in the model's execution, when the environment model is allowed to update, the light counter increments.

When the car is at the *Merging* interval, there may be merging traffic based on the transition logic in Fig. 10. Initially, there is no merging traffic. When the car is at the *Merging* interval, there can either be no merging traffic or a single merging car. The *Merging* state will automatically transition back to *NotMerging* when the environmental model next updates.

The environment model tracks the relative distance, or the interval, of the car from the traffic light (see Fig. 11). The car starts at the *VeryVeryFar* interval and proceeds through the intervals every time the environmental model updates if the car is not stopped. Once the *AtIntersection* interval is reached, the entire system model is at its final state.

D. Human Mission

The human mission model controls how fast the human operator wants to drive via the *MissionSpeed* variable. It is initialized to *Slow*, *Moderate*, or *Fast*.

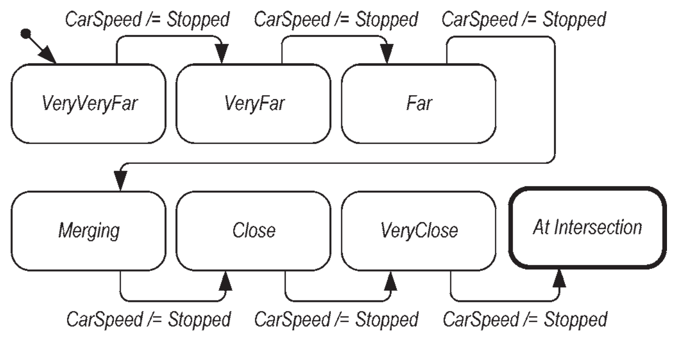

Fig. 11. State transition diagram depicting the formal model's behavior for the relative distance/interval of the car to the traffic light (*TrafficLightDistance*). A thick black line around a node indicates a final state.

E. Human Task Behavior Model

An instantiated EOFM describes the human driver task behavior using a single *humanoperator*. This human operator driver has access to input variables from the following:

1) the environment model: the traffic light's color (*TrafficLight*) and its relative interval distance (*TrafficLightDistance*);
2) the device automation and human–device interface models: the car's speed (*CarSpeed*), the car's acceleration (*CarAcceleration*), and the state of the accelerator pedal (*Accelerator*);
3) the mission model: mission speed (*MissionSpeed*).

The driver model generates *humanaction* outputs representing actions performed through the human–device interface: pressing the accelerator pedal (*PressAccelerator*), decreasing the pressure on the accelerator pedal (*EaseOffAccelerator*), releasing the accelerator pedal (*ReleaseAccelerator*), making no change to the accelerator pedal (*NoChange*), pressing the brake (*Brake*), enabling the cruise control (*EnableCruise*), and disabling the cruise control (*DisableCruise*).

Three goal-directed task models were constructed[4]: one for driving at the desired speed (see Fig. 12), one for avoiding merging traffic (see Fig. 13), and one for responding to the light (see Fig. 14).

The task model for driving at the desired speed (see Fig. 12) has the root activity *aDriveAtDesiredSpeed*. This has both a *precondition* and a *repeatcondition* that control when it can execute and when it can repeat the execution: when there is no merging traffic and either the traffic light is green or the car is not close to the intersection. The execution will stop (*completioncondition*) if the traffic light is not green and the car is close to the traffic light or if traffic is merging.

The driver performs this activity using one or more methods (the *or_seq* decomposition operator): accelerating (*aAccelerate*), decelerating (*aDecelerate*), or maintaining the current speed (*aMaintainSpeed*). Accelerating can be performed if the car's speed is below the mission speed. This activity completes when the car reaches or surpasses the mission speed or if the operator must respond to the light or merging traffic. The driver accelerates by pressing the accelerator pedal (the *ord* decomposition operator indicates a sequential order, but with one action, only the one action is performed).

[4]A full code listing can be found at http://cog.sys.virginia.edu/formalmethods/ccND.xml.

The driver can start or continue decelerating if the car's speed is greater than the desired (mission) speed. The activity completes when the car's speed is less than or equal to the desired speed or the driver needs to respond to the light or traffic. Deceleration is accomplished by one of two activities (the *xor* decomposition operator): decreasing the pressure on the accelerator pedal (*aEaseOffAccelerator*) or disabling the cruise control (*aDisableCruise*). If the accelerator pedal is pressed, the driver can decrease the pressure on the pedal. The driver can disable the cruise control if it is enabled: The pedal is not pressed and the car has not just decelerated.

The driver can maintain speed as long as the car's speed matches the desired speed. The activity completes if the car's speed does not match the desired speed or the driver needs to respond to the light or traffic. The driver maintains speed by either enabling the cruise (*aCruise*) or holding the current speed (*aHoldSpeed*). If the cruise control does not appear to be enabled (indicated by the pedal being pressed), it can be enabled by synchronously performing (the *sync* decomposition operator) the actions for enabling the cruise control and releasing the accelerator pedal. The driver knows that the cruise control has been enabled if the car does not decelerate when he removes his foot from the accelerator pedal. The driver holds the current speed by making no change to the accelerator pedal.

The driver can avoid merging traffic (*aAvoidMergingTraffic*; see Fig. 13) when traffic is merging by performing one of two activities: If the car is not at its minimum speed, the driver can let the merging traffic go in front (*aLetCarGoInFront*), or if the car is not at its maximum speed, the driver can let the merging traffic in behind (*aLetCarGoBehind*). Letting the traffic pull in front is achieved by: 1) decreasing the pressure on the accelerator pedal (via *aEaseOffAccelerator*) if the cruise is not enabled or 2) disabling the cruise (via *aDisableCruise*) if it is enabled. Letting the traffic in behind is achieved by pressing the accelerator pedal.

By performing one of three activities, the driver responds to the traffic light (*aRespondToLight* in Fig. 14) if the traffic light is not green and the vehicle is close to it: waiting to respond to the light until the car is closer (*aWaitTillCloser*), performing a brake stop (*aBrakeStop*), or performing a roll stop (*aRollStop*). If the traffic light is close, the driver can wait until the light is closer by making no change to the car's accelerator pedal.

If the traffic light is not green, the car is very close to the light, and the car has not decelerated, the driver can quickly slow by braking.

If the traffic light is close and the car is going fast, the driver can let the car slowly roll to a stop by first initiating a roll (*aInitiateRoll*) and then stopping at the intersection (*aStopAtIntersection*). He initiates a roll by either releasing the accelerator pedal (via *aReleaseAccelerator*) if the cruise is not enabled or disabling the cruise (via *aDisableCruise*) if it is.

The driver no longer considers stopping at the intersection when the light is green. Otherwise, the driver performs a brake stop (via *aBrakeStop*) if the car has not decelerated or continues to roll to a stop if it has.

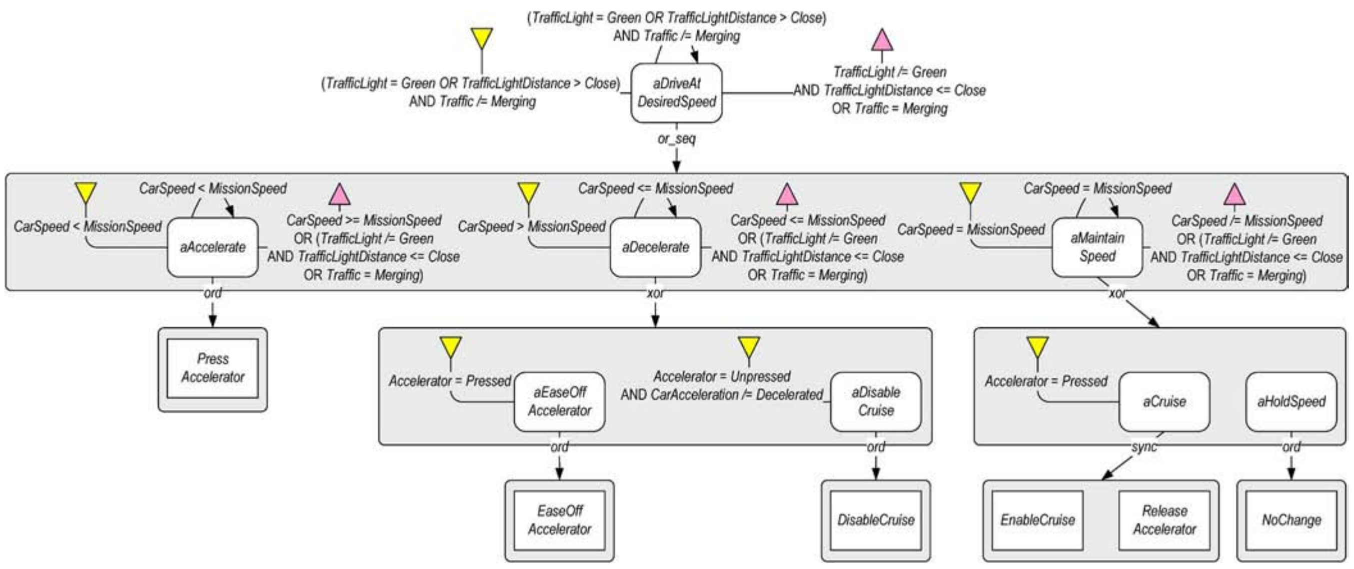
(TrafficLight = Green OR TrafficLightDistance > Close) AND Traffic /= Merging
aDriveAt DesiredSpeed
TrafficLight /= Green AND TrafficLightDistance <= Close OR Traffic = Merging
or_seq
CarSpeed < MissionSpeed
aAccelerate
CarSpeed >= MissionSpeed OR (TrafficLight /= Green AND TrafficLightDistance <= Close OR Traffic = Merging)
CarSpeed <= MissionSpeed
CarSpeed > MissionSpeed
aDecelerate
CarSpeed <= MissionSpeed OR (TrafficLight /= Green AND TrafficLightDistance <= Close OR Traffic = Merging)
CarSpeed = MissionSpeed
aMaintain Speed
CarSpeed /= MissionSpeed OR (TrafficLight /= Green AND TrafficLightDistance <= Close OR Traffic = Merging)
ord
xor
Press Accelerator
Accelerator = Pressed
aEaseOff Accelerator
Accelerator = Unpressed AND CarAcceleration /= Decelerated
aDisable Cruise
aCruise
aHoldSpeed
sync
EaseOff Accelerator
DisableCruise
EnableCruise
Release Accelerator
NoChange

Fig. 12. Visualization of the EOFM task model for driving at the desired speed. Activities are represented as rounded rectangles, actions as unrounded rectangles, preconditions as inverted yellow triangles, and completion conditions as magenta triangles. Boolean expressions in conditions use the syntax supported by transition guards in the SAL.

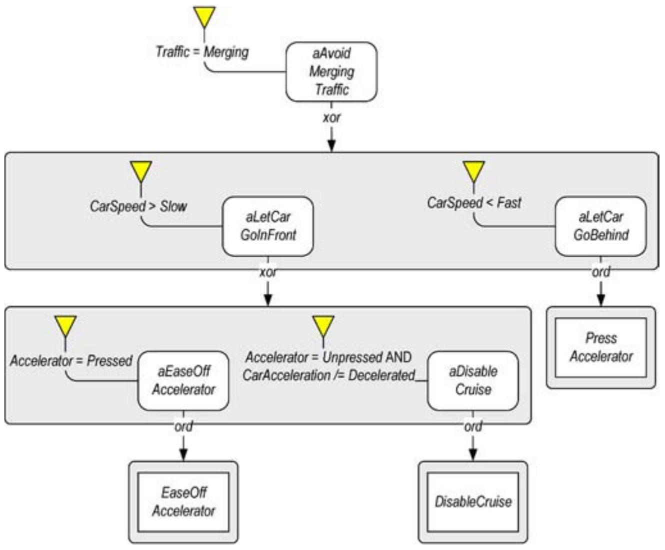

Fig. 13. Visualization of the EOFM for responding to merging traffic.

F. EOFM to SAL Translation

The EOFM instance was translated into a SAL code and incorporated into the larger formal system model so that it can be considered in the formal verification analyses of the system. In its original XML form, the human task behavior model was represented in 168 lines of code (43 lines were devoted to closing the XML tags). The translated SAL version of the model was represented in 439 lines of code (2.6 times more lines of code than the XML EOFM language representation).

G. Specification and Verification

To ensure safety, we wanted to use formal verification to check that the car would never reach the intersection while moving when the traffic light was red. This was represented in LTL as follows:

$$\mathbf{G}\neg\left(\begin{array}{l} TrafficLightDistance = AtIntersection \\ \wedge\, Car \neq Stopped \\ \wedge\, TrafficLight = Red \end{array}\right). \quad (1)$$

The attempt to verify this specification using the SAL-SMC on a workstation with a 3.0-GHz dual-core Intel Xeon processor and 16 GB of RAM running the Ubuntu 9.04 desktop resulted in a counterexample illustrating a violation. This occurred as follows.

1) The car starts *VeryVeryFar* from the traffic light going *Slow* with a *ConstantSpeed* acceleration. The pedal is *PressedToSlow*, and the cruise is *Disabled*. The traffic light is *Green*, and the driver wants to maintain a *Moderate* speed.
2) Because the current speed is too slow, the driver increases his speed by pressing the accelerator pedal (*PressAccelerator* via the *aDriveAtDesiredSpeed* and *aAccelerate* activities; see Fig. 12). The car proceeds to the *VeryFar* interval, having *Accelerated* to a *Moderate* speed.
3) Now, at his desired speed, the driver engages the cruise control (synchronously performing *EnableCruise* and *ReleaseAccelerator* via the *aDriveAtDesiredSpeed*, *aMaintainSpeed*, and *aCruise* activities; see Fig. 12). The car thus proceeds to the *Far* position at a *Moderate ConstantSpeed*.
4) The driver makes no changes to the speed of the car (*NoChange* via the *aDriveAtDesiredSpeed*, *aMaintainSpeed*, and *aHoldSpeed* activities). The car proceeds to the *Merging* position at a *Moderate ConstantSpeed*, where traffic is attempting to merge.
5) The driver lets the traffic in behind by pressing the accelerator pedal (*PressAccelerator* via the *aAvoidMergingTraffic* and *aLetCarGoBehind* activities; see Fig. 13). The

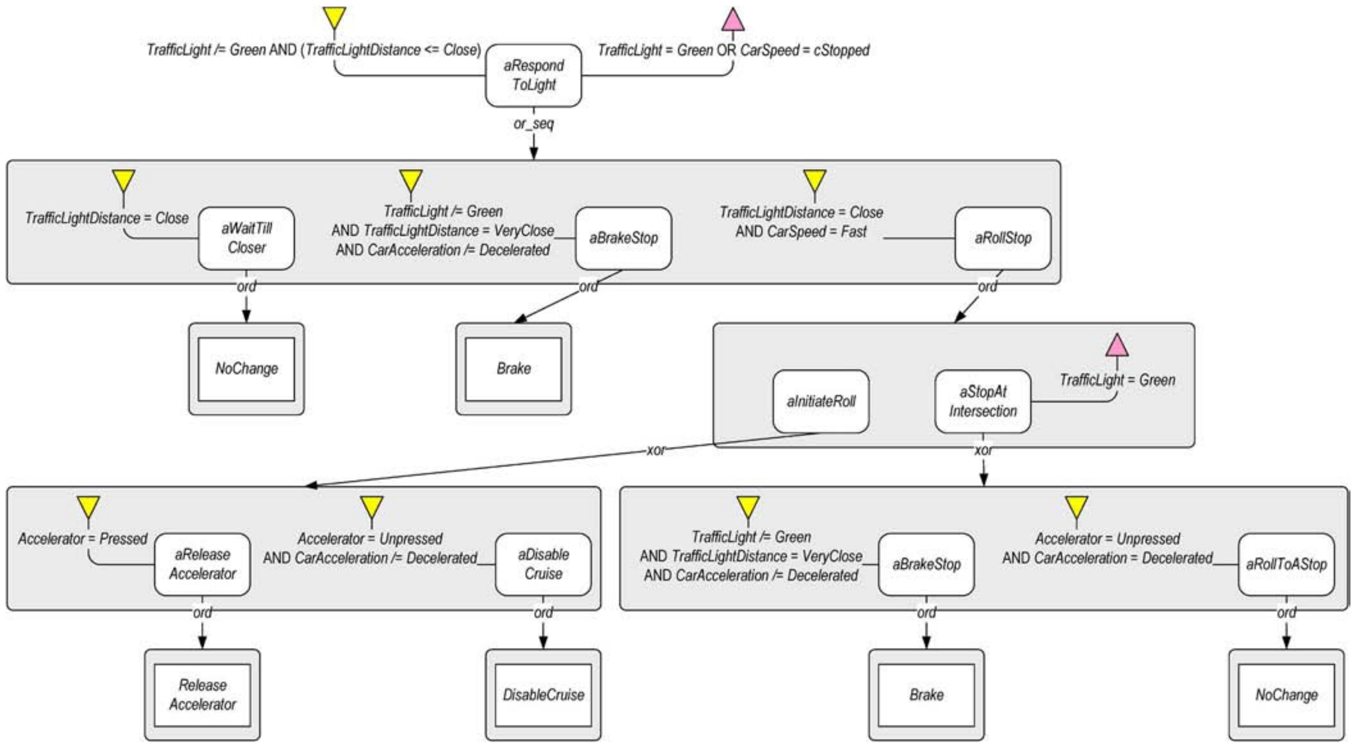
TrafficLight /= Green AND (TrafficLightDistance <= Close)
aRespond ToLight
TrafficLight = Green OR CarSpeed = cStopped
or_seq
TrafficLightDistance = Close
aWaitTill Closer
TrafficLight /= Green AND TrafficLightDistance = VeryClose AND CarAcceleration /= Decelerated
aBrakeStop
TrafficLightDistance = Close AND CarSpeed = Fast
aRollStop
ord
NoChange
Brake
aInitiateRoll
aStopAt Intersection
TrafficLight = Green
xor
Accelerator = Pressed
aRelease Accelerator
Accelerator = Unpressed AND CarAcceleration /= Decelerated
aDisable Cruise
TrafficLight /= Green AND TrafficLightDistance = VeryClose AND CarAcceleration /= Decelerated
aBrakeStop
Accelerator = Unpressed AND CarAcceleration = Decelerated
aRollToAStop
Release Accelerator
DisableCruise
Brake
NoChange

Fig. 14. Visualization of the EOFM for responding to a traffic light.

car advances to the *Close* interval, having *Accelerated* to the *Fast* speed. The traffic light turns *Yellow*.

6) The driver responds to the light by releasing his foot from the accelerator to perform a roll stop (*ReleaseAccelerator* via the *aRespondToLight*, *aRollStop*, *aInitiateRoll*, and *aReleaseAccelerator* activities; see Fig. 14). The car proceeds to the *VeryClose* interval, having *Decelerated* to the *Slow* speed. The light turns *Red*.
7) Having felt the car decelerate when he released the accelerator pedal, the driver attempts to continue rolling to a stop at the intersection by making no change (*NoChange* via the *aRespondToLight*, *aRollStop*, *aStopAtIntersection*, and *aRollToAStop* activities; see Fig. 14). However, the car has reached its cruising speed (*Moderate*) and has continued to the *AtIntersection* interval at a *Moderate ConstantSpeed*.

The specification has been violated with the car reaching the intersection without having stopped when the light is red. Thus, this example shows how normative human behavior modeled using the EOFM can be used in a larger process to find violations of system safety properties using formal verification with model checking.

H. Redesign

A potential explanation for the aforementioned problem is that the driver does not remember that the cruise control is engaged and therefore cannot properly perform a roll stop. In the current implementation, the only way that the operator can tell that the cruise control is engaged is if the accelerator pedal is not pressed and the car is not decelerating. We can use our method to explore potential design interventions that may correct this problem.

The car designer can add an indicator light with the state of the cruise control. In the formal model, this can be represented by a new variable (*Cruising*) which indicates whether cruise control is *Enabled* or *Disabled*. The driver's task model then changes to check this variable before executing a roll stop. To model this change, a *Cruising inputvariable* node is added to the driver's *humanoperator* node. In the *aDriveAtDesiredSpeed* activity (see Fig. 12), the *precondition*s for *aEaseOffAccelerator* and *aCruise* check that cruising is *Disabled*, and the *precondition* for *aDisableCruise* checks that the cruise is *Enabled*. In the *aAvoidMergingTraffic* (see Fig. 13) activity, the *precondition*s on *aEaseOffAccelerator* and *aDisableCruise* check that the cruise is *Disabled* and *Enabled*, respectively. Finally, in the *aRespondToLight* (see Fig. 14) activity, the *precondition*s to *aReleaseAccelerator* and *aDisableCruise* check that cruise is *Disabled* and *Enabled*, respectively. The decomposition of *aDisableCruise* also changes to the *sync* decomposition operator to support the actions for disabling the cruise control and releasing the accelerator pedal.

An EOFM instantiation modified to reflect these changes was translated into the SAL and incorporated into the compatibly modified system model. When the SAL-SMC was run using this new model and the specification in (1), it verified to true. Thus, the minor modifications to the human–device interface and the human task behavior model eliminated the potential human–automation interaction problem discovered in the previous verification.

VI. Discussion and Future Work

The EOFM task modeling language is defined formally, and its formal description has been used to construct a translator to support model checking. Translated models can be incorporated into larger system models so that they can be formally verified. The driving application system presented in this paper is simple, but it demonstrates how our EOFM task behavior modeling language and formal verification process can be used to discover potentially dangerous problems related to human–automation interaction.

Despite this success, our process is restricted to what systems it can be applied. These restrictions can be attributed to both the limitations of formal verification with model checking as well as the EOFM language. We discuss both of these in the following sections. We first examine how state space complexity and formalisms limit what types of systems can be evaluated with model checking. We then describe the limitation of the EOFM modeling notation as it relates to other task analytic modeling techniques and the types of analyses that it can facilitate. We recommend directions that future work can take to address the identified issues.

A. Limitations of Model Checking

1) Model Complexity: One of the challenges facing model checking verification is the state explosion problem. As the complexity of the modeled system increases, the memory and time required to store the combinatorially expanding state space can easily exceed the available resources [2]. One way that this has been addressed is through the use of extremely efficient means of representing large state spaces. This is utilized in symbolic model checking [35]. Other techniques allow select portions of the state space to be searched without compromising the accuracy of the verification, such as partial order reduction [36], symmetry reduction [37], abstract interpretation [38], and counterexample-guided abstraction refinement [39].

The SAL-SMC utilizes symbolic model checking. However, even with this efficiency, the size of the models that can be evaluated with our method is limited. For example, Bolton and Bass [30] found that an abstracted model of a patient controlled analgesia pump (a system containing large ranges of numerical integer data) represented a reasonable upper bound on the complexity of system models that could be verified using a high-end desktop computer. Future work will investigate whether other techniques for combating state space complexity can be incorporated into our method.

However, determining what the actual limits are on the size of the system that can be evaluated using our technique can be difficult given the influence of the modeled human task behavior on the state space complexity. The reported benchmarks highlight the tradeoffs associated with modeling different temporal orderings of human behavior within a given task. Furthermore, Bolton and Bass [30] indicate that, rather than add to the complexity of formal system models, the formal representation of the human task models can reduce model complexity for systems that depend on human–automation interaction by constraining the reachable state space to that associated with modeled human behavior. This suggests that the decomposition operators associated with higher model complexity in Table II would simply fail to constrain the system model as tightly as the less complex decomposition operators. Future work will investigate the frequency that each decomposition operator is used in task analytic modeling and in what contexts they are most appropriate. This will help provide a better understanding of what types of systems are best suited for the analysis presented in this paper.

There may be potential for reducing the complexity of the translated EOFM models. In the current implementation, the formal model represents every transitional step of an activity's execution state as a single model transition in the formal model. While this accurately adheres to the formal semantics of the language, it requires that the formal model have a discrete state for every intermediate transition in the task model hierarchy. These intermediary transitions are of no consequence to the other elements of the system model, which are only concerned with the resulting human actions. Thus, the complexity of the formal task model representation could be reduced by decreasing the number of internal transitions required to traverse an instantiated EOFM's translated task structure during execution. Future work will investigate the feasibility of this, as such an efficiency would allow for our method to be applied to more systems.

2) Formal Notation Expressiveness: A second major limitation of model checking is the expressive power of its modeling formalisms. Traditional model checking is applied to systems that can be modeled with discrete variables. However, complex systems can have continuous quantities. The field of hybrid systems has been developed to address this issue [40]. However, current techniques can only handle system models with about a half-dozen continuous variables. As such, continuous quantities are often modeled abstractly using discrete representations: like how velocity, distance, and acceleration were modeled in our driving application.

With respect to the modeling of time, discrete-state models can be augmented with a set of clocks [41]. While this technique can be used to model clock synchronization problems in digital systems, only very simple models can be fully verified.

Future work should investigate how our method can be adapted to the formal verification of hybrid systems.

B. Limitation of the EOFM Language

While the EOFM supports a superset of the cardinal and temporal relationships found in other task analytic modeling paradigms, the EOFM lacks some features that could make it even more expressive.

1) Task Priority and Interruption: One feature that is supported by the UAN and CTT is the ability for activities to be interrupted when a higher priority activity becomes relevant due to a change in the human–device interface or environmental conditions. For example, a driver already dealing with a traffic light may need to immediately respond to traffic. After the higher priority activity has been addressed, the UAN and CTT provide a means for restarting, resuming, or abandoning any interrupted activities. Future work will investigate how such a feature can be incorporated into the EOFM.

2) Root Activity Execution: The current EOFM implementation assumes that all root activities are temporally related with the equivalent of an *optor_seq* decomposition operator. Future work should investigate under what conditions or for which task domains, if any, additional options are necessary.

3) Cognitive Modeling: The EOFM currently only supports a simple model of cognitive and perceptual operations using local variables and thus does not utilize more realistic models of human memory, perception, or cognitive processing. Furthermore, the EOFM assumes that the human operator has access to all environmental and human–device interface information that is available at any given time, ignoring limitations on perception, attention, and memory, and the scanning process that humans often use for acquiring information in dynamic environments. The presented driving application (as well as other applications presented in [29]–[31] and [42]) shows that insights into the safety and design of systems can be obtained using human behavior models that exhibit these limitations. However, the limitation of our approach can also be seen in the presented example, where the discovered problem occurred because the model assumes that the driver will not remember that he engaged the cruise control. While local variables would allow the memory of this to be modeled, it would have prevented the discovery of the problem using formal verification. Cognitive modeling offers possibilities for addressing these types of limitations.

Lee and Sanquist's Operator Function Model with Cognitive Operations [43] extended the OFM to support cognitive operations related to the following: perceptual sensitivity, perceptual discrimination, selective attention, distributed attention, working memory, long-term memory, and response precision. They have shown how this can be used to identify design and training requirements to prevent erroneous human behavior and facilitate efficient human work.

Within the formal modeling literature, the work by Blandford, Curzon, and colleagues has focused on creating programmable user models (PUMs) [44] that capture the knowledge and cognitively plausible behavior that an operator might use when interacting with an automated system and implementing them as part of a formal system model [45]–[47]. PUMs encompass the goals that the operator wishes to achieve with the system, his beliefs and knowledge about the operation of the system, the information available to him from the human–device interface, and the actions that he can execute to interact with the system. Thus, the operator model must use its knowledge about the system with the currently available information to select actions that will fulfill its goals. PUMs and formal verification have been used to identify cognitively plausible errors based on a human operator model interacting with an automated system. These include repeating actions, omitting actions, committing actions too early, replacing one action with another, performing one or more actions out of order, and performing an unrelated action [48]. Means of identifying postcompletion errors (where the operator forgets to perform actions that occur after the completion of a high-level goal) have also been identified and illustrated using a system model of a vending machine [49]. PUMs have also been extended so that formal verification can investigate how humans might make errors due to issues related to salience, cognitive load, and cognitive interpretation of spatial cues [50], [51].

The GOMS family of models [21] supports the timing analysis of human task behavior based on cognitive modeling. Rukšėnas *et al.* [51] have shown how these types of analyses can be coupled with formal verification analyses with formal PUMs.

Future work should investigate how to couple the features of these cognitive models with the EOFM so that they can be factored into the formal verifications that the EOFM supports.

4) Multioperator Scalability: Although the EOFM has been designed to support multiple operators, to date, it has only been used to model single operator systems [23], [29], [30], [32], [42], [52]. There are a number of systems that depend on multiple human operators interacting with automation.

There are a variety of ways that the language could be modified in order to better support multiple operators. The current EOFM language allows multiple *humanoperator* nodes to share input information via *inputvariablelink* nodes. While this accomplishes the desired goal, it requires that all inputs be defined within a given *humanoperator* node. This results in a strong coupling between the humanoperator nodes and associates the input exclusively with a human operator rather than the source of that input like the environment or a human–device interface. Similarly, multiple human operators may be able to interact with a human–device interface that they share, allowing each to submit identical human actions to system automation. The current implementation of the EOFM language supports this but requires that these human actions be defined using different names in the associated *humanoperator* nodes. Thus, it is the job of the modeler to ensure that these human actions are treated the same when the language is interpreted. Finally, multihuman operator systems may support direct communication between human operators which does not occur through a human–device interface. The current implementation of the EOFM does not support this feature.

These limitations could be resolved by modifying the EOFM semantics so that they were more object oriented. *inputvariable* and *humanaction* nodes as well as communication channels between human operators could be defined independent of the *humanoperator* nodes and referenced where appropriate in these nodes. Future work will investigate how these features might be incorporated into the EOFM.

Another limitation comes in the way that task structures (activity and action hierarchies) can be reused between *humanoperator* nodes. In the current implementation, activities must be defined in one *humanoperator* node and referenced in others using *activitylink* nodes. This too results in strong coupling and does not support good object-oriented design. There are a variety of different architectures that might be supported in a multioperator system: There may be multiple operators with a mixture of shared and discrepant task structures, and there may be shared or discrepant human–device interfaces. Object-oriented concepts, such as inheritance, interfaces, and polymorphism, may allow these types of relationships to be more easily and flexibly instantiated in the EOFM. Future work should investigate how this might be accomplished.

C. Need for Unification

The EOFM was developed to not only extend the functionality of the OFM but also allow it to be incorporated into the formal modeling notation using defined formal semantics.

However, the discrepant feature sets present in the various task analytic modeling paradigms, the current limitations of the EOFM, and the limitations inherent to model checking analyses suggest that there is a need for a unifying study on exactly what features task analytic models need to support. Thus, before making significant changes to the EOFM, work should identify what these features are so that the EOFM, or some other task modeling language, can serve the human–automation interaction community at large.

Furthermore, while the EOFM's formal semantics are defined generically, our EOFM-to-SAL translator makes assumptions about the way that an instantiated EOFM will integrate with the rest of a formal system model. Future work could investigate whether there are generic and/or abstract methods for integrating task analytic models into formal analyses.

All of the conditions that have been formally verified using our technique have been safety properties specific to the particular application being evaluated. Work concerned with the formal modeling of human–device interfaces has identified generic temporal logic patterns for specifying usability properties that can be formally verified against models of human–device interfaces (see [53]–[55]). Future work should determine if there are generic temporal logic properties that can be used to check for properties important to HAI against formal system models containing task analytic models.

Finally, the system analyzed in this paper as well as all of the other systems evaluated using our technique [30], [31], [42] have used control automation. However, there are many different types of automation in human–automation interactive systems [56] such as information analysis automation [57]. Future work should investigate how applicable our method is to systems that utilize these other types of automation.

ACKNOWLEDGMENT

The research described herein originated when the first author was pursuing his Ph.D. in systems engineering at the University of Virginia. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institute of Aerospace, the National Aeronautics and Space Administration, the National Library of Medicine, or the National Institutes of Health.

REFERENCES

[1] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, no. 9, pp. 8, 10–22, 24, Sep. 1990.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.

[3] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, A. R. Meyer, M. Nivat, M. Paterson, and D. Perrin, Eds. Cambridge, MA: MIT Press, 1990, ch. 16, pp. 995–1072.

[4] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 626–643, 1996.

[5] D. Hughes and M. Dornheim, "Accidents direct focus on cockpit automation," *Aviation Week Space Technol.*, vol. 142, no. 5, pp. 52–54, 1995.

[6] P. Ladkin, *AA965 Cali Accident Report: Near Buga, Colombia, Dec. 20, 1995*, Bielefeld, Germany, 1996. [Online]. Available: http://sunnyday.mit.edu/accidents/calirep.html

[7] E. Sekigawa and M. Mecham, "Pilots, A300 systems cited in Nagoya crash," *Aviation Week Space Technol.*, vol. 29, pp. 6–37, 1996.

[8] B. Kirwan and L. K. Ainsworth, *A Guide to Task Analysis*. London, U.K.: Taylor & Francis, 1992.

[9] F. Paternò, C. Mancini, and S. Meniconi, "Concurtasktrees: A diagrammatic notation for specifying task models," in *Proc. IFIP TC13 Int. Conf. Human-Comput. Interact.*, 1997, pp. 362–369.

[10] C. M. Mitchell and R. A. Miller, "A discrete control model of operator function: A methodology for information display design," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-16, no. 3, pp. 343–357, May 1986.

[11] H. R. Hartson, A. C. Siochi, and D. Hix, "The UAN: A user-oriented representation for direct manipulation interface designs," *ACM Trans. Inf. Syst.*, vol. 8, no. 3, pp. 181–203, Jul. 1990.

[12] S. Basnyat, P. Palanque, B. Schupp, and P. Wright, "Formal socio-technical barrier modelling for safety-critical interactive systems design," *Safety Sci.*, vol. 45, no. 5, pp. 545–565, Jun. 2007.

[13] S. Basnyat, P. A. Palanque, R. Bernhaupt, and E. Poupart, "Formal modelling of incidents and accidents as a means for enriching training material for satellite control operations," in *Proc. Joint ESREL and 17th SRA-Eur. Conf.*, London, U.K., 2008, CD–ROM.

[14] E. L. Gunter, A. Yasmeen, C. A. Gunter, and A. Nguyen, "Specifying and analyzing workflows for automated identification and data capture," in *Proc. 42nd Hawaii Int. Conf. Syst. Sci.*, 2009, pp. 1–11.

[15] Y. Aït-Ameur, M. Baron, and P. Girard, "Formal validation of HCI user tasks," in *Proc. Int. Conf. Softw. Eng. Res. Pract.*, 2003, pp. 732–738.

[16] Y. Aït-Ameur and M. Baron, "Formal and experimental validation approaches in HCI systems design based on a shared event B model," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 6, pp. 547–563, Nov. 2006.

[17] F. Paternò, C. Santoro, and S. Tahmassebi, "Formal model for cooperative tasks: Concepts and an application for en-route air traffic control," in *Proc. 5th Int. Conf. Des., Specification, Verification Interactive Syst.*, 1998, pp. 71–86.

[18] F. Paternò and C. Santoro, "Integrating model checking and HCI tools to help designers verify user interface properties," in *Proc. 7th Int. Workshop Des., Specification, Verification Interactive Syst.*, 2001, pp. 135–150.

[19] P. A. Palanque, R. Bastide, and V. Senges, "Validating interactive system design through the verification of formal task and system models," in *Proc. IFIP TC2/WG2.7 Work. Conf. Eng. Human-Comput. Interact.*, 1996, pp. 189–212.

[20] R. E. Fields, "Analysis of erroneous actions in the design of critical systems," Ph.D. dissertation, Univ. York, York, U.K., 2001.

[21] B. E. John and D. E. Kieras, "Using GOMS for user interface design and evaluation: Which technique?" *ACM Trans. Comput.-Human Interact.*, vol. 3, no. 4, pp. 287–319, Dec. 1996.

[22] D. Hix and H. R. Hartson, *Developing User Interfaces: Ensuring Usability Through Product and Process*. New York: Wiley, 1993.

[23] M. L. Bolton and E. J. Bass, "Enhanced operator function model: A generic human task behavior modeling language," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, 2009, pp. 2983–2990.

[24] M. L. Bolton and E. J. Bass, "Using task analytic models to visualize model checker counterexamples," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, 2010, pp. 2069–2074.

[25] L. De Moura, S. Owre, and N. Shankar, "The SAL language manual," Comput. Sci. Lab., SRI Int., Menlo Park, CA, Tech. Rep. CSL-01-01, 2003.

[26] J. Clark and M. Murata, "Relax NG specification," *Committee Specification, Organization for the Advancement of Structured Information Standards*, 2001. [Online]. Available: http://relaxng.org/spec-20011203.html

[27] N. Shankar, "Symbolic analysis of transition systems," in *Proc. Int. Workshop Abstr. State Mach., Theory Appl.*, 2000, pp. 287–302.

[28] P. Le Hégaret, *The W3C Document Object Model (DOM)*, 2002. [Online]. Available: http://www.w3.org/2002/07/26-dom-article.html

[29] M. L. Bolton and E. J. Bass, "Building a formal model of a human-interactive system: Insights into the integration of formal methods and human factors engineering," in *Proc. 1st NASA Formal Methods Symp.*, 2009, pp. 6–15.

[30] M. L. Bolton and E. J. Bass, "Formally verifying human–automation interaction as part of a system model: Limitations and tradeoffs," *Innov. Syst. Softw. Eng.: NASA J.*, vol. 6, no. 3, pp. 219–231, Sep. 2010.

[31] M. L. Bolton, "Using task analytic behavior modeling, erroneous human behavior generation, and formal methods to evaluate the role of human–automation interaction in system failure," Ph.D. dissertation, Univ. Virginia, Charlottesville, VA, 2010.

[32] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, "Using formal methods to predict human error and system failures," in *Proc. 2nd Int. Conf. Appl. Human Factors Ergonom.*, 2008, CD–ROM.

[33] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[34] M. Sipser, *Introduction to the Theory of Computation*, 2nd ed. Boston, MA: Thomson Course Technol., 2006.

[35] J. R. Burch, E. M. Clarke, D. L. Dill, J. Hwang, and K. L. McMillan, "Symbolic model checking: 10^{20} states and beyond," *Inf. Comput.*, vol. 98, no. 2, pp. 142–171, Jun. 1992.
[36] G. Holzmann and D. Peled, "An improvement in formal verification," in *Proc. 7th Int. Conf. Formal Description Techn.*, 1994, pp. 197–211.
[37] S. Graf and H. Saïdi, "Verifying invariants using theorem proving," in *Proc. 8th Int. Conf. Comput. Aided Verification*, 1996, pp. 196–207.
[38] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Principles Program. Lang.*, 1977, pp. 238–252.
[39] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003.
[40] T. A. Henzinger, "The theory of hybrid automata," in *Proc. 11th Annu. IEEE Symp. Logic Comput. Sci.*, 1996, pp. 278–292.
[41] T. A. Henzinger, Z. Manna, and A. Pnueli, "Timed transition systems," in *Proc. REX Workshop*, 1991, pp. 226–251.
[42] M. L. Bolton and E. J. Bass, "A method for the formal verification of human interactive systems," in *Proc. 53rd Annu. Meeting Human Factors Ergonom. Soc.*, 2009, pp. 764–768.
[43] J. D. Lee and T. F. Sanquist, "Augmenting the operator function model with cognitive operations: Assessing the cognitive demands of technological innovation in ship navigation," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 30, no. 3, pp. 273–285, May 2000.
[44] R. M. Young, T. R. G. Green, and T. Simon, "Programmable user models for predictive evaluation of interface designs," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 1989, pp. 15–19.
[45] A. Blandford, R. Butterworth, and J. Good, "Users as rational interacting agents: Formalising assumptions about cognition and interaction," in *Proc. 4th Int. Eurographics Workshop, Des., Specification Verification Interactive Syst.*, vol. 97. Berlin, Germany: Springer-Verlag, 1997, pp. 45–60.
[46] R. Butterworth, A. Blandford, and D. Duke, "Demonstrating the cognitive plausibility of interactive system specifications," *Formal Aspects Comput.*, vol. 12, no. 4, pp. 237–259, 2000.
[47] A. Blandford, R. Butterworth, and P. Curzon, "Models of interactive systems: A case study on programmable user modelling," *Int. J. Human-Comput. Stud.*, vol. 60, no. 2, pp. 149–200, Feb. 2004.
[48] P. Curzon and A. Blandford, "From a formal user model to design rules," in *Proc. 9th Int. Workshop Interactive Syst. Des., Specification, Verification*, 2002, pp. 1–15.
[49] P. Curzon and A. Blandford, "Formally justifying user-centered design rules: A case study on post-completion errors," in *Proc. 4th Int. Conf. Integr. Formal Methods*, 2004, pp. 461–480.
[50] R. Rukšėnas, P. Curzon, J. Back, and A. Blandford, "Formal modelling of cognitive interpretation," in *Proc. 13th Int. Workshop Des., Specification, Verification Interactive Syst.*, 2007, pp. 123–136.
[51] R. Rukšėnas, P. Curzon, A. Blandford, and J. Back, "Combining human error verification and timing analysis," in *Proc. Conf. Eng. Interactive Syst.*, 2009, pp. 18–35.
[52] M. L. Bolton and E. J. Bass, "Formal modeling of erroneous human behavior and its implications for model checking," in *Proc. 6th NASA Langley Formal Methods Workshop*, 2008, pp. 62–64.
[53] G. D. Abowd, H. Wang, and A. F. Monk, "A formal technique for automated dialogue development," in *Proc. 1st Conf. Designing Interactive Syst.*, 1995, pp. 219–226.
[54] J. C. Campos and M. D. Harrison, "Systematic analysis of control panel interfaces using formal tools," in *Proc. 15th Int. Workshop Des., Verification Specification Interactive Syst.*, 2008, pp. 72–85.
[55] F. Paternò, "Formal reasoning about dialogue properties with automatic support," *Interacting Comput.*, vol. 9, no. 2, pp. 173–196, 1997.
[56] R. Parasuraman, T. Sheridan, and C. Wickens, "A model for types and levels of human interaction with automation," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 30, no. 3, pp. 286–297, May 2000.
[57] E. Bass and A. Pritchett, "Human-automated judge learning: A methodology for examining human interaction with information analysis automation," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 38, no. 4, pp. 759–776, Jul. 2008.
[58] "Relax NG schema diagram," *Syncro Soft*. [Online]. Available: http://www.oxygenxml.com/doc/ug-oxygen/topics/relax-ng-schema-diagram.html

Matthew L. Bolton (S'05–M'10) received the B.S. degree in computer science, the M.S. degree in systems engineering, and the Ph.D. degree in systems engineering from the University of Virginia, Charlottesville, in 2003, 2006, and 2010, respectively.

He is currently a Senior Research Associate with the San José State University Research Foundation, National Aeronautics and Space Administration Ames Research Center, Moffett Field, CA. His primary research focus is on the development of tools and techniques for using formal methods in the modeling, validation, verification, and design of safety-critical human–automation interactive systems.

Radu I. Siminiceanu received the B.S. and M.S. degrees in computer science from the University of Iaşi, Iaşi, Romania, and the Ph.D. degree in computer science from the College of William and Mary, Williamsburg, VA, in 2003.

He is currently a Senior Research Scientist with the National Institute of Aerospace, Hampton, VA. His research interests include formal methods, particularly model checking, applied to aerospace, air traffic management, and avionics systems.

Ellen J. Bass (M'98–SM'03) received the B.S. Eng. and B.S. Econ. degrees from the University of Pennsylvania, Philadelphia, the M.S. degree from the State University of New York at Binghamton, Vestal, and the Ph.D. degree from the Georgia Institute of Technology, Atlanta.

She is currently an Associate Professor of systems engineering with the Department of Systems and Information Engineering, University of Virginia, Charlottesville. She has over 25 years of industry and research experience in human-centered systems engineering in the domains of aviation, meteorology, bioinformatics, and medical informatics. Her research focuses on modeling human judgment and decision making in dynamic environments in order to inform the design of decision support and training systems.