Innovations Syst Softw Eng
DOI 10.1007/s11334-016-0272-z

ORIGINAL PAPER

Improving the scalability of formal human–automation interaction verification analyses that use task-analytic models

Matthew L. Bolton[1] · Xi Zheng[1] · Kylie Molinaro[1] · Adam Houser[1] · Meng Li[1]

Received: 12 June 2015 / Accepted: 27 January 2016
© Springer-Verlag London 2016

Abstract The enhanced operator function model with communications (EOFMCs) is a task-analytic modeling formalism used for including human behavior in formal models of larger systems. This allows the contribution of human behavior to the safety of the system to be evaluated with model checking. The previous method for translating the EOFMCs into model checker input language was conceptually straightforward, but extremely statespace inefficient. This limited the applications that could be formally verified using EOFMC. In this paper, we present an alternative approach for formally representing EOFMCs that substantially decreases the model's statespace size and verification time. This paper motivates this effort, describes how the improvement was achieved, presents benchmarks demonstrating the improvements in statespace size and verification time, discusses the implications of these results, and outlines directions for future improvement.

Keywords Model checking · Task analytic models · Formal methods · Scalability

✉ Matthew L. Bolton
mbolton@buffalo.edu

Xi Zheng
xzheng24@buffalo.edu

Kylie Molinaro
kyliemol@buffalo.edu

Adam Houser
adamhous@buffalo.edu

Meng Li
mli42@buffalo.edu

[1] Department of Industrial and Systems Engineering, State University of New York at Buffalo, Amherst, NY 14260, USA

1 Introduction

Human factors engineers use task-analytic behavior models to describe the normative human behaviors required to control a system [37]. These models represent the mental and physical activities operators use to achieve the goals the system was designed to support. The Enhanced Operator Function Model (EOFM) [16] and its extension the Enhanced Operator Function Model with Communications (EOFMCs) [6] (both derived from the Operator Function Model [40]) are task-analytic modeling formalisms that allow the task behavior of a single human or a team of humans, respectively, to be used in formal verification analysis. Specifically, EOFMC (henceforth used to refer to both EOFMC and EOFM given that EOFMC encapsulated the behavior of EOFM) can be used with model checking (an automated, search-based approach to formal verification [21]) and additional system modeling to allow analysts to prove whether system models will or will not be safe while considering the human behaviors in an EOFMC. Further, EOFMC supports miscommunication generation that allows its impact to be considered in the formal verification analyses.

However, because EOFMC-based verification analyses rely on model checking [21], they are subject to its scalability limitations. Specifically, model checking suffers from combinatorial explosion: where adding additional model components can result in exponential increases in model statespace size [21]. This can result in models that are too big or take too long to verify. EOFMC analyses have been evaluated for their scalability. This has revealed that statespace size and verification times of models that include EOFMC behavior scale exponentially with the human operator tasks [16,17] and the maximum number of miscommunications included in the models [9]. As has been seen in practice, this

can severely limit what systems and types of model behavior can be evaluated [12].

If task models are going to be used to formally evaluate the safety and performance of critical human-interactive systems, steps need to be taken to improve its scalability. This paper presents a reinterpretation of EOFMC's formal semantics and associated translator that allow for a significant improvement in the scalability of EOFMC-based verification analyses without the loss of functionality.

In the following, we first discuss the background that is necessary for understanding EOFMC and its supported analyses. Then we explain the theory behind the scalability improvement for EOFMC, mathematically describe how the new translator interprets the formal semantics, and reveal how this was implemented. Afterward, test scalability benchmarks are presented to demonstrate the improvements the new implementation affords for both artificial and realistic test cases. Finally, the results are discussed and avenues for future work are explored.

2 Background

2.1 Formal methods and model checking

Formal methods are well-defined mathematical languages and techniques for the specification, modeling, and verification of systems [46]. Specification properties mathematically describe desirable system conditions; systems are modeled using mathematical languages; and verification then mathematically proves whether or not the model satisfies the specification. Model checking is an automated approach to formal verification [21]. In model checking, a formal model describes a system as a state transition model: a set of variables and transitions between variable states. Desirable specification properties are usually represented in a temporal logic [29]. Verification is performed automatically by exhaustively searching a system's statespace to determine if these properties hold. If they do, the model checker returns a confirmation. Otherwise, a counterexample is produced that shows how the specification violation occurred as a trace through the statespace of the model.

Formal methods are typically used to evaluate computer systems [21]. However, they have been used successfully to analyze human–automation interaction [18]. Of relevance to this paper is the work that has considered human operator task behavior in formal verification analysis as part of larger formal system models. Task analytic models are typically represented as a hierarchy of activities that decompose into other activities and (at the lowest level) atomic actions. In these models, strategic knowledge (condition logic) controls when activities can execute, and modifiers between activities or actions control how they execute in relation to each other. Because task-analytic models can be represented discretely, they can be used to include human behavior in formal system models.

A number of researchers have incorporated task-analytic models into formal system models of human–automation interactive systems by either modeling task behavior natively in the formal notation [4,5,20,31] or translating task-analytic models implemented in task-analytic representations (such as ConcurTaskTrees [43], EOFMC, or UAN [33]) into the formal notation [1,2,11,12,16,30,41,42,44]. This allows system safety properties to be verified in light of the modeled human behavior.

Of these, EOFMC is one of the most advanced and feature rich in that it can support the modeling of single [16] and multiple operators [6]; allows for the generation of erroneous human behavior using two different, theoretically driven techniques [15,17]; supports counterexample visualization [13]; can generate miscommunications between human operators [9]; is capable of generating checkable specification properties from task models that assert properties important to human–automation interaction [8,19]; and can be used to automatically generate functional descriptions of human–machine interfaces from task models [39]. Thus, by improving the scalability of EOFMC, this work has the potential to have the most impact on the related formal analyses that can be done with task models.

2.2 EOFMC

EOFMC is an XML-based task-analytic modeling formalism that enables analysts to consider how human operator behavior (including multiple interacting and communicating humans) impacts system performance and safety using formal verification [6,16]. EOFMC represents a single human or groups of humans as an input/output system. Inputs may come from a human interface, environment, and/or mission goals. Output variables are human actions. The operators' task models describe how human actions may be generated and how the values of local variables change based on input and local variables (representing perceptual or cognitive processing, task behavior, and inner group coordination and communication). All variables are defined in terms of constants, user defined types, and basic types.

Tasks in an EOFMC instance are represented as a hierarchy of goal-directed activities that ultimately decompose into atomic actions. Each task descends from a top level activity, where there can be multiple tasks in a given EOFMC. Tasks either belong to one human operator or are shared between human operators. A shared task is assigned to two or more associates, and a subset of associates for the general task is identified for each activity. Thus, it is explicit which human operators are participating in which activity.

Activities can have preconditions, repeat conditions, and completion conditions. These are represented by Boolean expressions written in terms of input, output, and local variables, as well as constants. They specify what must be true before an activity can execute (precondition), when it can execute again (repeat condition), and what is true when it has completed execution (completion condition).

Actions appear at the bottom of the task hierarchy. They can be any of the following: (a) observable, singular ways the human operator can interact with the environment (output variables); (b) a cognitive or perceptual act, where a value is assigned to a local variable; or (c) human–human communications, where a communicator performs a communication action and the information conveyed is stored in recipient local variables.

A decomposition operator specifies the temporal relationships and the cardinality of the decomposed activities or actions (when they can execute relative to each other and how many can execute). EOFMC supports all of the decomposition operators in Table 1.

EOFMC instances can be visualized as tree-like graphs (see Fig. 1) where actions are depicted by rectangles and activities by rounded rectangles. Decompositions are arrows, labeled with the decomposition operator, extending below an activity that points to a large rounded rectangle with the decomposed activities or actions. In these visualizations, strategic knowledge conditions are connected to the activity they modify: a *Precondition* is represented by a yellow, downward pointing triangle connected to the left side of the activity; a *CompletionCondition* is presented as a magenta, upward pointing triangle connected to the right of the activity; and a *RepeatCondition* is conveyed as a recursive arrow attached to the top of the activity. More information can be found in [13].

By exploiting the shared activity and communication action feature of EOFMC, human–human communication protocols can be modeled as shared task activities. Human communication actions can represent human–human communication. However, other actions can model the way that the human operator interacts with other elements of the work environment. Thus a human–human communication protocol can represent the human–human communication procedure and the human operator responses.

Actions occur at the bottom of EOFMC task hierarchies. Actions are modeled as either an assignment to an output variable (indicating an action has been performed) or a local variable (representing a perceptual, cognitive, or communication action). Shared activities can explicitly include human–human communication action inside of a *com* decomposition. In such decompositions, communicated information from one human operator can be received by other human operators (modeled as an update to a local variable).

2.2.1 Miscommunication generation

EOFMC supports the ability to automatically generate miscommunications in EOFMC models [9], so their impact on safety can be evaluated with formal verification. In miscommunication generation, any given communication action can execute normatively, have the source of the communication convey the wrong information, have one or more of the communication recipients receive the wrong information, or both. In all analyses, the analyst is able to control the maximum number of miscommunications that can occur (*Max*). The net effect of this is that analysts can evaluate how robust a protocol is for all possible ways that *Max* or fewer miscommunications can occur.

$$Activity.StartCondition := Parent.Executing \wedge \begin{cases} PreviousSibling.Done, & \text{if } Parent.Decomposition = ord \\ \bigwedge_{S \in Siblings} S.Ready, & \text{if } Parent.Decomposition = xor \\ \bigwedge_{S \in Siblings} \neg S.Executing, & \text{for all other non-parallel decompositions} \\ True, & \text{otherwise} \end{cases} \tag{1}$$

$$\begin{aligned} Activity.EndCondition := & \left(\bigwedge_{C \in Children} \neg C.Executing\right) \\ & \wedge \left(\begin{cases} \bigvee_{C \in Children} C.Done, & \text{if } Activity.Decomposition \in \{or_seq, or_par, xor\} \\ True, & \text{if } Activity.Decomposition = optor \\ \bigwedge_{C \in Children} \neg C.Done, & \text{otherwise} \end{cases}\right) \end{aligned} \tag{2}$$

$$Action.StartCondition := Parent.Executing \wedge \begin{cases} PreviousSibling.Done, & \text{if } Parent.Decomposition = ord \\ \bigwedge_{S \in Siblings} S.Ready, & \text{if } Parent.Decomposition = xor \\ \bigwedge_{S \in Siblings} \neg S.Executing, & \text{for all other non-parallel decompositions} \\ True, & \text{otherwise} \end{cases} \tag{3}$$

$$Action.EndCondition := Action.Executing \tag{4}$$

Table 1 EOFMC decomposition operators

Operator	Description
optor_seq	Zero or more of the sub-acts must execute in any order one at a time
optor_par	Zero or more of the sub-acts must execute in any order and can execute in parallel
or_seq	One or more of the sub-acts must execute in any order one at a time
or_par	One or more of the sub-acts must execute in any order and can execute in parallel
and_seq	All of the sub-acts must execute in any order one at a time
and_par	All of the sub-acts must execute in any order and can execute in parallel
xor	Exactly one sub-act must execute
ord	All sub-acts must execute in the order they appear
sync	All sub-acts must execute synchronously
com	A communication action is performed

2.2.2 EOFMC formal semantics

EOFMC has formal semantics that specify how an EOFMC instance executes. Each activity or action has one of three execution states: waiting to execute (*Ready*), executing (*Executing*), and done (*Done*). An activity or action transitions between states (Fig. 2) based on its current state; its start condition (*StartCondition*—when it can start executing based on the state of its immediate parent, its parent's decomposition operator, and the execution state of its siblings); its end condition (*EndCondition*—when it can stop executing based on the state of its immediate children in the hierarchy and its decomposition operators); its reset condition (*Reset*—when it can revert to *Ready* based on the execution state of its parents); and, for an activity, its strategic knowledge (the *Precondition*, *RepeatCondition*, and *CompletionCondition*).

Strategic knowledge conditions are explicitly specified in EOFMC XML. However, the *StartCondition*, *EndCondition*, and *Reset* condition for each activity and action are derived from the execution state of itself, its parent, its siblings (acts in the same decomposition), and its children (acts that are decomposed from it). The logical relationship between these execution states is determined by the decomposition operator of the given activity's or action's parent and/or the given activity's decomposition operator.

The logical formulations for the start and end conditions are shown in (1)–(4). Note that in these, for any given activity or action (*Act*):

$$Act.Ready := Act.ExecutionState = Ready, \tag{5}$$

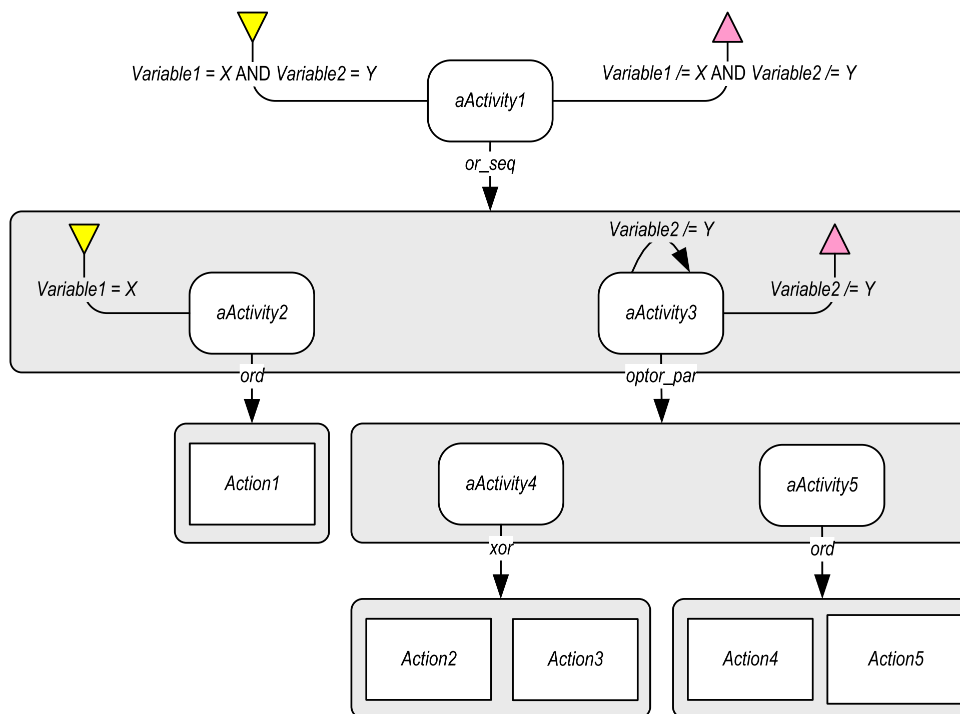

Fig. 1 An example of the visual representation of a task structure in an EOFMC instance (originally from [17]). Activity *aActivity1* has both a precondition and a completion condition. It decomposes into activities *aActivity2* and *aActivity3* with an *or_seq* decomposition operator. *aActivity2* has a precondition and decomposes into *Action1* with an *ord* decomposition operator. *aActivity3* has both repeat and completion conditions. It decomposes into *aActivity4* and *aActivity5* with an *optor_par* operator. *aActivity4* and *aActivity5* each decompose into two actions, *Action2* and *Action3* with an *xor* decomposition for the former and *Action4* and *Action5* with an *ord* decomposition for the latter

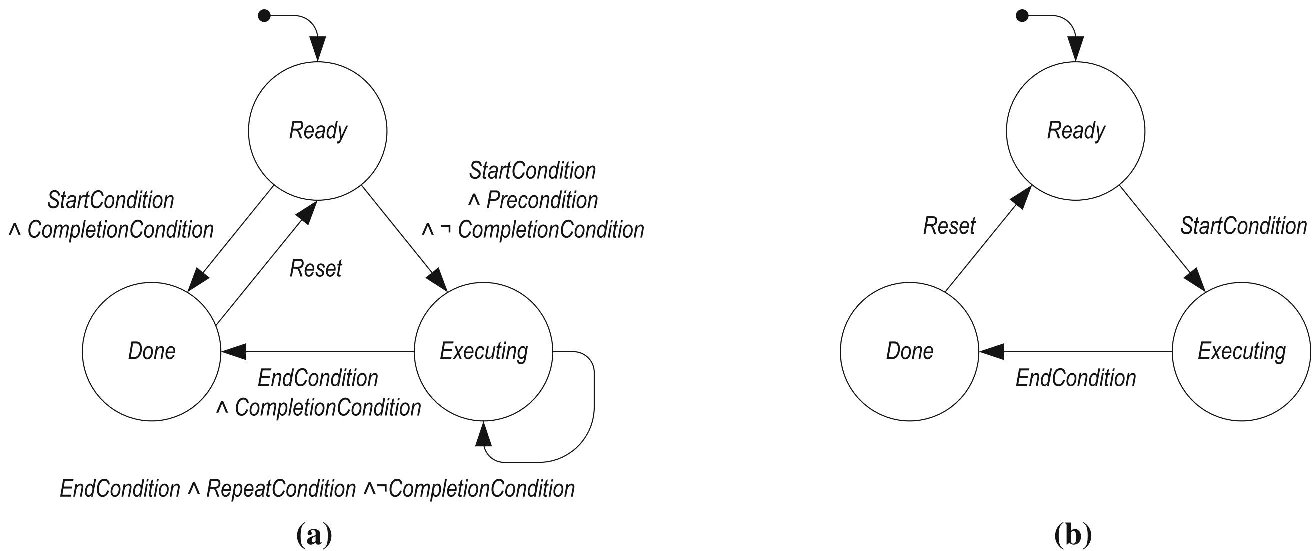

Fig. 2 Transition diagrams representing how activities (**a**) and actions (**b**) transition between execution states in EOFMC formal semantics

$$Act.Executing := Act.ExecutionState = Executing, \tag{6}$$

and

$$Act.Done := Act.ExecutionState = Done. \tag{7}$$

For any given activity or action in a decomposition, a *StartCondition* [(1) and (3), respectively] comprises two conjuncts: one stipulating conditions on the execution state of its parent and the other on the execution state of its siblings based on the parent's decomposition operator. Note that the start condition for an activity and action are formulated the same way.

An activity without a parent (a top-level activity) will eliminate the first conjunct. Top-level activities that are defined for a given *humanoperator* treat each other as siblings in the formulation of the second conjunct with an assumed *and_seq* relationship. All other activities are treated as if they are in an *and_par* relationship and are thus not considered in the formulation of the start condition. Top-level activities that are defined for *sharedeofm*s treat all other activities as if they are in an *and_par* relationship; thus they have start conditions that are always true.

An *EndCondition* also comprises two conjuncts, both related to an activity's sub-acts. The first asserts that none of the sub-acts are executing. The second asserts that the execution states of the activity's sub-acts satisfy the requirements stipulated by the activity's decomposition operator (2). Because an action has no sub-acts, an action's *EndCondition* defaults to *True* when an action is executing (4).

The *Reset* condition is different from the other transitions. Specifically, it is *True* when an activity's or action's parent transitions from *Done* to *Ready* or from *Executing* to *Executing* (when it repeats execution). If the activity has no parent (if it is at the top of the decomposition hierarchy), *Reset* is *True* if that activity is *Done*.

The transition criteria for an activity (Fig. 2a) are described in more detail below: (a) 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*. (b) When 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. (c) An activity will remain in the *Done* state until its *Reset* condition is satisfied, where it returns to the *Ready* state.

The transition criteria for an action are simpler (Fig. 2b) since an action cannot have strategic knowledge. Note that because actions do not have any sub-acts, their *EndConditions* are always *True* when the action is *Executing*. Also, note that because actions in a *sync* or *com* decomposition occur at the same time, the variables representing their execution states must transition through states at the same time. Further, note that when the action is *Executing*, this is when a given output variable action should be *True* or when the local variable or communication action information transfer to local variables occurs. This is also when any miscommunications can manifest (see [9] for more details).

2.2.3 Translation

Instantiated EOFMC task models can be translated into the language of the Symbolic Analysis Laboratory (SAL; see

[22]) using these formal semantics (in this case using a Java program) [16], where the task model can interact with other modeled system elements. For this, each activity's and action's execution state is explicitly modeled as a variable that can transition between *Ready*, *Executing*, and *Done* in accordance with the transitions in Fig. 2 and the other EOFMC formal semantics. Each of these transitions is represented as a single non-deterministic, guarded transition in a single SAL module.

The SAL model containing the human operator task behavior is ultimately asynchronously composed of one or more other modules meant to represent other system behavior analysts may wish to implement in the model. To support a coordination protocol between these asynchronously composed modules, the translator defines two Boolean variables in the group module (see [10] and [12]):

1. An input variable *InterfaceReady* to the human task module that is *True* when the interface is ready to receive input.
2. An output variable *ActionsSubmitted* to the human task module that is *True* when one or more human output actions are performed.

The *ActionsSubmitted* output variable is initialized to *False*.

A deeper description of this translation process can be found in [16] and [7].

This is an intuitive way of representing the execution state of activities and actions and realizing the EOFMC formal semantics in a translator. However, it can also be very statespace inefficient. This is true for two primary reasons. First, the number of variables required for representing the entire task model must match the number of activities and actions in the task model. Secondly, the inherently hierarchical nature of EOFMC models results in a number of intermediate states and state transitions that represent the execution state of activities. As such, there can be many states representing changes in a task model's aggregate statespace that do not result in changes to the state of a task's actions. Because the actions are the only part of the task model that can affect change in other elements of the system, these intermediate states are very inefficient.

3 A more statespace-efficient translator

In this section, we describe a new translator that uses a different method for implementing the EOFMC formal semantics to reduce the statespace size of the resulting model. The translator design presented here accomplishes this by removing unnecessary variables and intermediate transitions.

The major insight that makes this design possible is that the execution state of an activity can be viewed as an abstraction of the execution state of the activities and actions it decomposes into. Since every EOFMC activity must ultimately decompose into actions, the execution state of every activity can be represented as a Boolean expression based on the execution state of its actions. Thus, the modified translator design (with one exception) removes the variables for EOFMC activity execution states and represents an entire model's behavior based on the execution state and transitions between the execution state of its actions.

In this design, the explicit representation of action execution state remains the same as in the original translator implementation (each action has a variable representing its execution state). Thus, the Boolean expressions used to reason about those states are given by (8)–(10).

However, in most situations, each possible execution state of each activity is recursively represented as a Boolean expression of the execution state of its children. We define the execution state of an activity as seen in (11)–(13).

$$Action.Ready := Action.ExecutionState = Ready \tag{8}$$

$$Action.Done := Action.ExecutionState = Done \tag{9}$$

$$Action.Executing := Action.ExecutionState = Executing \tag{10}$$

$$Activity.Ready := \left(\bigwedge_{C \in Children} C.Ready \right) \wedge \neg Activity.Repeating \tag{11}$$

$$Activity.Done := \begin{cases} Activity.ExecutionState = Done & \text{if } Activity \text{ has a } RepeatCondition \text{ or a } CompletionCondition \\ (\bigwedge_{C \in Children} C.Done) & \text{Otherwise} \end{cases} \tag{12}$$

$$Activity.Executing := \neg Activity.Ready \wedge \neg Activity.Done \tag{13}$$

$$\begin{aligned} Activity.CanRepeat := {} & Activity.StartCondition \wedge Activity.EndCondition \wedge Activity.RepeatCondition \wedge \neg Activity.CompletionCondition \\ & \wedge \left(\begin{cases} Activity.Executing \vee \begin{pmatrix} Activity.Ready \\ \wedge Activity.Precondition \end{pmatrix} & \text{if } Activity.Decomposition \in \{optor_seq, optor_par\} \\ Activity.Executing & \text{otherwise} \end{cases} \right) \end{aligned} \tag{14}$$

$$Activity.CanDone := \begin{cases} Activity.CompletionCondition \wedge \neg Activity.CanRepeat \\ \quad \wedge \begin{pmatrix} (Activity.Executing \wedge Activity.EndCondition) \\ \vee (Activity.Ready \wedge Activity.StartCondition) \end{pmatrix} & \text{if } Activity \text{ has a } Completion.Condition \\ \neg Activity.CanRepeat \\ \quad \wedge \begin{pmatrix} (Activity.Executing \wedge Activity.EndCondition) \\ \vee \begin{pmatrix} Activity.Ready \wedge Activity.StartCondition \\ \wedge Activity.Precondition \end{pmatrix} \end{pmatrix} & \text{if } Activity.Decomposition \in \{optor_seq, optor_par\} \\ \neg Activity.CanRepeat \wedge Activity.Executing \\ \quad \wedge Activity.EndCondition & \text{otherwise} \end{cases} \quad (15)$$

$$Activity.CanExecute := Activity.Ready \wedge Activity.StartCondition \wedge Activity.Precondition \wedge \neg Activity.CompletionCondition \quad (16)$$

$$Action.StartCondition := \begin{pmatrix} Parent.Executing \\ \vee Parent.CanExecute \end{pmatrix} \wedge \begin{cases} PreviousSibling.Done, & \text{if } Parent.Decomposition = ord \\ \bigwedge_{S \in Siblings} S.Ready, & \text{if } Parent.Decomposition = xor \\ \bigwedge_{S \in Siblings} \neg S.Executing, & \text{for all other non-parallel decompositions} \\ True, & \text{otherwise} \end{cases} \quad (17)$$

$$Activity.StartCondition := \begin{pmatrix} Parent.Executing \\ \vee Parent.CanExecute \end{pmatrix} \wedge \begin{cases} PreviousSibling.Done, & \text{if } Parent.Decomposition = ord \\ \bigwedge_{S \in Siblings} S.Ready, & \text{if } Parent.Decomposition = xor \\ \bigwedge_{S \in Siblings} \neg S.Executing, & \text{for all other non-parallel decompositions} \\ True, & \text{otherwise} \end{cases} \quad (18)$$

An activity is *Ready* (11) if all of its children are *Ready* and the activity is not repeating. Note that because all actions are initially assigned to *Ready*, this means that an activity will always default to *Ready* in its initial state. Further, note that (11) uses the variable *Activity.Repeating*. *Activity.Repeating* is a Boolean variable that exists if the given *Activity* has a *RepeatCondition*. It is *True* if the *Activity* is repeating and *False* otherwise. This variable was created to ensure that, if an activity needs to repeat and thus reset all of its descendants to *Ready*, the activity will still be treated as if it is *Executing*.

An activity is *Done* (12) if all of its children are *Done* and it does not have a repeat or completion condition. However, if it does have one of these conditions, some additional infrastructure is required. Specifically, an executing activity with a repeat condition may have all of its children become *Done* before it repeats and/or an activity with a completion condition may have all of its children become done without the completion condition being satisfied. Thus, to avoid erroneous *Done* states, a variable is created for activities with these conditions *Activity.ExecutionState* that indicates if the activity is *Done* or not. It is important to note that this definition of *Done* is slightly stronger than a simple satisfaction of an activity *EndCondition*. This was done purposely to ensure that there was no ambiguity about whether an activity was *Done*: certain decompositions (for example, *or_seq* and *or_par*) can have associated end conditions that can be satisfied before an activity necessarily needs to become *Done*.

An activity is *Executing* (13) if it is not *Ready* or *Done*.

These definitions ignore some ambiguities about execution state that arise with the use of *optor* decompositions operators. Specifically, they ignore that an activity with such a decomposition could be *Executing* or *Done* even if all of its children are *Ready*. This potential problem arises because the modified translator eliminates most of the intermediate transitions between activity execution states that would have prevented such issues. To both address this and give actions enough information to determine when they can transition, expressions are formulated to indicate when an activity can transition to a new execution state (based on the transitions from Fig. 2), thus ensuring that actions will be able to determine when the necessary intermediary transitions would have allowed them to occur. We define expressions for indicating when an activity can repeat (14), can transition to *Done* (15), and can *Execute* (16).

An activity can repeat (14) if its *StartCondition*, *EndCondition*, and *RepeatCondition* are satisfied, its *CompletionCondition* is not, and it is *Executing* or, if it has an *optor* decomposition, it is *Ready* with a satisfied *Precondition*.

If an activity has a *CompletionCondition*, it can become done if its completion condition is satisfied and it is either *Executing* with a satisfied *EndCondition* or *Ready* with a satisfied *StartCondition*. If it has an *optor* decomposition, the condition is the same except that when the activity is *Ready* with a satisfied *StartCondition*, the *Precondition* must also be true. Otherwise, the activity need only be *Executing* with a true *EndCondition*.

Note that the expressions for representing *CanRepeat* (14) and *CanDone* (15) take some minor liberties with the formal semantics for how *optor* decompositions are handled. Specifically, an activity with such a decomposition can go from *Ready* to *Done* or *Ready* to *Repeating* (*Executing* as

if it was repeating) by automatically transitioning through (effectively skipping) *Executing* states where nothing would occur. Thus, although this does not explicitly follow the formal semantics from Fig. 2, it is still consistent with it.

An activity can execute (16) if it is *Ready*, its *StartCondition* and *Precondition* are *True*, and its *CompletionCondition* is *False*.

In (14)–(16), if a given activity does not contain a strategic knowledge condition (a *Precondition*, *CompletionCondition*, or *RepeatCondition*), the condition (or the negation of the condition) should be eliminated.

To account for the fact that there is no explicit state for determining if a parent activity is *Executing*, the *StartCondition* needs to be slightly modified to include the ability for an activity or act to satisfy its *StartCondition* if the activities it descends from can execute; see (17) and (18).

Even with these changes, the *EndCondition* definitions from (2) and (4) remain the same.

These mathematical expressions were collectively used to express the nondeterministic, guarded transitions that defined the behavior of the formal task behavior model. As with the previous version of the translator, task behavior was represented in a single module, where input variables represented system information available to the human operator task model (corresponding to the input variables from the EOFMC's XML), outputs represented the human actions, and local variables were used to keep track of activity (where necessary) and action execution state (all initialized to *Ready*). This module also contained the Boolean input *ActionsSubmitted* and Boolean output *InterfaceReady* variables used for coordination between other system elements in the formal model and the formal representation of the task behavior (this was discussed in Sect. 2.2.3).

As with the original model, non-deterministic transitions were used to define how transitions occurred in between activity and action execution state. However, these were formulated slightly differently given the elimination of the explicit representation of most activity execution states.

For each action, a *Ready* to *Executing* transitions could occur if $Action.Ready \wedge Action.StartCondition \wedge InterfaceReady$ (the transition guard). In such a transition, the associated action's execution state would be set to *Executing*, the variables representing the performance of the action would be set to the appropriate value, and *ActionsSubmitted* would be set to *True*. A single general transition was responsible for handling all action *Executing* to *Done* transitions. Specifically, if $ActionsSubmitted \wedge \neg InterfaceReady$, then (for the next state) *ActionsSubmitted* would be set to *False*, the execution state of all executing actions would be set to *Done*, and the variables representing action outputs would be reset. Note that these transitions are consistent with the behavior from the previous version of the translator [9,12, 16], where only the reformulation of *Action.StartCondition* would impact the final translated form.

Although the execution states of activities were, for the most part, not explicitly represented, there were still several activity-related transitions. First, a transition was created to transition the activity to *Done* (encapsulating possible *Ready* to *Done* and *Executing* to *Done* transitions from Fig. 2a). For this, if the guard asserting *Activity.CanDone* was satisfied, the execution state of all descendent actions would be set to *Done*; the execution state of the activity or any descendent activity with repeat or completion conditions would be set to *Done*; and all *Activity.Repeating* variables associated with the activity or any of its decedents would be set to *False*. Second, for every activity with a repeat condition, a transition was created to represent its repetition behavior (an *Executing* to *Executing* transition from Fig. 2a). Specifically, if a guard asserting that *Activity.CanRepeat* was satisfied, then *Activity.Repeating* would be set to *True* and all of the variables associated with decedent activities and actions would be set to *Ready*: all descendent execution states set to *Ready* and all sub-activity *Repeating* variables set to false. Finally, if an activity is at the very top of the task model hierarchy, it needs to be able to reset itself to *Ready*. Thus, for every top-level activity, a transition was created that could only occur if a guard asserting *Activity.Done* was satisfied. If the transition occurs, all variables representing descendent execution states would be set to *Ready* and any *Repeat* variables associated with descendent activities would be set to *False*. Note that these last two transitions account for the *Reset* behavior required by the formal semantics (Fig. 2).

Aside from these changes, the modified translator acted in accordance with the original formal semantics and translator (see Sect. 2.2).

4 Testing

To evaluate whether the modified translator was correctly replicating the behavior of the original translator, it was tested using a series of simple instantiated models. These models were created to test both that the translators were producing the expected sequences of actions for the different EOFMC decomposition operators and that consistent sequences were being produced between translators.

To accomplish this, three different EOFMC tasks were created (Fig. 3). For each of these, versions were constructed that used each of the possible decomposition operators (Table 1).[1] These particular task models were used because they

[1] Note that because the *com* operator is so different from the others, it was not included in these tests. Because the *com* operator behavior only effects action behavior, it behaves in accordance with the previous translator [9]. Thus, the new translator should not affect the way

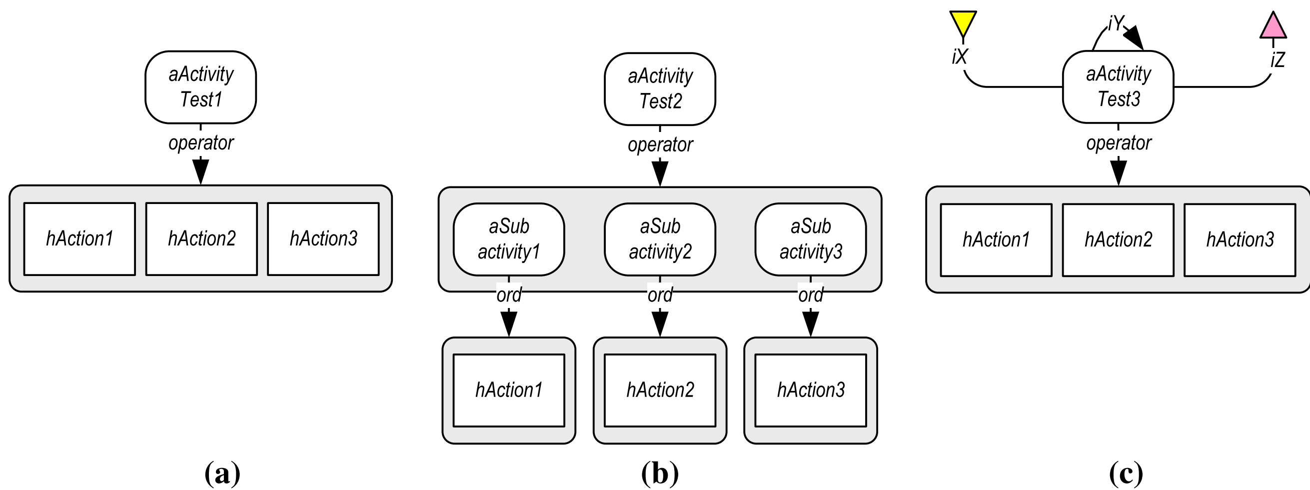

Fig. 3 EOFMC task structures used to test that the modified translator correctly replicated EOFMC behavior. *operator* in all of the task structures above shows where the decomposition operator was varied in each model. **a** Task for Test 1. **b** Task for Test 2. **c** Task for Test 3

encapsulate a representative spectrum of EOFM behavior sufficient for testing the features of the new translator. Specifically, the use of three actions allows for a number of different action sequences for different decomposition operators. The differences in the levels of hierarchy between the task for Test 2 (Fig. 3b) and the other tasks ensures that we are testing conditions, both where activities decompose into actions and when activities decompose into other activities. This was important because of the different way that activity execution states were handled in the new version of the translator. The task for Test 3 (Fig. 3c) accounts for the presence of strategic knowledge conditions, while the other two tasks do not use strategic knowledge. This is also important because the presence of the strategic knowledge affects the behavior of EOFMC activities and the new formulation of the formal semantics.

The EOFMC task models (each of the tasks from Fig. 3 with the different decomposition operators) were translated into SAL using the original and the new, modified translator.[2] Formal models created with both translators were completed by pairing the formal task representing with a module capable of accepting all of the actions it performed. An additional module was also included in the formal representation that would observe the actions being performed and update the value of variables that indicate if any of the 26 possible action sequences had executed. Note that, across all decompositions, an action sequence could have between 0 and 3 different ordered entries. Each entry could be a single action, a pair of concurrently executing actions, or all three actions executing concurrently (full listings of these can be seen in Tables 2, 3 and 4).

Additional model behaviors were also included in the formal models for tests using the task in Fig. 3c. Specifically, the formal module that accepted human operator inputs ensured that variables *iX* and *iY* were always true and that *iZ* would only become true once the task was repeating. This was meant to ensure that the task would repeat exactly once. Additionally, these formal models contained two modules for recognizing action sequences: one when the task was not repeating and one when it was.

All of the formal models were given specifications asserting the absence of each action sequence. Formal models with task behavior from the model in Fig. 3c had specifications for asserting the absence of each action sequence twice, once before the repeat and once after. Model checking was used to verify all of the properties for each model. With this setup, if the model checker returned a counterexample, the associated action sequence was possible. A confirmation showed the opposite.[3]

The results of these analyses are shown in Tables 2, 3 and 4, corresponding to the models associated with the task shown in Fig. 3a, b and c, respectively. These results show two things. First, each of the tests produced all of the action sequences that would be expected for the given task and decomposition operator and none that would not be expected. Second, the action sequences for comparable models created using the original and the new translator were identical in all

com decompositions are executed. Further, no anomalies were observed in the verification results of the realistic benchmarks reported subsequently. Thus, the evidence suggests that *com* decompositions are behaving the way they are supposed to.

[2] Note that the formal representation was slightly modified to remove the topmost activities' *Done* to *Ready* transitions. This ensured that the task would not repeat due to a *Reset* and thus not produce action execution sequences outside of a single execution.

[3] A full listing of all of the models used in these analyses can be found at http://fhsl.eng.buffalo.edu/resources/.

Table 2 Testing results for models created using the task from Fig. 3a

	Model Decomposition Operator and Translator																	
	optor_seq		optor_par		or_seq		or_par		and_seq		and_par		xor		ord		sync	
Sequence	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New
∅	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×
1	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×
2	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×
3	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×
$1 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$1 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$2 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$2 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$3 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$3 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$1 \rightarrow 2 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	×	×
$1 \rightarrow 3 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$2 \rightarrow 1 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$2 \rightarrow 3 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$3 \rightarrow 1 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$3 \rightarrow 2 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
1 & 2	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×
1 & 3	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×
2 & 3	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×
$1 \rightarrow 2$ & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
2 & $3 \rightarrow 1$	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
$2 \rightarrow 1$ & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
1 & $3 \rightarrow 2$	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
$3 \rightarrow 1$ & 2	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
1 & $2 \rightarrow 3$	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
1 & 2 & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓

1, 2, and 3 are used to reference *hAction1*, *hAction2*, and *hAction3* respectively from the tasks in 3. Actions to the left and right of an & symbol indicate that they occurred concurrently. Actions or groups of concurrent actions separated by a → indicate sequential execution from left to right. A ✓ indicates that the associated sequence was observed in the given version of the model. The × indicates it was not

cases. This provides strong evidence that the new translator properly represents EOFMC task behavior.[4]

Additionally, a number of models from the EOFMC literature (see Sect. 6) were retranslated into SAL using the modified translator and paired with the other elements of their original formal models. These were evaluated for a variety of different specifications (related to the specific context of each model) to ensure that they produced the same analysis results with the new translator and the original.[5] In all cases, the models created using the modified translator produced the expected verification results. Further, these were consistent with the results obtained from models created using the original translator.

[4] It is important to note the the original translator was involved in rigorous validation testing to ensure that it was behaving in conformance with the formal semantics (see [7]).

[5] Note that more verifications were run beyond those used in the realistic benchmarks discussed in Sect. 6. Deadlock checking was also performed on all of the models. No deadlock states were detected.

5 Artificial benchmarks

To compare the scalability of the modified translator to the original translator, a series of artificial benchmarks were used. These benchmarks iteratively increased the number of EOFMC task structures used in a formal model created from each respective translator. Verification was performed using a valid specification property and results (number of visited states and verification time) were compared.

All of these benchmarks used the task structure shown in Fig. 4. In this task structure, a top-level activity *aActX* decomposes into two lower-level activities using an *optor_par* decomposition. Each of these sub-activities then uses an *optor_par* decomposition to decompose into a single action: *h1* for *aActXa* and *h2* for *aActXb*. Note that the *optor_par* decomposition was used in the benchmark tasks, because it is the decomposition operator associated with the largest statespace complexity [16].

Each benchmark case used one or more instances of this task structure. There were ten total benchmarks, where the first benchmark contained one instance of the benchmark

Table 3 Testing results for models created using the task from Fig. 3b

	Model Decomposition Operator and Translator															
	optor_seq		optor_par		or_seq		or_par		and_seq		and_par		xor		ord	
Sequence	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New
∅	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×
1	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×
2	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×
3	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×
1 → 2	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×
1 → 3	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×
2 → 1	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×
2 → 3	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×
3 → 1	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×
3 → 2	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×
1 → 2 → 3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓
1 → 3 → 2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×
2 → 1 → 3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×
2 → 3 → 1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×
3 → 1 → 2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×
3 → 2 → 1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×
1 & 2	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×
1 & 3	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×
2 & 3	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×
1 → 2 & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×
2 & 3 → 1	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×
2 → 1 & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×
1 & 3 → 2	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×
3 → 1 & 2	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×
1 & 2 → 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×
1 & 2 & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×

No *sync* operator is used here because they are only allowed for decompositions of an activity into one or more actions

task (Fig. 4) and every subsequent benchmark contained one more instance than the one previous.

A separate EOFMC XML file was created for each benchmark. Each was translated into SAL using both the original and modified translator. The formal representation of the task models were each synchronously composed with another module that would simply receive all performed human actions in accordance with the EOFMC-supported coordination protocol [12,16].

Each of the resulting 20 benchmark models were checked against the specification property shown in (19), which was valid for all models.

$$\mathbf{G}\neg(aAct1.Ready \wedge aAct1.Executing) \tag{19}$$

When SAL's symbolic model checker (SAL-SMC) was used to perform formal verification, (19) verified as being true for every model. For each model's verification, the number of visited states and total verification times (which included prepossessing time for SAL-SMC to convert the model into a binary decision diagram) were recorded. To compare the performance of the original translator to the new one, a reduction factor was computed (*OldTranslatorValue* / *NewTranslatorValue*) for both measures. Finally, an exponential function was fit to the data for each translator's model and an R^2 was computed. These results are presented in Table 5.

These show that for both the number of visited states and the verification time, the models generated by both the original and modified translator scale exponentially with the number of tasks included in the model (the exponential models fit with $R^2 \geq 0.99$). However, the models generated by the modified translator have much fewer states and verify much faster than those created using the original translator. Given this relationship, it is not surprising that the reduction factor from the statistics observed for the original translator models to those from the modified translator's models increased as the number of included tasks increased.

6 Realistic benchmarks

While the artificial benchmarks give us some indication of how models created by the two translators scale as the number of task models increases, they do not necessarily give us an indication of how they will perform on realistic examples. To evaluate this, additional benchmarks were conducted using instantiated EOFMs and EOFMCs (and their asso-

Table 4 Testing results for models created using the task from Fig. 3c

	Model Decomposition Operator and Translator																	
	optor_seq		optor_par		or_seq		or_par		and_seq		and_par		xor		ord		sync	
Sequence	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New
∅	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×
1	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×
2	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×
3	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×
$1 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$1 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$2 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$2 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$3 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$3 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$1 \rightarrow 2 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	×	×
$1 \rightarrow 3 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$2 \rightarrow 1 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$2 \rightarrow 3 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$3 \rightarrow 1 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$3 \rightarrow 2 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
1 & 2	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×
1 & 3	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×
2 & 3	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×
$1 \rightarrow 2$ & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
2 & $3 \rightarrow 1$	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
$2 \rightarrow 1$ & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
1 & $3 \rightarrow 2$	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
$3 \rightarrow 1$ & 2	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
1 & $2 \rightarrow 3$	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
1 & 2 & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓
Sequence on Repeat																		
∅	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×
1	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×
2	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×
3	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×
$1 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$1 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$2 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$2 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$3 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$3 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
$1 \rightarrow 2 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	×	×
$1 \rightarrow 3 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$2 \rightarrow 1 \rightarrow 3$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$2 \rightarrow 3 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$3 \rightarrow 1 \rightarrow 2$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
$3 \rightarrow 2 \rightarrow 1$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
1 & 2	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×
1 & 3	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×
2 & 3	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×
$1 \rightarrow 2$ & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
2 & $3 \rightarrow 1$	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
$2 \rightarrow 1$ & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
1 & $3 \rightarrow 2$	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
$3 \rightarrow 1$ & 2	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
1 & $2 \rightarrow 3$	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	×	×
1 & 2 & 3	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓

ciated complete formal system models) used in previous analyses. In all cases, the EOFMC models were translated into SAL using both translators. The translated version of the task behavior was then paired with the rest of their respective formal system models. SAL-SMC was then used to verify the same valid property for both versions of the model. The number of visited states and verification times were recorded.

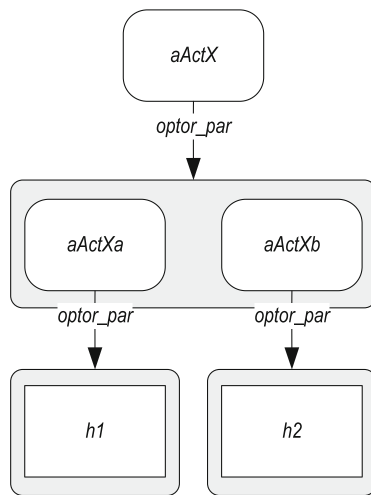

Fig. 4 The EOFMC task pattern used in benchmark experiments. Note that *X* represents the number of the instance in the given benchmark, where *X* can be between 1 and 10

Six different benchmark models were used. The first, which we will call "aircraft," represents a pilot's checklist-guided behavior (and the related aircraft systems) for performing the before-landing checklist on an instrument landing (see [14,19]). In this model, formal verification was used to ensure that, if pilots prefer to arm aircraft spoilers, the spoilers will be armed when the aircraft is ready to descend to the runway (see [14]). A second model, henceforth referred to as "Therac-25," represents a radiation therapy machine through which a technician must program and administer treatments [17]. In this case, the verified specification property checked that a full, unshielded radiation dose was never administered without a protective spreader in place. A third model, called "PCA," represents a pain medication pump that is programmed by a practitioner (see [12,15]). This was verified against a property specifying that, when treatment was being administered, the prescription programmed into the device matched what was prescribed. Finally, three different communication protocols were used, each representing a different procedure for communicating a heading change between air traffic control and the two pilots flying an aircraft [9]. Each protocol (1–3) was designed to be robust for different maximum numbers of miscommunications. Thus, four versions of each of these models were created, each allowing for a different maximum on the number of miscommunications allowed to be generated[6] between 0 and 3. In all instances of the protocol model, a specification was verified that the topmost activity associated with the coordinated heading change eventually finishes executing (becomes *Done*).

[6] Note that the modified translator includes miscommunication generation in the same way as the original translator.

The verification statistics for each of these models are presented in Table 6. These also include a calculated reduction factor indicating how much of a reduction was observed for each collected statistic between the models generated by the original translator and the modified one. Further, because previous results showed that the collected verification statistics increased exponentially with the maximum number of generated miscommunications in communication protocols [9], an R^2 statistic was calculated based on a fitted exponential function between the maximum number of miscommunications (as the independent variable) and both the number of visited states and verification times for models generated by each translator for each communication protocol.

These results show that, in all cases, the modified translator outperformed the original translator in terms of both the size of the model (number of visited states) and the total verification time. Excluding the PCA model, all of the other models saw an average reduction factor of approximately 2 for both measures. However, much higher reduction factors were found for the PCA model. The R^2 statistics for the communication protocol models were all evaluated close to 1. Thus, for communication protocols, the number of visited states and the verification time appear to scale exponentially with the maximum number of miscommunications for models generated by both translators.

7 Discussion

The modified translator does not prevent EOFMC models from scaling exponentially with the number of task structures or with the maximum number of allowable miscommunications. However, the benchmarks clearly indicate that the models created by the modified translator are smaller and verify faster than their counterparts created using the original translator. Thus, the use of the modified translator should improve the scalability of formal verification analyses that use EOFMC and thus enable more complex systems to ultimately be evaluated.

The variability observed in the results suggests that scalability improvements are dependent on the nature of the model. However, it does appear that the improvements in scalability of models produced by the modified translator become more apparent as the complexity of the EOFMC model instance increases. For example, in the artificial benchmarks (Table 5), the reduction factor between the two translated versions of the models for both verification metrics increased drastically as the number of task structures increased. This is reflected in the realistic benchmarks as well. Specifically, the PCA model has the most complex EOFMC instance of all of the other benchmarks and clearly saw a higher reduction factor for both verification metrics.

Table 5 Artificial verification benchmark results

Number of Tasks	Number of Visited States: Original	Number of Visited States: Modified	Reduction Factor	Percent Decrease	Verification Time (s): Original	Verification Time (s): Modified	Reduction Factor	Percent Decrease
1	92	18	5.11	80.43%	0.06	0.04	1.50	33.33%
2	1640	64	25.63	96.10%	0.13	0.07	1.86	46.15%
3	23,600	184	128.26	99.22%	0.39	0.13	3.00	66.67%
4	308,000	480	641.67	99.84%	2.07	0.18	11.50	91.30%
5	3,800,000	1184	3209.46	99.97%	10.34	0.30	34.47	97.10%
6	45,200,000	2816	16,051.14	99.99%	47.98	0.45	106.62	99.06%
7	524,000,000	6528	80,269.61	100.00%	48.02	0.62	77.45	98.71%
8	5,960,000,000	14,848	401,400.86	100.00%	1890.95	0.86	2198.78	99.95%
9	66,800,000,000	33,280	2,007,211.54	100.00%	12,475.62	1.28	9746.58	99.99%
10	740,000,000,000	73,728	10,036,892.36	100.00%	42,889.46	1.60	26,805.91	100.00%
	$\hat{y} = 10.96e^{2.51x}$ $R^2 \approx 1$	$\hat{y} = 10.70e^{0.90x}$ $R^2 \approx 1$			$\hat{y} = 0.01e^{1.55x}$ $R^2 = 0.98$	$\hat{y} = 0.03e^{0.41x}$ $R^2 = 0.99$		

Table 6 Realistic application verification benchmark results

Model	Number of Visited States: Original	Number of Visited States: Optimized	Reduction Factor	Percent Decrease	Verification Time (s): Original	Verification Time (s): Optimized	Reduction Factor	Percent Decrease
Aircraft	231	193	1.20	16.45%	0.41	0.37	1.11	9.76%
Cruise Control	11,964	5162	2.32	56.85%	0.77	0.33	2.33	57.14%
Therac 25	31,968	7200	4.44	77.48%	0.44	0.23	1.91	47.73%
PCA	4,072,083	15,388	264.63	99.62%	90.4	5.60	16.14	93.81%
Protocol 1 $_{Max=0}$	6351	2371	2.68	62.67%	1.98	0.98	2.02	50.51%
Protocol 1 $_{Max=1}$	75,806	39,844	1.90	47.44%	2.32	1.30	1.78	43.97%
Protocol 1 $_{Max=2}$	1,297,497	713,299	1.82	45.02%	2.92	1.51	1.93	48.29%
Protocol 1 $_{Max=3}$	2,697,676	1,483,300	1.82	45.02%	3.56	1.75	2.03	50.84%
	$\hat{y} = 8689.30e^{2.10x}$ $R^2 = 0.95$	$\hat{y} = 3578.30e^{2.22x}$ $R^2 = 0.95$			$\hat{y} = 1.95e^{0.20x}$ $R^2 \approx 1$	$\hat{y} = 1.02e^{0.19x}$ $R^2 = 0.97$		
Protocol 2 $_{Max=0}$	6250	2390	2.62	61.76%	1.67	0.98	1.70	41.32%
Protocol 2 $_{Max=1}$	59,935	35,843	1.67	40.20%	2.93	1.39	2.11	52.56%
Protocol 2 $_{Max=2}$	259,983	156,087	1.67	39.96%	3.61	2.06	1.75	42.94%
Protocol 2 $_{Max=3}$	1,013,966	635,336	1.60	37.34%	4.58	2.51	1.82	45.20%
	$\hat{y} = 8099.90e^{1.67x}$ $R^2 = 0.98$	$\hat{y} = 3510.60e^{1.82x}$ $R^2 = 0.97$			$\hat{y} = 1.85e^{0.32x}$ $R^2 = 0.94$	$\hat{y} = 1.01e^{0.32}$ $R^2 = 0.98$		
Protocol 3 $_{Max=0}$	3216	1221	2.63	62.03%	2.67	1.61	1.66	39.70%
Protocol 3 $_{Max=1}$	34,531	20,734	1.67	39.96%	4.73	2.87	1.65	39.32%
Protocol 3 $_{Max=2}$	142,001	103,678	1.37	26.99%	9.79	5.98	1.64	38.92%
Protocol 3 $_{Max=3}$	392,742	343,966	1.14	12.42%	18.72	11.08	1.69	40.81%
	$\hat{y} = 4643.10e^{1.58x}$ $R^2 = 0.96$	$\hat{y} = 1912.70e^{1.85x}$ $R^2 = 0.96$			$\hat{y} = 2.59e^{0.66x}$ $R^2 \approx 1$	$\hat{y} = 1.57e^{0.65x}$ $R^2 \approx 1$		

7.1 Counterexample interpretation

The original EOFMC translator supported a counterexample visualization to help analysts diagnose specification violations [13] (discussed in Sect. 2.1). Although not reported here, this same visualization was adapted for use with the modified translator. In this, the modified translator uses the new expressions representing execution state to determine what the execution state is of each activity in each counterexample step. Thus, the modified translator does not negatively impact the ability of analysts to interpret verification results when using EOFMC.

7.2 Erroneous human behavior generation

The original version of the EOFMC to SAL translator supported two methods for generating erroneous human behavior [15,17] beyond the miscommunication generation technique [9] that was retained in the new translator presented here. Given that erroneous behavior generation scales poorly

as the maximum number of erroneous behaviors increases [15,17], it could benefit from the statespace and verification time improvements of the new translator. Future work should investigate if EOFMC's erroneous behavior generation capabilities can be adapted to the new translation method.

7.3 Additional analysis options

In removing the intermediary state transitions between the performance of actions in the formal representation of EOFMC instances, the modified translator opens up possibilities for formal analyses that would not have been previously achievable. Specifically, the presence of the intermediary transitions discouraged the use of bounded model checking with EOFMC-based models, because they would prevent interesting behavior from occurring with reasonable upper search bounds. However, models produced with the modified translator do not contain these transitions and should thus be more appropriate for bounded analyses. Future work should explore this option.

A potential extension of this would be to include the modeling of time in EOFMC-supported verifications. In SAL, explicit time can be modeled for and evaluated with the infinite bounded model checker. Thus, the ability to do bounded model checking will also enable the explicit representation of time in EOFMC analyses. The ability to model explicit time could open up a range of formal analyses for evaluating interface usability [36] and timing-related safety concerns [38]. Of course, including explicit time in EOFMC-supported analyses will degrade scalability. Thus, future work should investigate how to include explicit time and evaluate its impact on scalability.

7.4 Future scalability improvements

The modified translator has shown itself to be successful at reducing the complexity of formal models that use EOFMC. However, further improvements might be possible. For example, EOFMC models currently use asynchronous composition to attach the modules created by the EOFMC translators to the larger formal system models. As part of this, the coordination protocol [12] is used to determine whether the human module or other modules are allowed to transition. The use of this protocol adds additional states, transitions, and verification time to the aggregated system model. The protocol could be eliminated by synchronously composing the formal task model module to the other modules of the system. Future work should investigate ways of making EOFMC work with synchronous composition, so that the states and transitions associated with the coordination protocol can be eliminated.

Other opportunities for scalability improvement may be offered through other analysis environments. For example, labeled transition systems like those supported by state charts [32] allow separate parallel model components to be combined together using labels to identify synchronized transitions. Such a system has shown itself to naturally support the modeling of human interactive systems [23–28,34,35]. Future work should investigate if labeled transition systems would afford additional improvements in EOFMC analysis scalability.

Finally, both the original and the modified translators separately represent the execution state of each task structure in an EOFMC instance. This is an intuitive approach. However, additional statespace improvements could potentially be gained by allowing task structures to share model resources. One potential approach would be to alter the architecture assumed by formal models using EOFMC. For example, EOFMC could use a synchronous observer architecture [45], where the module generated by the EOFMC translator would examine the global model state and each simulated human action as it occurred. This module would also be responsible for indicating if the current sequence of actions was valid. When such an implementation would be used in formal verifications, specifications would be formulated to only consider valid human action sequences. This architecture would be effectively treating the EOFMC model as an automaton that accepts strings of system conditions and human actions. Thus, automata theory could be used to find a minimal representation of the EOFMC instance so as to minimize its complexity. For example, algorithms such as L* [3] could potentially be used to automatically learn such minimal models. Future work should explore this direction.

Acknowledgments The project described was supported by NASA under award NNA10DE79C and the National Science Foundation under Grant No. IIS-1429910.

References

1. Aït-Ameur Y, Baron M (2006) Formal and experimental validation approaches in HCI systems design based on a shared event B model. Int J Softw Tools Technol Transfer 8(6):547–563
2. Aït-Ameur Y, Baron M, Girard P (2003) Formal validation of HCI user tasks. In: Proceedings of the international conference on software engineering research and practice. CSREA Press, Las Vegas, pp 732–738
3. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106
4. Basnyat S, Palanque P, Schupp B, Wright P (2007) Formal sociotechnical barrier modelling for safety-critical interactive systems design. Saf Sci 45(5):545–565
5. Basnyat S, Palanque PA, Bernhaupt R, Poupart E (2008) Formal modelling of incidents and accidents as a means for enriching training material for satellite control operations. In: Proceedings of the Joint ESREL 2008 and 17th SRA-Europe Conference, Taylor and Francis Group, London, pp CD–ROM
6. Bass EJ, Bolton ML, Feigh K, Griffith D, Gunter E, Mansky W, Rushby J (2011) Toward a multi-method approach to formalizing

human–automation interaction and human–human communications. In: Proceedings of the IEEE international conference on systems, man, and cybernetics. IEEE, Piscataway, pp 1817–1824
7. Bolton ML (2010) Using task analytic behavior modeling, erroneous human behavior generation, and formal methods to evaluate the role of human–automation interaction in system failure. PhD thesis, University of Virginia, Charlottesville
8. Bolton ML (2013) Automatic validation and failure diagnosis of human-device interfaces using task analytic models and model checking. Comput Math Organ Theory 19:288–312
9. Bolton ML (2015) Model checking human–human communication protocols using task models and miscommunication generation. J Aerosp Inf Syst. doi:10.2514/1.I010276
10. Bolton ML, Bass EJ (2009a) Building a formal model of a human-interactive system: insights into the integration of formal methods and human factors engineering. In: Proceedings of the 1st NASA formal methods symposium. NASA Ames Research Center, Moffett Field, pp 6–15
11. Bolton ML, Bass EJ (2009b) A method for the formal verification of human interactive systems. In: Proceedings of the 53rd annual meeting of the human factors and ergonomics society. HFES, Santa Monica, pp 764–768
12. Bolton ML, Bass EJ (2010a) Formally verifying human–automation interaction as part of a system model: limitations and tradeoffs. Innov Syst Softw Eng NASA J 6(3):219–231
13. Bolton ML, Bass EJ (2010) Using task analytic models to visualize model checker counterexamples. In: Proceedings of the 2010 IEEE international conference on systems, man, and cybernetics. IEEE, Piscataway, pp 2069–2074
14. Bolton ML, Bass EJ (2012) Using model checking to explore checklist-guided pilot behavior. Int J Aviat Psychol 22(4):343–366
15. Bolton ML, Bass EJ (2013) Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking. IEEE Trans Syst Man Cybern Syst 43(6):1314–1327
16. Bolton ML, Siminiceanu RI, Bass EJ (2011) A systematic approach to model checking human–automation interaction using task-analytic models. IEEE Trans Syst Man Cybern Part A 41(5):961–976
17. Bolton ML, Bass EJ, Siminiceanu RI (2012) Using phenotypical erroneous human behavior generation to evaluate human–automation interaction using model checking. Int J Hum Comput Stud 70(11):888–906
18. Bolton ML, Bass EJ, Siminiceanu RI (2013) Using formal verification to evaluate human–automation interaction in safety critical systems, a review. IEEE Trans Syst Man Cybern Syst 43(3):488–503
19. Bolton ML, Jimenez N, van Paassen MM, Trujillo M (2014) Automatically generating specification properties from task models for the formal verification of human–automation interaction. IEEE Trans Hum Mach Syst 44(5):561–575
20. Campos JC (2003) Using task knowledge to guide interactor specifications analysis. In: Proceedings of the 10th international workshop on interactive systems. Design, specification, and verification. Springer, Berlin, pp 171–186
21. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge
22. De Moura L, Owre S, Shankar N (2003) The SAL language manual. Technical report CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park
23. Degani A (2004) Taming HAL: designing interfaces beyond 2001. Macmillan, New York
24. Degani A, Heymann M (2002) Formal verification of human–automation interaction. Hum Factors 44(1):28–43
25. Degani A, Kirlik A (1995) Modes in human–automation interaction: initial observations about a modeling approach. In: Proceedings of the IEEE international conference on systems, man and cybernetics, vol 4. IEEE, Piscataway, pp 3443–3450
26. Degani A, Heymann M, Shafto M (1999a) Formal aspects of procedures: the problem of sequential correctness. In: Proceedings of the 43rd annual meeting of the human factors and ergonomics society. HFES, Santa Monica, pp 1113–1117
27. Degani A, Shafto M, Kirlik A (1999b) Modes in human–machine systems: review, classification, and application. Int J Aviat Psychol 9(2):125–138
28. Degani A, Gellatly A, Heymann M (2011) HMI aspects of automotive climate control systems. In: Proceeding of the IEEE international conference on systems, man, and cybernetics. IEEE, Piscataway, pp 1795–1800
29. Emerson EA (1990) Temporal and modal logic. In: van Leeuwen J, Meyer AR, Nivat M, Paterson M, Perrin D (eds) Handbook of theoretical computer science, chapter 16. MIT Press, Cambridge, pp 995–1072
30. Fields RE (2001) Analysis of erroneous actions in the design of critical systems. PhD thesis, University of York, York
31. Gunter EL, Yasmeen A, Gunter CA, Nguyen A (2009) Specifying and analyzing workflows for automated identification and data capture. In: Proceedings of the 42nd Hawaii international conference on system sciences. IEEE Computer Society, Los Alatimos, pp 1–11
32. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274
33. Hartson HR, Siochi AC, Hix D (1990) The UAN: a user-oriented representation for direct manipulation interface designs. ACM Trans Inf Syst 8(3):181–203
34. Heymann M, Degani A (2007) Formal analysis and automatic generation of user interfaces: approach, methodology, and an algorithm. Hum Factors 49(2):311–330
35. Heymann M, Degani A, Barshi I (2007) Generating procedures and recovery sequences: a formal approach. In: Proceedings of the 14th international symposium on aviation psychology. Wright State University, Dayton
36. John BE (2009) CogTool user guide. Carnegie Mellon University, Pittsburgh
37. Kirwan B, Ainsworth LK (1992) A guide to task analysis. Taylor and Francis, London
38. Leveson NG, Turner CS (1993) An investigation of the therac-25 accidents. Computer 26(7):18–41
39. Li M, Molinaro K, Bolton ML (2015) Learning formal human–machine interface designs from task analytic models. In: Proceedings of the HFES annual meeting. HFES, Santa Monica (in press)
40. Mitchell CM, Miller RA (1986) A discrete control model of operator function: a methodology for information display design. IEEE Trans Syst Man Cybern Part A Syst Hum 16(3):343–357
41. Palanque PA, Bastide R, Senges V (1996) Validating interactive system design through the verification of formal task and system models. In: Proceedings of the IFIP TC2/WG2.7 working conference on engineering for human–computer interaction. Chapman and Hall, London, pp 189–212
42. Paternò F, Santoro C (2001) Integrating model checking and HCI tools to help designers verify user interface properties. In: Proceedings of the 7th international workshop on the design, specification, and verification of interactive systems. Springer, Berlin, pp 135–150
43. Paternò F, Mancini C, Meniconi S (1997) Concurtasktrees: a diagrammatic notation for specifying task models. In: Proceedings of the IFIP TC13 international conference on human–computer interaction. Chapman and Hall, London, pp 362–369
44. Paternò F, Santoro C, Tahmassebi S (1998) Formal model for cooperative tasks: concepts and an application for en-route air traffic control. In: Proceedings of the 5th international conference on

the design, specification, and verification of interactive systems. Springer, Vienna, pp 71–86

45. Rushby J (2014) The versatile synchronous observer. In: Iida S, Meseguer J, Ogata K (eds) Specification, algebra, and software: essays dedicated to Kokichi Futatsugi. Springer, Berlin, pp 110–128

46. Wing JM (1990) A specifier's introduction to formal methods. Computer 23(9):8, 10–22, 24