Chapter 13
Enhanced Operator Function Model (EOFM): A Task Analytic Modeling Formalism for Including Human Behavior in the Verification of Complex Systems

Matthew L. Bolton and Ellen J. Bass

Abstract The enhanced operator function model (EOFM) is a task analytic modeling formalism that allows human behavior to be included in larger formal system models to support the formal verification of human interactive systems. EOFM is an expressive formalism that captures the behavior of individual humans or, with the EOFM with communications (EOFMC) extension, teams of humans as a collection of tasks, each composed representing a hierarchy of activities and actions. Further, EOFM has a formal semantics and associated translator that allow its represented behavior to be automatically translated into a model checking formalism for use in larger system verification. EOFM supports a number of features that enable analysts to use model checking to investigate human-automation and human-human interaction. Translator variants support the development of different task models with methods for accounting for erroneous human behaviors and miscommunications, the creation of specification properties, and the automated design of human-machine interfaces. This chapter provides an overview of EOFM, its language, its formal semantics and translation, and analysis features. It addresses the different ways that EOFM has been used to evaluate human behavior in human-interactive systems. We demonstrate some of the capabilities of EOFM by using it to evaluate the air traffic control case study. Finally, we discuss future directions of EOFM and its supported analyses.

M.L. Bolton (✉)
University at Buffalo, State University of New York, Buffalo, NY, USA
e-mail: mbolton@buffalo.edu

E.J. Bass
Drexel University, Philadelphia, PA, USA
e-mail: ejb96@drexel.edu

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_13

13.1 Introduction

In complex systems, failures often occur as a result of unexpected interactions between components including contributions from human behaviors (Hollnagel 1993; Perrow 1999; Reason 1990; Sheridan and Parasuraman 2005). To design and analyze complex systems, human factors engineers use task analysis (Kirwan and Ainsworth 1992) to describe required normative human behaviors. Erroneous human behavior models provide engineers with means for exploring the potential impact of human error (Hollnagel 1993; Reason 1990). To support design and analysis, Enhanced Operator Function Model (EOFM) (Bolton et al. 2011), based on the Operator Function Model (Mitchell and Miller 1986), was developed to allow engineers to represent task analytic human behavior formally and to include both normative and erroneous human behavior in formal verification analyses. EOFM with communication (EOFMC) further supports models with coordination and communication among a team of people (Bass et al. 2011).

The analyses supported by EOFM and EOFMC have evolved from older techniques that use formal verification to evaluate human-automation interaction (Bolton et al. 2013). In particular, EOFM is similar to techniques that attempt to include human behavior in formal verification analyses by using formal interpretations of task analytic models. As such, EOFM supports similar sorts of evaluations offered by other systems such as AMBOSS (Giese et al. 2008), ConcurTaskTrees (Aït-Ameur and Baron 2006; Paternò and Santoro 2001; Paternò et al. 1997), User Action Notation (Hartson et al. 1990; Palanque et al. 1996), HAMSTER (Martinie et al. 2011, 2014), and various approaches that require task concepts to be directly represented in other modeling formalisms (Basnyat et al. 2007; Degani et al. 1999; Fields 2001; Gunter et al. 2009). However, EOFM and EOFMC distinguish themselves by the rich feature set and the analyses they support. EOFM and EOFMC have formal semantics that give the task behaviors specified by the XML language unambiguous, mathematical meanings. The semantics serve as the basis for a series of translators that automatically convert EOFMs, written in XML, into formal models that can be used by a model checker. These translators support a number of different features that generate alternate task models with erroneous behavior, properties that can be model checked, and interface designs.[1] A deeper discussion about the place of EOFM in the larger formal HAI literature can be found in Bolton et al. (2011, 2013).

This chapter provides a general overview of EOFM and EOFMC. This includes a description of the EOFM-supported analysis process, EOFM and EOFMC syntax, formal semantics, translation, and a description of different analyses that leverage EOFM and EOFMC. As concepts are introduced, we illustrate some of EOFMC's capabilities with a variation of the air traffic control case study. Below, the case study is introduced. This is followed by a discussion of EOFM and its capabilities along

[1] The EOFM language specifications, translators, tools, documentation, and examples are freely available at http://fhsl.eng.buffalo.edu/EOFM/.

with applications to the case study. Finally, both the case study analysis results and EOFM are generally discussed with pointers to additional applications and descriptions of future research directions.

13.2 Case Study

To illustrate how EOFM can be used to evaluate a safety critical procedure involving human-human communication and coordination, we present a variation of Case Study 2. Specifically, we present an EOFMC model where an aircraft heading clearance is communicated by an air traffic controller (ATCo) to two pilots: the pilot flying (PF) and the pilot monitoring (PM).

In this scenario, both the pilots and the air traffic controller have push-to-talk switches which they press down when they want to verbally communicate information to each other over the radio. They can release this switch to end communication.

For the aircraft, the Autopilot Flight Director System consists of Flight Control Computers and the Mode Control Panel (MCP). The MCP provides control of the Autopilot (A/P), Flight Director, and the Autothrottle system. When the A/P is engaged, the MCP sends commands to the aircraft pitch and roll servos to operate the aircraft flight control surfaces. Herein the MCP is used to activate heading changes. The Heading (HDG)/Tracking (TRK) window of the MCP displays the selected heading or track (Fig. 13.1). The numeric display shows the current desired heading in compass degrees (from 0 and 359). Below the HDG window is the heading select knob. Changes in the heading are achieved by rotating and pulling the knob. Pulling the knob tells the autopilot to use the selected value and engages the HDG mode.

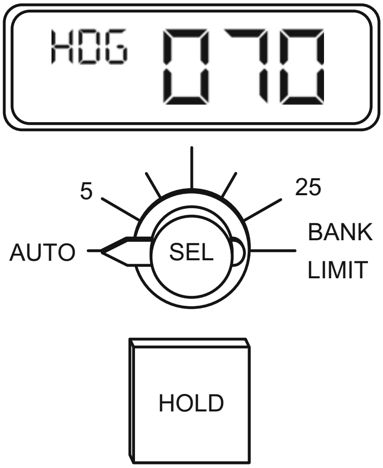


Fig. 13.1 Heading control and display

The following describes a communication protocol designed to ensure that this heading is correctly communicated from the ATCo to the two pilots:

1. The air traffic controller contacts the pilots and gives them a new heading clearance.
2. The PF starts the process of dialing the new heading into the heading window.
3. The pilots then attempt to confirm that the correct heading was entered. They do this by having the PM point at the heading window, contact the ATCo, and repeat back the heading the PM heard from the ATCo.
4. If the heading the PF hears read back to the air traffic controller does not match the heading he or she entered in the heading window, the PF enters the heading heard from the PM during the read back. The PM then points at the heading window again and repeats the heading he or she originally heard from the ATCo. This process (step 4) repeats while the heading window heading does not match what the PM heard from the ATCo. It completes if the heading the PF heard from the PM matches what is in the heading window.
5. If the heading the ATCo hears read back from the pilots (from step 3) does not match the heading the air traffic controller intended, then the entire process (starting at step 1) needs to be repeated until the correct heading is read back.
6. The PF engages the entered heading.

In what follows, we will show how EOFM concepts can be applied to this application and various EOFM features are introduced.

13.3 Enhanced Operator Function Model (EOFM) and EOFM with Communication (EOFMC)

EOFM and EOFMC (henceforth collectively referred to as EOFM except where additional clarification about EOFMC is required) are task analytic modeling languages for representing human task behavior. The EOFM languages support the formal verification approach shown in Fig. 13.2. An analyst examines target system information (i.e. design document, observational data, manuals) to manually

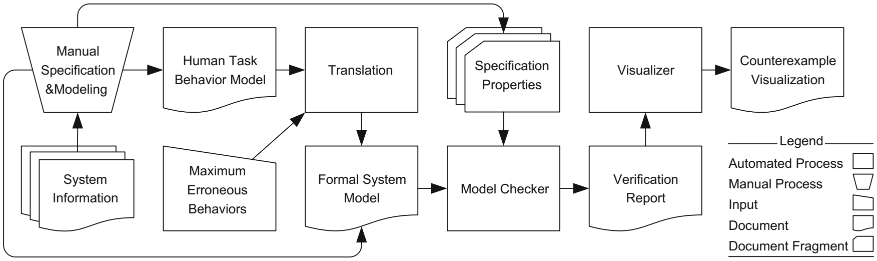


Fig. 13.2 Flow diagram showing how the verification method supported by EOFM works

model normative human operator behavior (which can be a single human operator or a team of human operators) using a task analytic representation, a formal system model (absent human operator behavior), and specification properties that he or she wants to check. The task analytic model is then automatically translated into the larger formal model by a translation process. As part of this translation process, the analyst can specify a maximum number of erroneous behaviors that will be modeled as optional paths through the formal representation of the human operator task behavior. The formal system model and the specifications are then run through the model checker that produces a verification report. If a violation of the specification is found, the report will contain a counterexample. Our visualizer uses the counterexample and the original human task behavior model to illustrate the sequence of human behaviors and related system states that led to the violation.

EOFM has an XML syntax that supports the hierarchical concepts compatible with task analytic models. This represents task behavior as a hierarchy of goal-directed activities (representing the behaviors and strategic knowledge required to accomplish a goal) that can decompose into sub-activities (for achieving sub-goals) and, at the bottom of the hierarchy, atomic actions (specific things the human operator(s) can do to the environment). The language allows for the modeling of human behavior, either individuals or groups, as an input/output system. Inputs may come from other elements of the system like human-device interfaces or the environment. Output variables are human actions. The operators' task models describe how human actions may be generated based on input and local variables (representing perceptual or cognitive processing as well as group coordination and communication). To support domain specific information, variables are defined in terms of constants, analyst-defined types, and basic types (reals, integers, and Boolean values).

To represent the strategic knowledge associated with procedural activities, activities can have preconditions, repeat conditions, and completion conditions (Boolean expressions written in terms of input, output, and local variables as well as constants) that specify what must be true before an activity can execute, when it can execute again, and what is true when it has completed execution respectively. Activities are decomposed into lower-level sub-activities and, finally, actions. Decomposition operators specify how many sub-activities or actions can execute and what the temporal relationships are between them. Actions are either an assignment to an output variable (indicating an action has been performed) or a local variable (representing a perceptual, cognitive, or group communication action). Optionally an action can include a value so that the human operator can set a value as part of the action.

All tasks in an EOFM descend from a top level activity, where there can be many tasks. Tasks can either belong to one human operator, or, in EOFMC, they can be shared among human operators. A shared task must be associated with two or more associates, and a subset of associates for the general task can be identified for each activity. This makes it explicit which human operators are participating in which activity. While the activities in these shared tasks can decompose in the same ways as their single-operator counterparts, they can explicitly include human-human communication (a non-shared activity cannot). For communication, a value (communicated information) from one human operator can be received by one or more other human operators (modeled as an update to a local variable).

13.3.1 Syntax

The EOFM language's XML syntax is defined using the REgular LAnguage for XML Next Generation (RELAX NG) standard (Clark and Murata 2001) (see Figs. 13.3, 13.4 and 13.5). These provide an explicit description of the EOFM language and can thus be employed by analysts when developing their own EOFM and EOFMC models.[2] Below we describe the EOFMC syntax. An example of the syntax used in an actual model is shown in Fig. 13.7, with highlights showing specific features.

XML documents contain a single root node whose attributes and sub-nodes define the document. For the EOFM, 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 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 basictype attribute. A constant node also allows for the definition of constant mappings and functions through the addition of optional (zero or more) parameter attributes for defining function arguments.

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) or human communication actions (humancomaction 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 three values: autoreset (for modeling a single discrete action such as flipping a switch), 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), or setvalue for committing a value in a single action. This type of human action has an additional attribute that specifies the type of the value being set

[2] Note that although EOFM does not have its own language creation tool, the RELAX NG language specification allows professional XML development environments (such as oXygen XML Editor; https://www.oxygenxml.com/) to facilitate rapid model development with code completion and syntax checking capabilities.

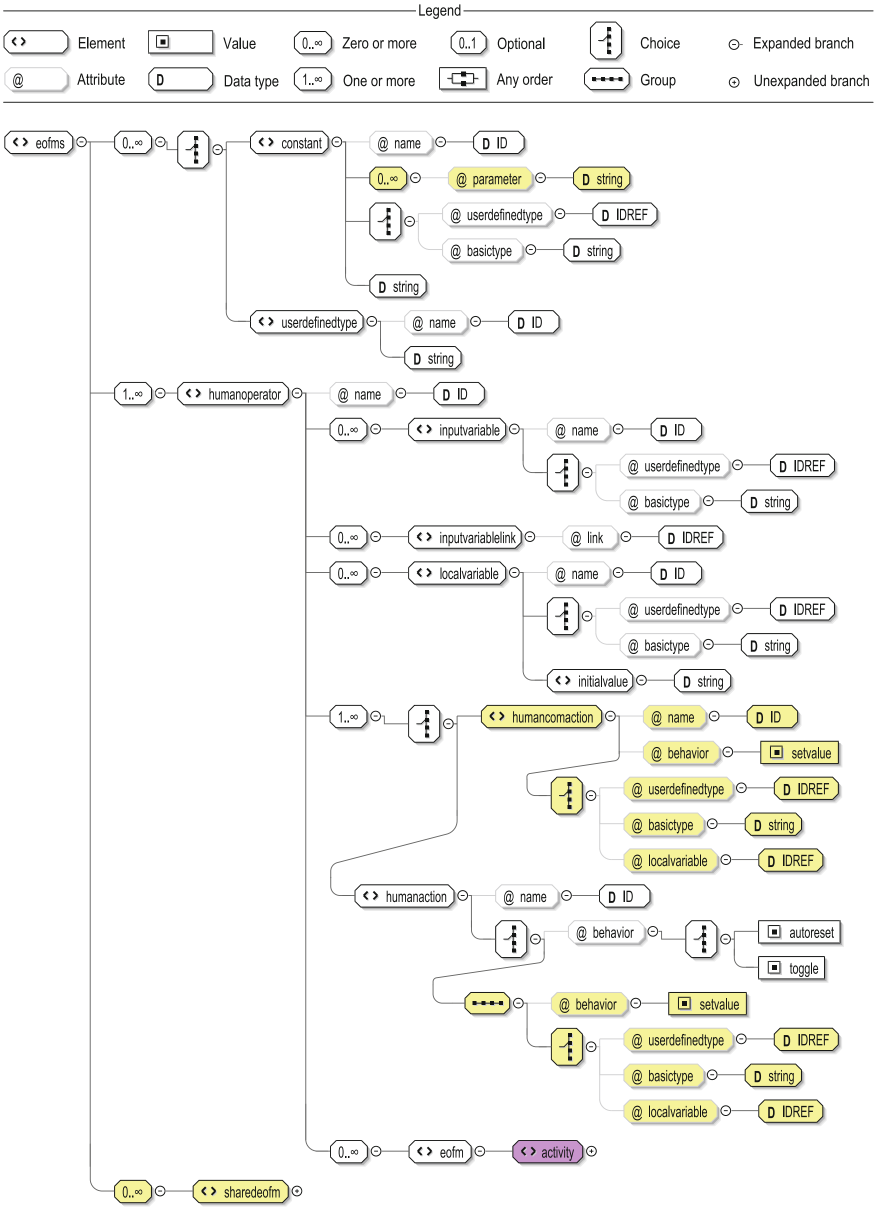


Fig. 13.3 A visualization of the EOFM Relax NG language specification that species EOFM syntax (see Syncro Soft 2016 for more details). *Yellow* indicates constructs added to EOFM for EOFMC. *Other colors* are used to indicate nodes representing identically defined syntax elements across Figs. 13.3, 13.4 and 13.5. The language specification for activity and sharedeofm nodes can be found in Figs. 13.4 and 13.5 respectively

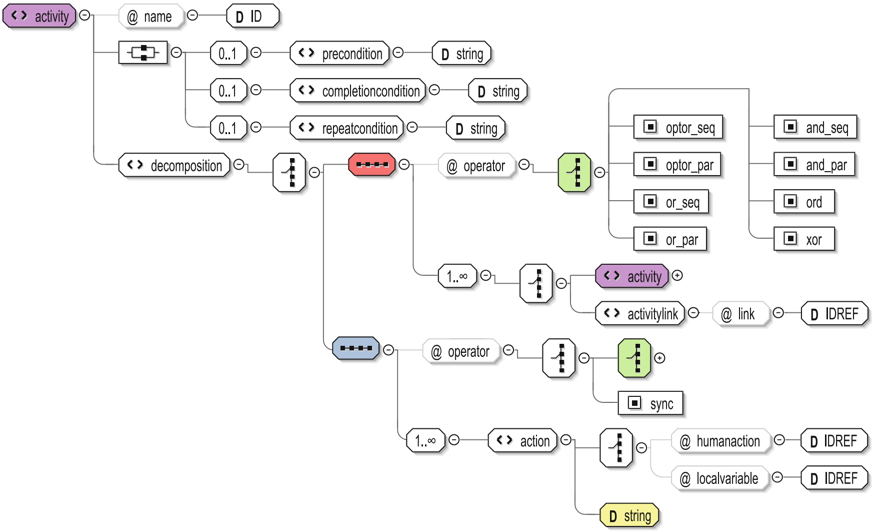


Fig. 13.4 Syntax visualization of the activity node from Fig. 13.3

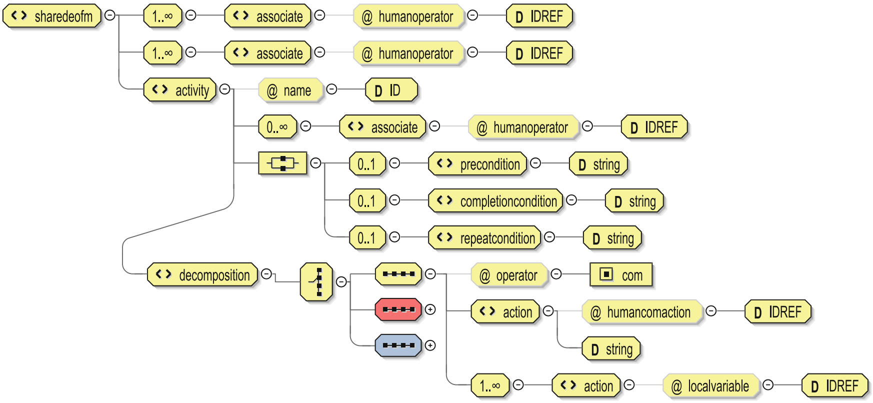


Fig. 13.5 Syntax visualization of the sharedeofm node from Fig. 13.3

(a reference to a userdefinedtype or a basictype) or a reference to a local variable which will contain the value to be set. A human communication action (a humancomaction node) describes a special type of setvalue action that a human operator can use to communicate something (a value or the value stored in an associated variable) to one or more other human operators.

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 sub-node, initialvalue, 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 and action 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:

- *Precondition* (precondition in the XML): criterion to start executing;
- *RepeatCondition* (repeatcondition in the XML): criterion to repeat execution; and
- *CompletionCondition* (completioncondition in the XML): criterion to complete 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 (Table 13.1) controls the cardinal and

Table 13.1 Decomposition operators

Operator	Description
optor_seq	Zero or more of the activities or actions in the decomposition must execute in any order one at a time
optor_par	Zero or more of the activities or actions in the decomposition must execute in any order and can execute in parallel
or_seq	One or more of the activities or actions in the decomposition must execute in any order one at a time
or_par	One or more of the activities or actions in the decomposition must execute in any order and can execute in parallel
and_seq	All of the activities or actions in the decomposition must execute in any order one at a time
and_par	All of the activities or actions in the decomposition must execute in any order and can execute in parallel
xor	Exactly one activities or actions in the decomposition must execute
ord	All activities or actions in the decomposition must execute in the order they appear
sync	All activities or actions in the decomposition must execute synchronously
com	All activities or actions in the decomposition must execute synchronously, where information is transferred between human operators

temporal execution relationships between the sub-activity and action nodes in the decomposition (its children). The EOFM language implements the following decomposition operators: and, or, optor, xor, ord, and sync. The com decomposition operator is exclusive to EOFMC. Some operators have two modalities: sequential (suffixed_seq) and parallel (suffixed_par). For the sequential mode, the activities or actions must be executed one at a time. In parallel mode, the execution of activities and action in the decomposition may overlap in any manner. For the xor, ord, sync, and com decomposition operators, only one modality can be defined: xor and ord are always sequential and sync and com are always parallel.

The activity nodes represent lower-level sub-activities and are defined identically to those higher in the hierarchy. Activity links (activitylink nodes) allow for 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.

Shared tasks are defined in sharedeofm nodes. Each sharedeofm contains two or more associate nodes explicitly defining which human operators collaborate to perform the shared tasks defined within these nodes. Tasks themselves are represented with the same hierarchy of activities and actions as in individual human operator tasks. The associates of each activity must be a subset of the associates of its ancestors. If no associates are defined in an activity, the associates are inherited from its parent node. Each action node is associated with a humanaction, localvariable, or communicationaction defined under the different human operators. Thus an action in a shared task is already affiliated with a humanoperator and does not require an explicitly defined associate.

While the activities in these shared tasks can decompose in the same ways as their single-operator counterparts, they have an additional decomposition option. The com decomposition operator explicitly models human-human communication. Assuming the associates are participating in the communication, com is a special form of the sync decomposition operator and assumes information is being transferred between human operators. The decomposition must start with the performance of a humancomaction, which commits a value (communicated information either explicitly defined in the XML markup or from a variable originally associated with the humancomaction when it was defined). The decomposition ends with one or more actions explicitly pointing to local variables to allow other human operators to register the information communicated via the humancomaction.

13.3.2 Visual Notation

The structure of an instantiated[3] EOFMC's task behaviors can be represented visually as a tree-like graph structure, where actions are depicted by rectangular nodes and activities by rounded rectangle nodes (see Fig. 13.6). In these visualizations, 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. These standard colors are used for condition shapes to help distinguish them from each other and the other task structures. A decomposition is presented as an arrow, labeled with the decomposition operator, extending below an activity that points to a large rounded rectangle containing the decomposed activities or actions. Activities are labeled with the associated activity name. Actions are labeled with the name of the associated humanaction, localvariable, or humancomaction name. Values committed by a humanaction with set value behavior, assigned to a localvariable assignment, or communicated by a humancomaction are shown in bold below this label. Communication actions are presented in the same decomposition as the local variables used for receiving the communication. Arrows are used to show how the communicated value is sent to the other local variables in the decomposition. These visualizations can be automatically generated from the XML source, as is currently done with a MS Visio plugin we have created (Bolton and Bass 2010c). The visualizations are useful because they help communicate encapsulated task concepts in publications and presentations. They also facilitate counterexample visualization, something discussed in more depth in Sect. 13.3.8.

13.3.3 Case Study Model

We implemented the task from our communication protocol application in EOFMC. Figure 13.6 depicts its visualization while Fig. 13.7 shows the corresponding XML. The entire process starts when the ATCo has a new, correct clearance to communicate to the pilots (lATCoClearance = CorrectHeading; the precondition to aChangeHeading). This allows for the performance of aChangeHeading and its first subactivity aCommandAndConfirm. The ATCo performs the aToggleATCoTalk activity by pressing his push-to-talk switch (hATCoToggleTalkSwitch). Then, the ATCo communicates the heading (aToggleATCoTalk[4]) to the pilots (lATCoClearance via the sATCoTalk human communication action), such as “AC1 Heading 070 for spacing.” Both pilots remember this heading (stored in the local variables lPFHead-

[3] An EOFM model created using the EOFM XML-based language.

[4] In this example, all headings are modeled abstractly as either being CorrectHeading, if it matches the heading clearance the ATCo intended to communicate, or IncorrectHeading, if it does not.

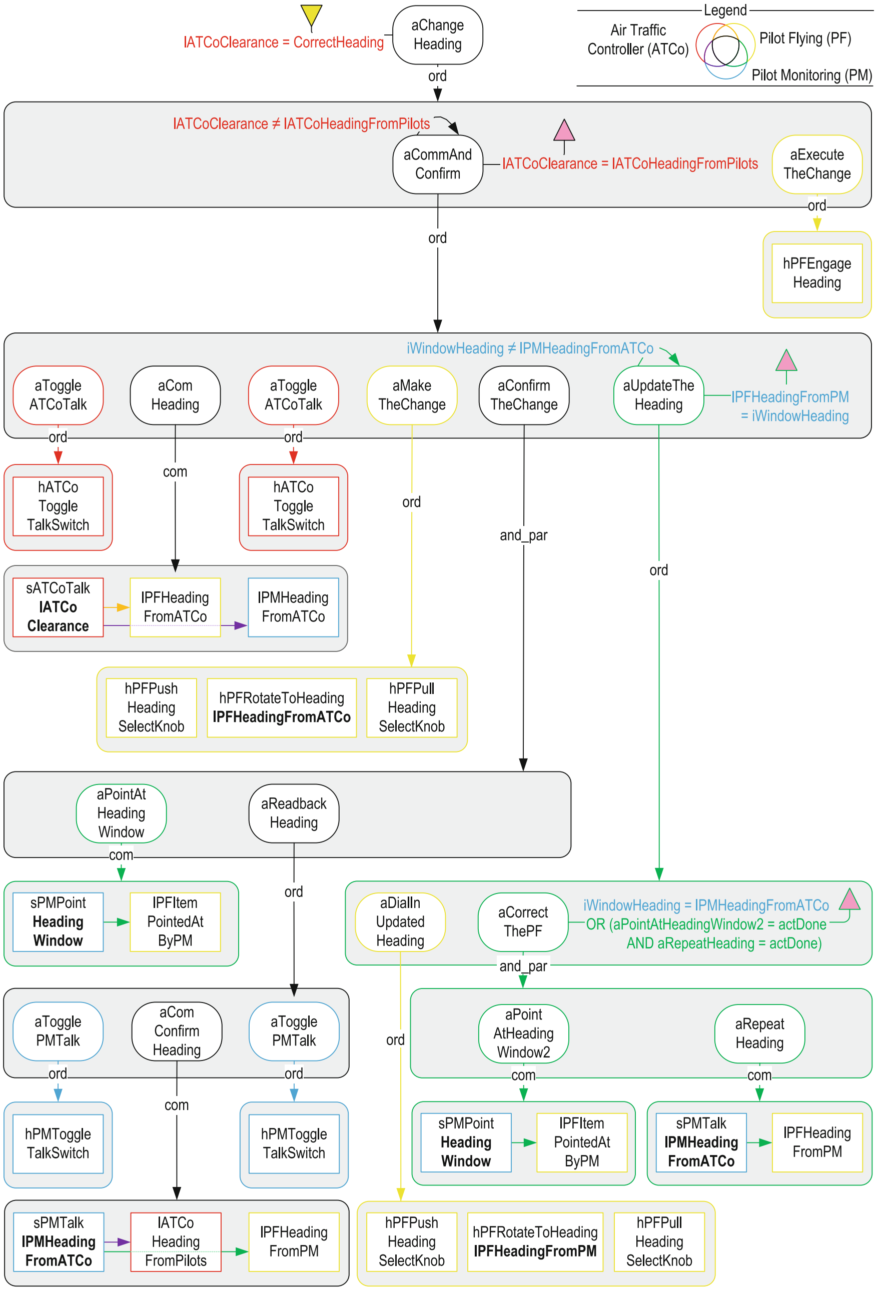
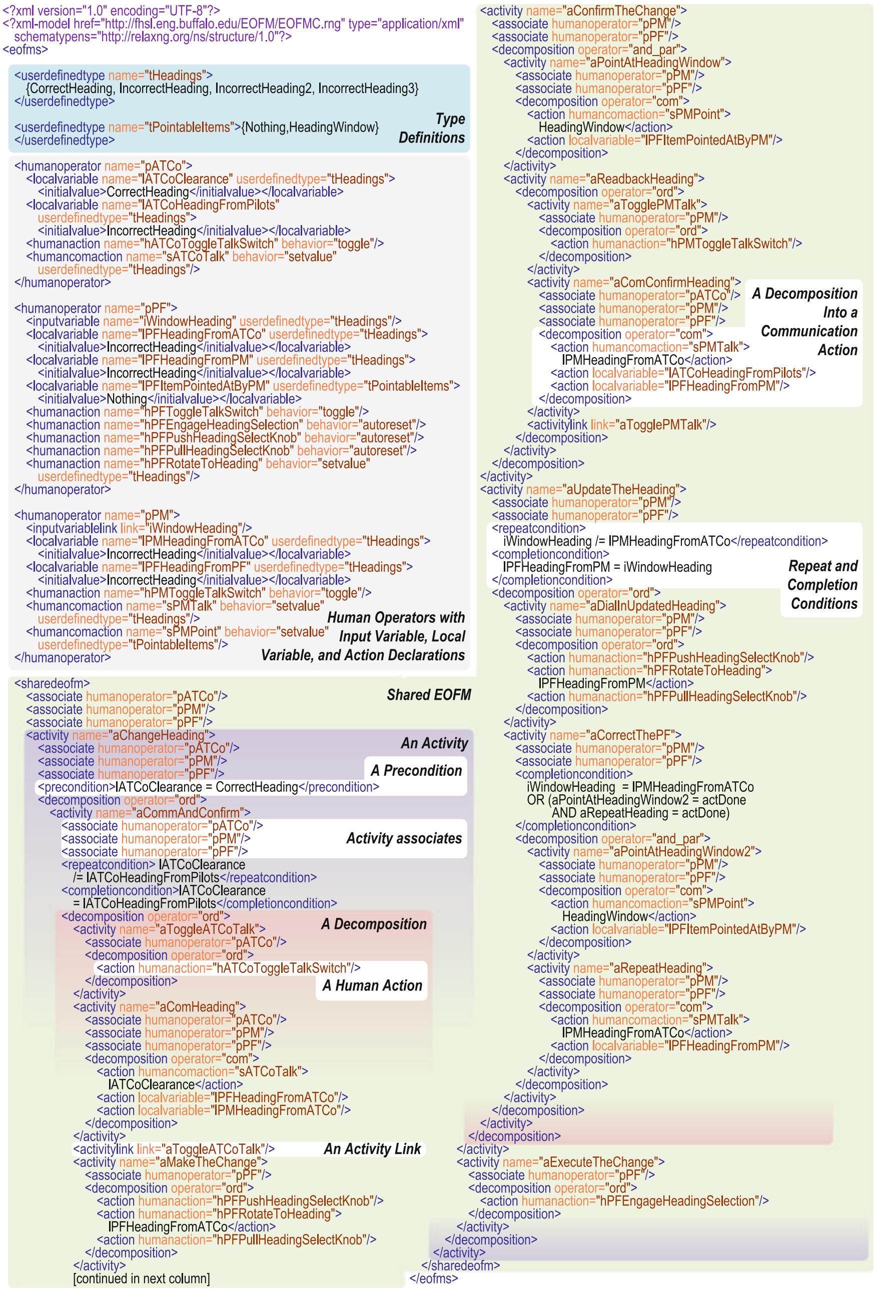


Fig. 13.6 Visualization of the instantiated EOFMC communication protocol from Fig. 13.7. The task is *colored* based on the associates (human operators) performing the task (see the legend)



```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="http://fhsl.eng.buffalo.edu/EOFM/EOFMC.rng" type="application/xml"
  schematypens="http://relaxng.org/ns/structure/1.0"?>
<eofms>

 <userdefinedtype name="tHeadings">
   {CorrectHeading, IncorrectHeading, IncorrectHeading2, IncorrectHeading3}
 </userdefinedtype>

 <userdefinedtype name="tPointableItems">{Nothing,HeadingWindow}
 </userdefinedtype>

 <humanoperator name="pATCo">
   <localvariable name="lATCoClearance" userdefinedtype="tHeadings">
     <initialvalue>CorrectHeading</initialvalue></localvariable>
   <localvariable name="lATCoHeadingFromPilots"
     userdefinedtype="tHeadings">
     <initialvalue>IncorrectHeading</initialvalue></localvariable>
   <humanaction name="hATCoToggleTalkSwitch" behavior="toggle"/>
   <humancomaction name="sATCoTalk" behavior="setvalue"
     userdefinedtype="tHeadings"/>
 </humanoperator>

 <humanoperator name="pPF">
   <inputvariable name="iWindowHeading" userdefinedtype="tHeadings"/>
   <localvariable name="lPFHeadingFromATCo" userdefinedtype="tHeadings">
     <initialvalue>IncorrectHeading</initialvalue></localvariable>
   <localvariable name="lPFHeadingFromPM" userdefinedtype="tHeadings">
     <initialvalue>IncorrectHeading</initialvalue></localvariable>
   <localvariable name="lPFItemPointedAtByPM" userdefinedtype="tPointableItems">
     <initialvalue>Nothing</initialvalue></localvariable>
   <humanaction name="hPFToggleTalkSwitch" behavior="toggle"/>
   <humanaction name="hPFEngageHeadingSelection" behavior="autoreset"/>
   <humanaction name="hPFPushHeadingSelectKnob" behavior="autoreset"/>
   <humanaction name="hPFPullHeadingSelectKnob" behavior="autoreset"/>
   <humanaction name="hPFRotateToHeading" behavior="setvalue"
     userdefinedtype="tHeadings"/>
 </humanoperator>

 <humanoperator name="pPM">
   <inputvariablelink link="iWindowHeading"/>
   <localvariable name="lPMHeadingFromATCo" userdefinedtype="tHeadings">
     <initialvalue>IncorrectHeading</initialvalue></localvariable>
   <localvariable name="lPFHeadingFromPF" userdefinedtype="tHeadings">
     <initialvalue>IncorrectHeading</initialvalue></localvariable>
   <humanaction name="hPMToggleTalkSwitch" behavior="toggle"/>
   <humancomaction name="sPMTalk" behavior="setvalue"
     userdefinedtype="tHeadings"/>
   <humancomaction name="sPMPoint" behavior="setvalue"
     userdefinedtype="tPointableItems"/>
 </humanoperator>

 <sharedeofm>
   <associate humanoperator="pATCo"/>
   <associate humanoperator="pPM"/>
   <associate humanoperator="pPF"/>
   <activity name="aChangeHeading">
     <associate humanoperator="pATCo"/>
     <associate humanoperator="pPM"/>
     <associate humanoperator="pPF"/>
     <precondition>lATCoClearance = CorrectHeading</precondition>
     <decomposition operator="ord">
       <activity name="aCommAndConfirm">
         <associate humanoperator="pATCo"/>
         <associate humanoperator="pPM"/>
         <associate humanoperator="pPF"/>
         <repeatcondition> lATCoClearance
           /= lATCoHeadingFromPilots</repeatcondition>
         <completioncondition>lATCoClearance
           = lATCoHeadingFromPilots</completioncondition>
         <decomposition operator="ord">
           <activity name="aToggleATCoTalk">
             <associate humanoperator="pATCo"/>
             <decomposition operator="ord">
               <action humanaction="hATCoToggleTalkSwitch"/>
             </decomposition>
           </activity>
           <activity name="aComHeading">
             <associate humanoperator="pATCo"/>
             <associate humanoperator="pPM"/>
             <associate humanoperator="pPF"/>
             <decomposition operator="com">
               <action humancomaction="sATCoTalk">
                 lATCoClearance</action>
               <action localvariable="lPFHeadingFromATCo"/>
               <action localvariable="lPMHeadingFromATCo"/>
             </decomposition>
           </activity>
           <activitylink link="aToggleATCoTalk"/>
           <activity name="aMakeTheChange">
             <associate humanoperator="pPF"/>
             <decomposition operator="ord">
               <action humanaction="hPFPushHeadingSelectKnob"/>
               <action humanaction="hPFRotateToHeading">
                 lPFHeadingFromATCo</action>
               <action humanaction="hPFPullHeadingSelectKnob"/>
             </decomposition>
           </activity>
           [continued in next column]
```

```
           <activity name="aConfirmTheChange">
             <associate humanoperator="pPM"/>
             <associate humanoperator="pPF"/>
             <decomposition operator="and_par">
               <activity name="aPointAtHeadingWindow">
                 <associate humanoperator="pPM"/>
                 <associate humanoperator="pPF"/>
                 <decomposition operator="com">
                   <action humancomaction="sPMPoint">
                     HeadingWindow</action>
                   <action localvariable="lPFItemPointedAtByPM"/>
                 </decomposition>
               </activity>
               <activity name="aReadbackHeading">
                 <decomposition operator="ord">
                   <activity name="aTogglePMTalk">
                     <associate humanoperator="pPM"/>
                     <decomposition operator="ord">
                       <action humanaction="hPMToggleTalkSwitch"/>
                     </decomposition>
                   </activity>
                   <activity name="aComConfirmHeading">
                     <associate humanoperator="pATCo"/>
                     <associate humanoperator="pPM"/>
                     <associate humanoperator="pPF"/>
                     <decomposition operator="com">
                       <action humancomaction="sPMTalk">
                         lPMHeadingFromATCo</action>
                       <action localvariable="lATCoHeadingFromPilots"/>
                       <action localvariable="lPFHeadingFromPM"/>
                     </decomposition>
                   </activity>
                   <activitylink link="aTogglePMTalk"/>
                 </decomposition>
               </activity>
             </decomposition>
           </activity>
           <activity name="aUpdateTheHeading">
             <associate humanoperator="pPM"/>
             <associate humanoperator="pPF"/>
             <repeatcondition>
               iWindowHeading /= lPMHeadingFromATCo</repeatcondition>
             <completioncondition>
               lPFHeadingFromPM = iWindowHeading
             </completioncondition>
             <decomposition operator="ord">
               <activity name="aDialInUpdatedHeading">
                 <associate humanoperator="pPM"/>
                 <associate humanoperator="pPF"/>
                 <decomposition operator="ord">
                   <action humanaction="hPFPushHeadingSelectKnob"/>
                   <action humanaction="hPFRotateToHeading">
                     lPFHeadingFromPM</action>
                   <action humanaction="hPFPullHeadingSelectKnob"/>
                 </decomposition>
               </activity>
               <activity name="aCorrectThePF">
                 <associate humanoperator="pPM"/>
                 <associate humanoperator="pPF"/>
                 <completioncondition>
                   iWindowHeading  = lPMHeadingFromATCo
                   OR (aPointAtHeadingWindow2 = actDone
                       AND aRepeatHeading = actDone)
                 </completioncondition>
                 <decomposition operator="and_par">
                   <activity name="aPointAtHeadingWindow2">
                     <associate humanoperator="pPM"/>
                     <associate humanoperator="pPF"/>
                     <decomposition operator="com">
                       <action humancomaction="sPMPoint">
                         HeadingWindow</action>
                       <action localvariable="lPFItemPointedAtByPM"/>
                     </decomposition>
                   </activity>
                   <activity name="aRepeatHeading">
                     <associate humanoperator="pPM"/>
                     <associate humanoperator="pPF"/>
                     <decomposition operator="com">
                       <action humancomaction="sPMTalk">
                         lPMHeadingFromATCo</action>
                       <action localvariable="lPFHeadingFromPM"/>
                     </decomposition>
                   </activity>
                 </decomposition>
               </activity>
             </decomposition>
           </activity>
         </decomposition>
       </activity>
       <activity name="aExecuteTheChange">
         <associate humanoperator="pPF"/>
         <decomposition operator="ord">
           <action humanaction="hPFEngageHeadingSelection"/>
         </decomposition>
       </activity>
     </decomposition>
   </activity>
 </sharedeofm>
</eofms>
```

Type Definitions
Human Operators with Input Variable, Local Variable, and Action Declarations
Shared EOFM
An Activity
A Precondition
Activity associates
A Decomposition
A Human Action
An Activity Link
A Decomposition Into a Communication Action
Repeat and Completion Conditions

Fig. 13.7 EOFM XML for the communication protocol application. Specific EOFM language features are highlighted and labeled with *bolded italic* text

ingFromATCo and lPMHeadingFromATCo for the PF and PM respectively). The ATCo releases the switch (under the second instance of aToggleATCoTalk; an activitylink in Fig. 13.7). Then, the PF performs the procedure for changing the heading in the heading window (aMakeTheChange): pushing the heading select knob (hPFPushHeadingSelectKnob), rotating it (hPFRotateToHeading) to the heading that the PF heard from the ATCo (lPFHeadingFromATCo), and pulling the heading select knob (hPFPullHeadingSelectKnob).

Next, the pilots perform a read back procedure (aConfirmTheChange). They do this by having the PM point at the heading window value (which can be observed by the PF) and the PM performs the procedure for repeating back the heading that the PM heard from the ATCo to the PF and the ATCo.

If the heading in the window does not match the heading the PF just heard read back to the ATCo, the pilots must collaborate to update the heading (aUpdateTheHeading). To do this, the PF enters the heading he or she heard read back to the ATCo by the PM (aCorrectThePF). If this heading in the window still does not match the heading the PM heard from the ATCo, the PM again points at the heading window and repeats the heading he or she heard from the ATCo back to the PF (all under aCorrectThePF). This entire process repeats as long as the heading window does not match the heading the PM heard from the ATCo.

If the clearance the ATCo heard read back from the pilots does not match what the ATCo intended, aCommAndConfirm must repeat until this is true. Once this is true, the PF engages the entered heading.

13.3.4 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 is *Executing*.

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

- *Ready*: the initial (inactive) state which indicates that the activity or action is waiting to execute;
- *Executing*: the active state which indicates that the activity or action is executing; and
- *Done*: the secondary (inactive) state which indicates that the activity has finished executing.

While *Precondition*s, *RepeatCondition*s, and *CompletionCondition*s 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, the execution state of its parent, children (activities or actions into which the activity decomposes), and siblings (activities or actions contained within the same decomposition).

- *StartCondition*: implicit condition that triggers the start of an activity or action defined in terms of the execution states of its parent and siblings.
- *EndCondition*: implicit condition to end the execution of an activity or action defined in terms of the execution state of its children.
- *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 comprised of two conjuncts with 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, generally formulated as:

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

This is formulated differently depending on the 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 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 *StartCondition*. Top level activities that are defined for sharedeofms treat all other activities as if they are in an and_par relationship, thus they have *StartCondition*s that are always *true*.

An *EndCondition* is also comprised of two conjuncts both related to an activity's children. Since an action has no children, an action's *EndCondition* defaults to *true*. The first conjunct asserts that the execution states of the activity's children satisfy the requirements stipulated by the activity's decomposition operator. The second asserts that none of the children 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, sync, or com decomposition operator; and $\vee$ 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 decompo-

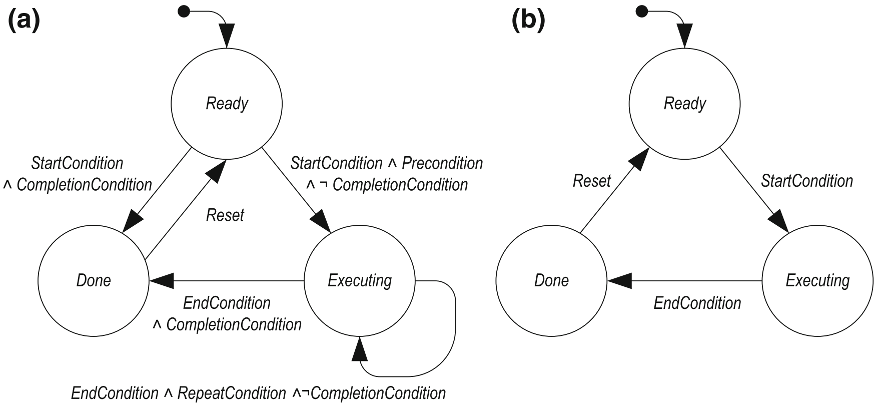


Fig. 13.8 **a** Execution state transition diagram for a generic activity. **b** Execution state transition diagram for a generic action

sition operator, the first conjunct asserts that the last activity or action in the decomposition 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 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 execution states. This is presented in Fig. 13.8 where states are represented as nodes (rounded rectangles) and transitions as arcs. Guards are attached to each arc.

The transitions for an activity (Fig. 13.8a) are described in more detail below:

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

Note that if an activity does not have a *Precondition*, the *Precondition* condition is considered to be *true*. If the activity does not have a *CompletionCondition*, the *CompletionCondition* clause is removed from all possible transitions and the *Ready* to *Done* transition is completely removed (Fig. 13.8a). If the activity does not have a *RepeatCondition*, the *Executing* to *Executing* transition is removed (Fig. 13.8a).

The transition criteria for an action is simpler (Fig. 13.8b) since an action cannot have a *Precondition*, *CompletionCondition*, or *RepeatCondition*. Because actions do not have any children, their *EndCondition*s are always *true*. Actions in a sync or com decomposition must transition through their execution states at the same time.

The behavior of human action outputs and local variable assignments are dependent on the action formal semantics. For a humanaction with autoreset behavior, the human action output occurs when a corresponding action node in the EOFM task structure is *Executing* and does not otherwise. For a humanaction with toggle behavior, the human action switches between occurring or not occurring when a corresponding action node is *Executing*. A humanaction with a setvalue behavior will set the corresponding human action's value when the action is *Executing*. Similarly, a communication action or a local variable assignment associated with an action node will occur when the action is *Executing*.

13.3.5 EOFM to SAL Translation

To be utilized in a model checking verification, models written using EOFM's XML notation must be translated into a model checking language. We use the formal semantics to translate XML into the language of the Symbolic Analysis Laboratory (SAL). SAL was selected for use with EOFM because of the expressiveness of its notation, its support for a suite of checking and auxiliary tools, and its cutting edge performance at the time of EOFM's development. SAL is a framework for combining different tools to calculate properties of concurrent systems (De Moura et al. 2003; Shankar 2000). The SAL language (see De Moura et al. 2003) 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 SAL's mathematical operators. The transition assignment defines how the value of local and output variables change when the guard is satisfied. The SAL language is supported by a tool suite which includes state of the art symbolic (SAL-SMC), bounded (SAL-BMC), and "infinite" bounded (SAL-INF-BMC) model checkers. Auxiliary tools include a simulator, deadlock checker, and an automated test case generator.

The EOFM to SAL translation is automated by custom Java software that uses the Document Object Model (Le Hégaret 2002) to parse EOFM's XML code and convert it into SAL code. Currently, several different varieties of EOFM to SAL translators exist; each supports different subsets of EOFM functionality. For example, there is a different translator for EOFMC that allows for the modeling and analyses of human-human communication and coordination. Despite the difference, the translators generally function on the same principles.

For a given instantiated EOFM, the translator defines SAL constants and types using the constant and userdefinedtype nodes. The translator creates a single SAL module for the group of human operators represented in humanoperator and sharedeofm nodes. Input, local, and output variables are defined in each module corresponding to the humanoperator nodes' inputvariable, localvariable, and humanaction nodes respectively. Input and local variables are defined in SAL using the name and type (basictype or userdefinedtype) attributes from the XML. Local variables are initialized to their values from the markup. All output variables represent humanactions. They are either treated as Boolean (for actions with autoreset and toggle behavior; *true* indicates this action is being performed) or are given the type associated with the value they carry (for setvalue behavior).

The translator defines two Boolean variables in the group module to handle a coordination handshake with the human-device interface module (see Bolton and Bass 2010a):

1. An input variable *InterfaceReady* that is *true* when the interface is ready to receive input; and
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* (Fig. 13.8). As described previously, the activity and action state transactions define the individual and coordinated tasks of the group of human operators (Fig. 13.8). Each activity and action in a task structure has an associated local variable of type *ActivityState*. The transitions between the execution state for each activity and action are explicitly defined as nondeterministic transitions in the module representing human task behavior. Figure 13.9 presents patterns that show how the transitions from Fig. 13.8a for each activity are implemented in SAL by the translator. Note that for a given activity, the *StartCondition* and *EndCondition* are defined in accordance with the formal semantics (see Sect. 13.3.4). It is also important to note that, in these transitions, if an activity does not have a particular strategic knowledge condition, the clause in the transition guard is eliminated. Further, if an activity does not have a repeat condition, the *Executing* to *Executing* condition is eliminated completely. If an activity does not have a *CompletionCondition*, the *Ready* to *Done* transition is eliminated. 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 children's execution state variables (and all activities and actions lower in the hierarchy) are assigned the *Ready* state (see the *Executing* to *Executing* transition in Fig. 13.9a). Additionally, for each activity at the top of a task hierarchy, a transition (Fig. 13.9b) 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.

(a)

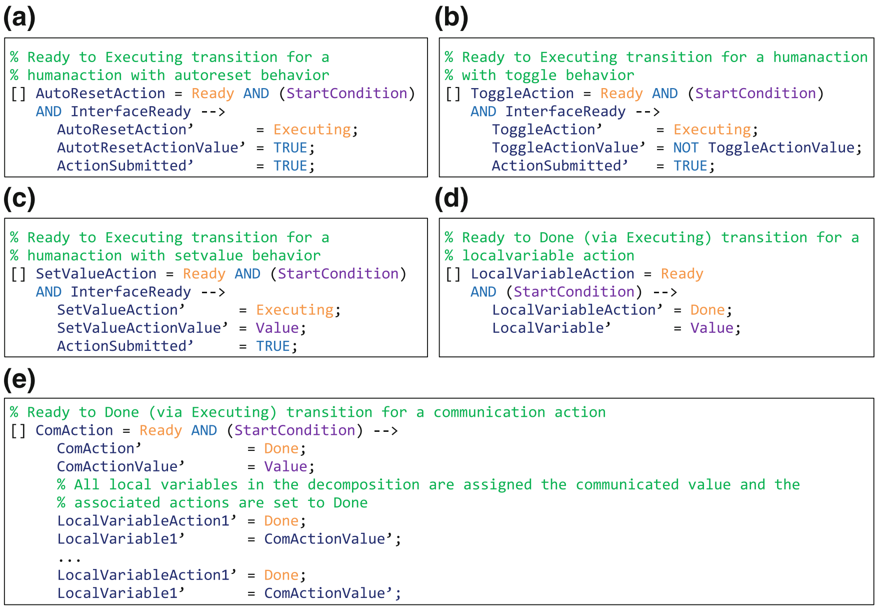
```
% Ready to Executing transition for an activity
[] Activity = Ready AND (StartCondition) AND (Precondition) AND NOT (CompletionCondition) -->
      Activity’ = Executing;
% Ready to Done transition for an activity
[] Activity = Ready AND (StartCondition) AND (CompletionCondition) -->
      Activity’ = Done;
% Executing to Executing (a repeat) transition for an activity
[] Activity = Executing AND (EndCondition) AND (RepeatCondition) AND NOT (CompletionCondition) -->
      Activity’ = Executing;
      % All sub-activities and actions are set to Ready (they are Reset)
      SubAct’   = Ready;
      ...
% Executing to Done transition for an activity
[] Activity = Executing AND (EndCondition) AND (CompletionCondition) -->
      Activity’ = Done;
```

(b)

```
% Done to Ready transition (Reset) for an activity with no parent
[] Activity = Done -->
      Activity’ = Ready;
      % All sub-activities and actions are set to Ready (the are Reset)
      SubAct’   = Ready;
      ...
```

Fig. 13.9 Patterns of SAL transitions used for an activity to transition between its execution state. **a** Represents all but the Reset transition. **b** Represents the Reset transition. Note that this Reset transition only occurs for activities at the top of a task model hierarchy (ones with no parent). Other resets occur based on the assignments that occur in these types of transitions or in repeat transitions (see a). In SAL notation (see De Moura et al. 2003), `[]` indicates the beginning of a nondeterministic transition. This is followed by a Boolean expression representing the transition’s guard. The guard ends with a `->`. This is followed underneath by a series of variable assignments where a `'` on the end of a variable indicates that the variable is being used in the next state. Color is used to improve code readability. Comments are *green*, variables are *dark blue*, reserved SAL words are *light blue*, and values are *orange*. *Purple* is used to represent expressions, such activity conditions, or values that would be explicitly inserted into the code by the translator

Transitions between execution states for variables associated with action nodes that represent humanactions are handled differently due to the variable action types and behaviors, simpler formal semantics, and the coordination handshake. Because resets are handled by activity transitions, explicit transitions are only required for activity *Ready* to *Executing* and *Executing* to *Done* transitions. Figure 13.10 shows how the *Ready* to *Executing* transitions are represented for each of the different action types. Note that only actions that are humanactions must wait for *InterfaceReady* to be *true* to execute. This is because these are the only actions that produce behavior that needs to be observed by other elements in the formal model (Fig. 13.10a–c). In the variable assignment for each of the actions, *ActionsSubmitted* is set to true. Similarly, to improve model scalability, other types of human actions (local variable assignments and communication actions; Fig. 13.10d, e) do not make use of the coordination protocol. Their *Executing* to *Done* transitions also occur automatically following the *Ready* to *Executing* because an action’s *EndCondition* is always *true*. Thus, the transitions shown in Fig. 13.10 effectively show a *Ready* to *Done* transition with all of the work associated with the action *Executing* occurring. Actions that fall in a sync decomposition are handled in accordance with the transitions in Fig. 13.10. Specifically, the *StartCondition* of the first action in the decomposition



(a)

```
% Ready to Executing transition for a
% humanaction with autoreset behavior
[] AutoResetAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      AutoResetAction’        = Executing;
      AutotResetActionValue’  = TRUE;
      ActionSubmitted’        = TRUE;
```

(b)

```
% Ready to Executing transition for a humanaction
% with toggle behavior
[] ToggleAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      ToggleAction’       = Executing;
      ToggleActionValue’  = NOT ToggleActionValue;
      ActionSubmitted’    = TRUE;
```

(c)

```
% Ready to Executing transition for a
% humanaction with setvalue behavior
[] SetValueAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      SetValueAction’      = Executing;
      SetValueActionValue’ = Value;
      ActionSubmitted’     = TRUE;
```

(d)

```
% Ready to Done (via Executing) transition for a
% localvariable action
[] LocalVariableAction = Ready
   AND (StartCondition) -->
      LocalVariableAction’ = Done;
      LocalVariable’       = Value;
```

(e)

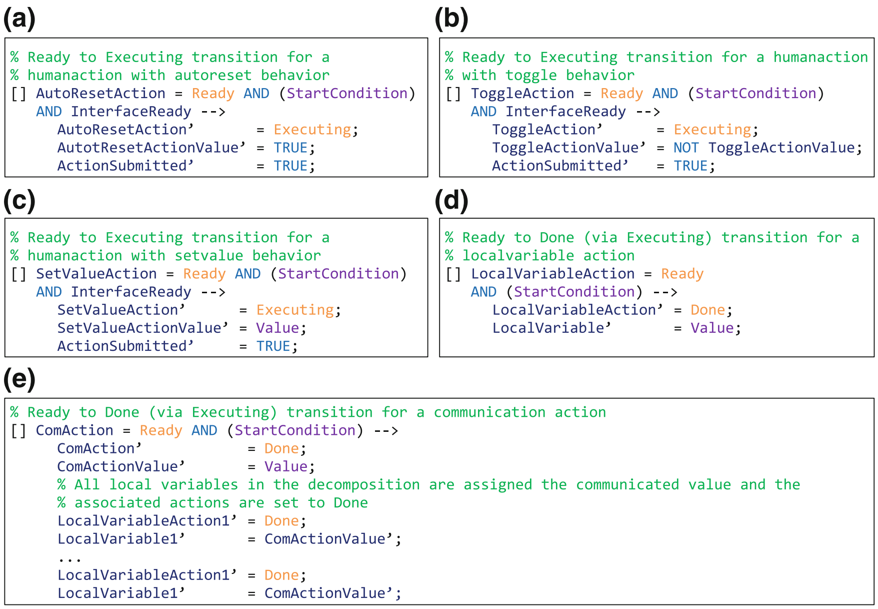
```
% Ready to Done (via Executing) transition for a communication action
[] ComAction = Ready AND (StartCondition) -->
      ComAction’             = Done;
      ComActionValue’        = Value;
      % All local variables in the decomposition are assigned the communicated value and the
      % associated actions are set to Done
      LocalVariableAction1’ = Done;
      LocalVariable1’        = ComActionValue’;
      ...
      LocalVariableAction1’ = Done;
      LocalVariable1’        = ComActionValue’;
```

Fig. 13.10 Patterns of SAL transitions used for an action to transition from *Ready* to *Executing* and/or *Ready* to *Done* with an implied execution in between. **a** The pattern used for a humanaction with autoreset behavior. **b** The pattern used for a humanaction with toggle behavior. **c** The pattern used for a humanaction with setvalue behavior. **d** The pattern used for a localvariable action. **e** The pattern used for a communication action. Note that `LocalVariable1–LocalVariableN` represent local variables in a com decomposition that received the value communicated. Further note that in **c**, **d**, and **e**, `Value` is used to represent a variable or specific value from the XML markup

must be satisfied in the transition guard and the variable assignment next state values for all actions in the decomposition are done in accordance with their type.

Because the *EndCondition* for all actions is always true, the *Executing* to *Done* transition for humanactions is handled by a single guard and transition assignment (Fig. 13.11). In this, the guard accounts for the handshake protocol. Thus the guard specifies that *ActionsSubmitted* is *true* and that *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 *Executing* humanaction is set to *Done* (it is unchanged otherwise); and any humanaction output variables that supports the autoreset behavior are set to *false*.

Because all of the transitions are non-deterministic, multiple activities can be executed independently of each other when _par decomposition operators are used, when they are in non-shared task structures of different human operators, and when they are in *sharedeofms*. Multiple human actions resulting from such relationships 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*. However, human actions need not wait for all



(a)

```
% Ready to Executing transition for a
% humanaction with autoreset behavior
[] AutoResetAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      AutoResetAction’        = Executing;
      AutotResetActionValue’  = TRUE;
      ActionSubmitted’        = TRUE;
```

(b)

```
% Ready to Executing transition for a humanaction
% with toggle behavior
[] ToggleAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      ToggleAction’      = Executing;
      ToggleActionValue’ = NOT ToggleActionValue;
      ActionSubmitted’   = TRUE;
```

(c)

```
% Ready to Executing transition for a
% humanaction with setvalue behavior
[] SetValueAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      SetValueAction’      = Executing;
      SetValueActionValue’ = Value;
      ActionSubmitted’     = TRUE;
```

(d)

```
% Ready to Done (via Executing) transition for a
% localvariable action
[] LocalVariableAction = Ready
   AND (StartCondition) -->
      LocalVariableAction’ = Done;
      LocalVariable’       = Value;
```

(e)

```
% Ready to Done (via Executing) transition for a communication action
[] ComAction = Ready AND (StartCondition) -->
      ComAction’             = Done;
      ComActionValue’        = Value;
      % All local variables in the decomposition are assigned the communicated value and the
      % associated actions are set to Done
      LocalVariableAction1’ = Done;
      LocalVariable1’       = ComActionValue’;
      ...
      LocalVariableAction1’ = Done;
      LocalVariable1’       = ComActionValue’;
```

Fig. 13.11 SAL transition pattern used to handle action *Executing* to *Done* transitions

possible actions to occur once *InterfaceReady* has become *true*. In this way, all possible overlapping and interleaving of parallel actions can be considered.

For our communication protocol case study, the EOFMC XML (Fig. 13.7) was translated into SAL using the automated translator.[5] The original model contained 164 lines of XML code. The translated model contained 490 lines of SAL code. A full listing of the model code can be found at http://tinyurl.com/EOFMCBook.[6] This model was asynchronously composed with a simple model representing the heading change window, where the heading can be changed when the pilot rotates the heading knob. These two models were composed together to create the full system model used in the verification analyses.

13.3.6 Erroneous Behavior Generation

EOFM supports three different types of erroneous behavior generation, each based on different theory and supported by different translators. For each of these generation techniques, an analyst can specify a maximum number of erroneous behaviors

[5] This translation also included miscommunication generation. This is discussed subsequently in Sect. 13.3.6.3.

[6] Note that variables presented in the text are slightly different than in the model to improve readability.

to consider. The associated translator then creates a formal model that will generate a formal representation of the EOFM represented behavior with the ability to perform erroneous behaviors based on the theory being used. When paired with a larger formal model and evaluated with model checking, the model checker will verify that the specification properties are either true or not with up to the maximum number of erroneous behaviors occurring, in all of the possible ways the erroneous behavior can occur up to the maximum. Thus, this feature allows consideration of the impact that potentially unanticipated erroneous behavior can have on system safety.

13.3.6.1 Phenomenological Erroneous Behavior Generation

The first erroneous behavior generation technique is only supported by non-EOFMC versions of EOFM. In this generation technique (introduced in Bolton and Bass 2010b and further developed in Bolton et al. 2012), each action in an instantiated EOFM task is replaced with a generative task structure that allows for the performance of Hollnagel's (1993) zero-order phenotypes of erroneous action (Fig. 13.12). Specifically, when it is time to execute an action, the new model allows for the

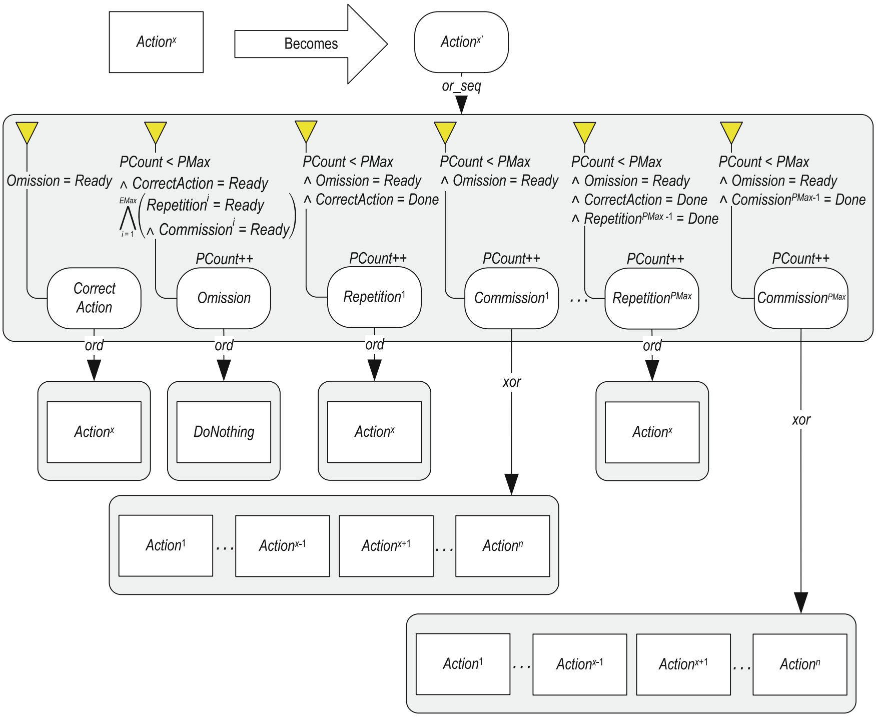


Fig. 13.12 EOFM task pattern that replaces actions at the bottom of EOFM task hierarchies to generate zero-order phenotypes of erroneous action

performance of the original action, the omission of that action, the repetition of the original action, and the commission of another action. Multiple erroneous repetitions and commissions can occur after either the performance of the original action and/or other repetitions or commissions. It is through multiple performances of erroneous actions that more complicated erroneous behaviors (more complex phenotypes) are possible.

Whenever an erroneous activity executes, a counter (*PCount*) increments. Preconditions on each activity keep the number of erroneous acts from exceeding the analyst specified maximum (*PMax*). The preconditions also ensure that there is only one way to produce a given sequence of actions for each instance of the structure. For example, an omission can only execute if no other activity in the structure has executed (all other activities must be *Ready*) and every other activity can only execute if there has not been an omission. See Fig. 13.12 and Bolton et al. (2012) for more details.

Note that analysts wishing to use this or the following erroneous behavior generation features will likely want to do so iteratively. That is, start with a maximum number of erroneous behaviors of 0 and incrementally increase it while checking desired properties at each iteration. This approach helps analysts keep model complexity under control while allowing them to evaluate the robustness of a system until they find a failure or are satisfied with the performance of their model.

13.3.6.2 Attentional Failure Generation

The second erroneous behavior generation technique (Bolton and Bass 2011, 2013) is supported by translators for both the original EOFM and EOFMC. Rather than generate erroneous behavior from the action level up, this second approach attempts to replicate the physical manifestation of Reason's slips (1990). Specifically, humans may have failures of attention that can result in them erroneously omitting, repeating, or committing an activity. We can replicate this behavior by changing the way the translator interprets the EOFM formal semantics. Specifically, all of the original transitions from the original formal semantics are retained (Fig. 13.8). However, additional, erroneous transitions are also included (Fig. 13.13) to replicate a human operator not properly attending to the environmental conditions described in EOFM strategic knowledge. A human operator can fail to properly attend to when an activity should be completed and perform an omission (an erroneous *Ready* to *Done* or *Executing* to *Done* transition), can fail to properly attend to an activity's *Precondition* and perform an erroneous *Ready* to *Executing* transition (a commission), or not properly attend to when an activity can repeat and perform a repetition (an erroneous *Executing* to *Executing* transition). Whenever an erroneous transition occurs, a counter (*ACount*) is incremented. An erroneous transition is only allowed to occur if the counter has not reached a maximum (*ACount* < *AMax*). More information on this erroneous behavior generation technique can be found in Bolton and Bass (2013).

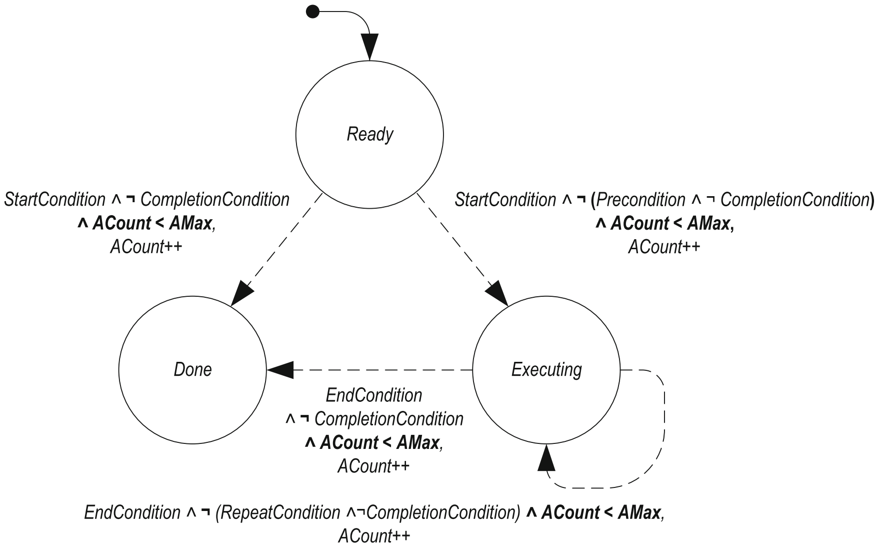

Ready
StartCondition ∧ ¬ CompletionCondition
∧ ACount < AMax,
ACount++
StartCondition ∧ ¬ (Precondition ∧ ¬ CompletionCondition)
∧ ACount < AMax,
ACount++
Done
Executing
EndCondition
∧ ¬ CompletionCondition
∧ ACount < AMax,
ACount++
EndCondition ∧ ¬ (RepeatCondition ∧¬CompletionCondition) ∧ ACount < AMax,
ACount++

Fig. 13.13 Additional transitions, beyond those shown in Fig. 13.8a, used to generate erroneous behaviors caused by a human failing to attend to information in EOFM strategic knowledge conditions

13.3.6.3 Miscommunication Generation

The final erroneous behavior generation is only supported by EOFMC translation and relates to the generation of miscommunications in communication events (Bolton 2015). This generation approach works similarly to attentional failure generation in that it adds additional formal semantics representing erroneous behaviors. Thus, miscommunication generation keeps all of the formal semantics described in Sect. 13.3.4. However, for each transition representing a human-human communication action, an additional transition is created (Fig. 13.14). In this, the value communicated can either assume the correct value meant for the original, non-erroneous, transition or any other possible communicable value as determined by the type of the information being communicated. Similarly, the local variables that are assigned the communicated value received by the other human operators can be assigned the value of what was actually communicated or some other possible communicable value. In this way, our method is able to model miscommunications as an incorrect message being sent, an incorrect message being received, or both. As with the other erroneous behavior generation methods, this approach is regulated by a counter (*CCounter*) and an analyst specified maximum (*CMax*). Every time a miscommunication occurs, a counter is incremented; and an erroneous transition can only ever occur if the counter is less than the maximum. More information on this process can be found in Bolton (2015).

```
% Ready to Done (via Executing) transition for a communication action with
% miscommunication generation
[] ComAction = Ready AND (StartCondition) AND CCount < CMax -->
      ComAction'            =  Done;
      ComActionValue'       IN {x: ComType | TRUE};
      LocalVariableAction1' =  Done;
      LocalVariable1'       IN {x: ComType | TRUE};
      ...
      LocalVariableAction1' =  Done;
      LocalVariable1'       IN {x: ComType | TRUE};
      LocalVariableAction1' =  Done;
      CCount'               =  IF    ComActionValue' /= Value
                                  OR LocalVariable1' /= Value
                                  OR ...
                                  OR LocalVariableN' /= Value
                               THEN Count + 1
                               ELSE Count ENDIF;
```

Fig. 13.14 SAL transition pattern used for generating miscommunications. In this, `ComType` is meant to represent the data type of the particular communication being performed. The statement `IN x: ComType | TRUE` is thus saying that the variable to the left of the expression can be assigned any value from the type `ComType` nondeterministically. Note that `Value` here represents the value that should have been communicated. Since `Value` is in `ComType`, the nondeterministic assignments can produce a correct communication. For this reason, the `IF...THEN...ELSE...ENDIF` expression checks to ensure that a miscommunication actually occurred. If it did, the `CCount` is incremented. Otherwise, it remains the same

In our communication protocol application, miscommunication generation was used when the original EOFMC XML code (Fig. 13.7) was converted into SAL's input language with our translator. Doing this, three versions of the model were created, starting with *CMax* values of 0, 1, and 2.

13.3.7 Specification and Verification

EOFM-supported verification analyses are designed to be conducted with SAL's symbolic model checker. Because of this, specification properties are formulated in linear temporal logic.

In the air traffic application, the purpose of the communication protocol is to ensure that the pilots set the aircraft to the heading intended by the air traffic controller. Thus, we formulate this in linear temporal logic as follows:

$$\mathsf{G}\left(\begin{array}{l}(aChangeHeading = Done)\\ \rightarrow (iHeadingWindowHeading = lATCSelectedClearance)\end{array}\right) \qquad (13.1)$$

This specification was checked against the formal model with varying levels of *CMax* using SAL's symbolic model checker (sal-smc) on a windows laptop with an Intel Core i7-3667U running SAL on cygwin.

Verification analysis results are show in Table 13.2.[7]

[7] Note that an additional verification was conducted using the specification $\mathsf{F}(aChangeHeading = Done)$ for every version of the model to ensure that (13.1) was not true due to vacuity.

Table 13.2 Communication protocol formal verification results

Model with *CMax*	Verification data		
	Outcome	Time (s)	Visited states
0	✓	3.588	6498
1	✓	4.461	77177
2	×	4.602	415537

Note: A ✓ indicates verification of (13.1) was successful. A × indicates verification failed and produced a counterexample. Because verification times occur via symbolic model checking, we do not expect the verification times to vary linearly with respect the number of visited states

These results show that for maximums of zero and one miscommunications, the presented protocol will always be capable of allowing air traffic control to communicate headings to the pilot successfully and have the pilots execute that heading as long as the protocol is adhered to. However, a failure occurs when there are up to two miscommunications ($CMax = 2$).

13.3.8 Counterexample Visualization

Any produced counterexamples can be visualized and evaluated using EOFM's visual notation (Bolton and Bass 2010c). In a visualized counterexample, each counterexample step is drawn on a separate page of a document. Any task with *Executing* activities or actions is drawn next to a listing of other model variables and their values at the given step. Any drawn task hierarchy will have the color of its activities and actions coded to represent their execution state at that step. Other listed model variables can be categorized based on the model concept they represent: human mission, human-automation interface, automation, environment, and other. Any changes in an activity or action execution state or variable values from previous steps are highlighted. More information on the visualizer can be found in Bolton and Bass (2010c).

When counterexample visualization was applied to the counterexample produced for the communication protocol application, it revealed the following failure sequence:

1. When the ATCo first communicates the heading to the pilots (under aComHeading), a miscommunication occurs and the pilots both hear different incorrect headings.
2. Then when the PM reads the heading back the ATCo and the PF (under aReadbackHeading), a second miscommunication occurs and both the ATCo and the PF think they heard the original intended heading and the heading originally heard from the ATCo respectively.
3. As a result of this, the procedure proceeds without any human noticing an incorrect heading, and the incorrect heading is engaged.

A full listing of the counterexample visualization can be found at http://tinyurl.com/EOFMCBook.

13.4 Discussion

The presented application demonstrates the power of EOFM to find complex problems in systems that rely on human behavior for their safe execution, both with normative and generated erroneous human behavior. From the perspective of the application domain, the presented results are encouraging: as long as the protocol in Fig. 13.6 is adhered to, heading clearances will be successfully communicated and engaged. Ultimately it must be up to designers to determine how robust a protocol must be to miscommunication to support system safety. If further reliability is required, analysts might want to use additional pilot and ATCo read backs. Additional analyses that explore how further read backs can increase reliability can be found in Bolton (2015). Further, analysts may wish to conduct additional analyses to see how robust to other types of erroneous human behaviors the protocol is both with and without miscommunication. An example of how this can be done can be found in Pan and Bolton (2015).

Beyond this case study, EOFM has demonstrated its use in the analysis of a number of applications in a variety of domains. It is also being continually developed to improve its analyses and expand the ways it can be used in the design and evaluation of human-machine systems. These are discussed below.

13.4.1 Applications

EOFM and EOFMC have been used to evaluate an expanding set of applications. These are described below. While these have predominantly been undertaken by the authors and their collaborators, the EOFM toolset is freely available (see http://fhsl.eng.buffalo.edu/EOFM/). We encourage others to make use of these tools as EOFM development continues.

13.4.1.1 Aviation Checklist Design

Pilot noncompliance with checklists has been associated with aviation accidents. For example on May 31, 2014, procedural noncompliance of the flight crew of a Gulfstream Aerospace Corporation G-IV contributed to the death of the two pilots, a flight attendant, and four passengers (NTSB 2015). This noncompliance can be influenced by complex interactions among the checklist, pilot behavior, aircraft automation, device interfaces, and policy, all within the dynamic flight environment. We used EOFM to model a checklist procedure and employed model checking to evaluate checklist-guided pilot behavior while considering such interactions (Bolton and Bass 2012). Although pilots generally follow checklist items in order, they can complete them out of sequence. To model both of these conditions, two task models were created: one where the pilot will always perform the task in order (enforced by an ord decomposition) and one where he or she can perform them in any order (using the and_par decomposition operator).

Spoilers are retractable plates on the wings that, when deployed, slow the aircraft and decrease lift. If spoilers are not used, the aircraft can overrun the runway (NTSB 2001). A pilot can arm the spoilers for automatic deployment using a lever. Alternatively, a pilot can manually deploy the spoilers after touchdown. Arming the spoilers before the landing gear has been lowered or the landing gear doors have fully opened can result in automatic premature deployment which can cause the aircraft to lose lift and have a hard landing. Spoiler deployment is part of the *Before Landing* checklist. In our analyses, for the system model using the human task behavior (that is, the *Before Landing* checklist) with the and_par decomposition operator, we identified that a pilot could arm the spoilers early and then open the landing gear doors, a situation that could cause premature spoiler deployment. We explored how different design interventions could impact the safe arming and deployment of spoilers.

13.4.1.2 Medical Device Design

EOFM has been used to evaluate the human-automation interaction (HAI) of two different safety-critical medical devices. The first was a radiation therapy machine (Bogdanich 2010; Leveson and Turner 1993) based on the Therac-25. This device was a room-sized, computer-controlled, medical linear accelerator. It had two treatment modes: electron beam mode is used for shallow tissue treatment, and x-ray mode is used for deeper treatments—requiring electron beam current approximately 100 times greater than that used for the other mode. The x-ray mode used a beam spreader (not used in electron beam mode) to produce a uniform treatment area and attenuate the radiation of the beam. An x-ray beam treatment application without the spreader in place could deliver a lethal dose of radiation. We used EOFM with its phenomenological erroneous behavior generation to evaluate how normative and/or unanticipated erroneous behavior could result in the administration of an unshielded x-ray treatment (Bolton et al. 2012). We discovered that this could occur if a human operated accidentally selected x-ray mode (a generated erroneous act) and corrected it and administered treatment too quickly.

In the second medical device application, we used verifications with both normative behavior (Bolton and Bass 2010a) and attentional failure generation (Bolton and Bass 2013) to evaluate the safety of a patient controlled analgesia (PCA) pump. A PCA pump is a medical device that allows a patient to exert some control over intravenously delivered pain medication, where medication is delivered in accordance with a prescription programmed into the device by a medical practitioner. Using EOFM with its attentional failure generation, we were able to both discover when an omission caused by an attentional slip could result in an incorrect prescription being administered and how a simple modification to the device's interface could prevent this failure from occurring.

13.4.1.3 Medical Device User Manual Design

The EOFM language, the ability to create new models in EOFM, and the ease of composing formal task analytic models in SAL into larger system models helped to provide insights relevant to patient user manual evaluation. User manual designers generally use written procedures, figures and illustrations to convey procedural and device configuration information. However ensuring that the instructions are accurate and unambiguous is difficult. To analyze a user manual, our approach integrated EOFM's formal task analytic models and device models with safety specifications via a computational framework similar to those used for checklists and procedures. We demonstrated the value of this approach using alarm troubleshooting instructions from the patient user manual of a left ventricular assist device (LVAD) (Abbate et al. 2016).

During the process of encoding the written instructions into the formal EOFM task model, we discovered problems with task descriptions in the manual as statements were open to interpretation. During the process of developing the XML description of the procedure, for example, it became clear that the description in the user manual did not define which end of a battery cable to disconnect. We also identified issues with the order of troubleshooting steps in that the procedure included steps that did not fix the problem before ones that did. The ability to change a single decomposition operator in the model (from ord to to _seq) and subsequent model translation allowed model checking analyses visualized with our counter example visualizer to show that a better ordering was possible. In addition the ability to include a formal device model with the formal task model highlighted that the instructions did not consider all possible initial device conditions.

13.4.1.4 Human-Human Communication and Coordination Protocols

EOFMC has been used to evaluate the robustness of protocols humans use to communicate information and coordinate their collaborative efforts. In one set of such analyses, we evaluated the protocols air traffic controllers use to communicate clearances to pilots. These analyses modeled the protocols in EOFMC and were formally verified to determine if incorrect clearances could be engaged. Analyses without any miscommunication generation (Bass et al. 2011) did not discover any problems. Miscommunication generation was used to identify that a maximum of one miscommunication could cause the protocol to fail and show how this protocol could be modified to make it robust to higher maximum numbers of miscommunications (Bolton 2015). A variant of this protocol is used for the case study analyses presented in the next section.

A variant of this analysis was used to evaluate the protocol that a team of engineers use to diagnose alarms in a nuclear power plant (Pan and Bolton 2015, 2016). Rather than simply look for binary failure conditions in the performance of the team, this work compared different protocols based on guaranteed performance levels determined by the degree of correspondence between team member conclusions and system knowledge at the end of the protocol. This was used to evaluate two different versions of the protocol: one where confirmation or contradiction statements were used to show agreement or disagreement respectively and one where read backs were used. In both cases, miscommunication and attentional failure generation were included in formal verifications. These analyses revealed that the read back procedure outperformed the other, producing correct operator conclusions for any number of miscommunications and up to one attentional slip.

13.4.2 EOFM Extensions

Three EOFM extensions beyond its standard methods (Fig. 13.2) are described next.

13.4.2.1 Scalability Improvements

As the size of the EOFM used in formal verification analyses increases, the associated formal model size and verification times increase exponentially (Bolton and Bass 2010a; Bolton et al. 2012). This can limit what system EOFM can be used to evaluate. Because of this, efforts have attempted to improve its scalability. To accomplish this, the EOFM formal semantics (see Sect. 13.3.4) are interpreted slightly differently. In this new interpretation, the formal representation of the execution state are “flattened” so that they are defined as Boolean expressions purely in terms of the execution state of actions at the bottom of the EOFM hierarchy. This allows the transitions associated with activities to be represented in the transition logic of the actions. As such, there are fewer intermediary transitions in the formal EOFM representation. This has resulted in significant reductions in model size and verification time without the loss of expressive power for both artificial benchmarks and realistic applications pulled from the current EOFM literature (Bolton et al. 2016). For example, a PCA pump model that contained seven different tasks and the associated interface and mission components of the PCA pump in the larger formal model originally had 4,072,083 states and took 90.4 s to verify a true property. With the new translator, the model statespace size was reduced to 15,388 states and took only 5.6 s to verify the same property (Bolton et al. 2016).

13.4.2.2 Specification Property Generation

While EOFM can be used to evaluate HAI with model checking, most work requires analysts to manually formulate the properties to check, a process that may be error-prone. Further, analysts may not know what properties to check and thus fail to specify properties that could find unanticipated HAI issues. As such, unexpected dangerous interaction may not be discovered even when formal verification is used.

To address this, an extension of EOFM's method (Fig. 13.2) includes automatically generating specification properties from task models (Bolton 2011, 2013; Bolton et al. 2014). In general, these approaches work by using modified EOFM to SAL translators to automatically generate specification properties from the EOFM task models.

Two types of specification generation have been developed. The first types of generated specifications are task-based usability properties (Bolton 2011, 2013). These allow analysts to automatically check that a human-automation interface will support the fulfillment of the goals from EOFM tasks. While initial versions of this process only allowed the analyst to find problems (Bolton 2011), later extensions provided a diagnostic algorithm that enabled the reason for the usability failures to be systematically identified (Bolton 2013). This method was ultimately used to evaluate the usability of the PCA pump application discussed previously.

The second class of generated properties asserted qualities about the execution state of the task models based on their formal semantics (Bolton et al. 2014). This enabled analysts to look for problems in the HAI of a system by finding places where the task models would not perform as expected and thus elucidate potential unanticipated interaction problems. For example, properties would assert that every execution state of each activity and action were reachable, that every transition between execution states was achievable, that all activities and actions would eventually finish, and that any human task would always eventually be performable. This method was used to re-evaluate the previously discussed aircraft before landing checklist procedure, where it discovered an unanticipated issue (Bolton et al. 2014). It also was used in the evaluation of an unmanned aerial vehicle control system (van Paassen et al. 2014). These last two applications are particularly illustrative of the specification property generation feature's power because both discovered system problems previously unanticipated by the analysts.

13.4.2.3 Interface Generation

User-centered design is an approach for creating human-machine interfaces so that they support human operator tasks. While useful, user-centered design can be challenging because designers can fail to account for human-machine interactions that occur due to the inherent concurrence between the human and the other elements of the system. This extension of EOFM has attempted to better support user-centered designed by automatically generating formal designs of human-machine interface functional behavior from EOFM task models guaranteed to always support the task. To accomplish this, the method uses a variant of the L* algorithm (Angluin 1987) to learn a minimal finite state automation description of an interface's behavior; it uses a model checker and the formal representation of EOFM task behavior to answer questions about the interface's behavior (Li et al. 2015). This method has been used to successfully generate a number of interfaces including light switches, a vending machine, and a pod-based coffee machine (Li et al. 2015). All generated interfaces have been verified to support the various task-based properties discussed in the previous section and Bolton et al. (2014). Future work is investigating how to include usability properties in the generation process.

13.4.2.4 Improve Tool Interoperability and Usability

The use of XML allows EOFM-supported analyses to be platform independent. Despite this, to date, EOFM has only been used with the model checkers found in SAL. Future work should investigate how to make EOFM compatible with other analysis environments.

Additionally, using the XML to create task models is relatively easy for someone who is familiar with the language and had access to sophisticated XML development environments that support syntax checking and code completion. However, the usability of EOFM could be significantly improved with the addition of visual modeling tools that would allow model creation with a simple point and click interface. This will be investigated in future work.

13.5 Conclusions

As the above narrative demonstrates, EOFM offers analysts a generic, flexible task modeling system that supports a number of different formal analyses. This is evidenced by the different applications that have been evaluated using EOFM and the different analyses both supported by and extending the method (Fig. 13.2). It is important to note that although all of the analyses discussed here use the model checking tools in SAL, this need not be the case. Given that EOFM is XML-based, it should be adaptable to many other formal verification analysis environments. This is further supported by the fact that SAL's input notation is very similar to those offered by other environments. Future work will investigate how EOFM could be adapted to other model checkers to increase the scope of its analyses.

References

Abbate AJ, Throckmorton AL, Bass EJ (2016) A formal task analytic approach to medical device alarm troubleshooting instructions. IEEE Trans Hum-Mach Syst 46(1):53–65

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 Transf 8(6):547–563

Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106

Basnyat S, Palanque P, Schupp B, Wright P (2007) Formal socio-technical barrier modelling for safety-critical interactive systems design. Saf Sci 45(5):545–565

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

Bogdanich W (2010) The radiation boom: radiation offers new cures, and ways to do harm. New York Times 23:23–27

Bolton ML, Bass EJ (2011) Using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking to evaluate human-automation interaction. In: Proceedings of the IEEE international conference on systems man and cybernetics. IEEE, Piscataway, pp 1788–1794

Bolton ML (2011) Validating human-device interfaces with model checking and temporal logic properties automatically generated from task analytic models. In: Proceedings of the 20th behavior representation in modeling and simulation conference. The BRIMS Society, Sundance, pp 130–137

Bolton ML (2013) Automatic validation and failure diagnosis of human-device interfaces using task analytic models and model checking. Comput Math Organ Theory 19(3):288–312

Bolton ML (2015) Model checking human-human communication protocols using task models and miscommunication generation. J Aerosp Inf Syst 12:476–489

Bolton ML, Bass EJ (2010a) Formally verifying human-automation interaction as part of a system model: limitations and tradeoffs. Innov Syst Softw Eng: A NASA J 6(3):219–231

Bolton ML, Bass EJ (2010b) Using task analytic models and phenotypes of erroneous human behavior to discover system failures using model checking. In: Proceedings of the human factors and ergonomics society annual meeting. HFES, Santa Monica, pp 992–996

Bolton ML, Bass EJ (2010c) 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

Bolton ML, Bass EJ (2012) Using model checking to explore checklist-guided pilot behavior. Int J Aviat Psychol 22:343–366

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

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

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 70:888–906

Bolton ML, Bass EJ, Siminiceanu RI (2013) Using formal verification to evaluate human-automation interaction: a review. IEEE Trans Syst Man Cybern: Syst 43(3):488–503

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

Bolton ML, Zheng X, Molinaro K, Houser A, Li M (2016) Improving the scalability of formal human-automation interaction verification analyses that use task analytic models. Innov Syst Softw Eng: A NASA J (in press). doi:10.1007/s11334-016-0272-z

Clark J, Murata M (2001) Relax NG specification. Committee Specification. http://relaxng.org/spec-20011203.html

De Moura L, Owre S, Shankar N (2003) The SAL language manual. Technical Report CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park. http://staffwww.dcs.shef.ac.uk/people/A.Simons/z2sal/saldocs/SALlanguage.pdf

Degani A, Heymann M, Shafto M (1999) Formal aspects of procedures: the problem of sequential correctness. Proceedings of the 43rd annual meeting of the human factors and ergonomics society. HFES, Santa Monica, pp 1113–1117

Fields RE (2001) Analysis of erroneous actions in the design of critical systems. PhD thesis, University of York, York

Giese M, Mistrzyk T, Pfau A, Szwillus G, von Detten M (2008) AMBOSS: a task modeling approach for safety-critical systems. In: Proceedings of the second international conference on human-centered software engineering. Springer, Berlin, pp 98–109

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 Alamitos, pp 1–11

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

Hollnagel E (1993) The phenotype of erroneous actions. Int J Man-Mach Stud 39(1):1–32

Kirwan B, Ainsworth LK (1992) A guide to task analysis. Taylor and Francis, London

Le Hégaret P (2002) The w3c document object model (DOM). http://www.w3.org/2002/07/26-dom-article.html

Leveson NG, Turner CS (1993) An investigation of the therac-25 accidents. Computer 26(7):18–41

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, pp 652–656

Martinie C, Palanque P, Barboni E, Ragosta M (2011) Task-model based assessment of automation levels: application to space ground segments. In: Proceedings of the 2011 IEEE international conference on systems, man, and cybernetics. Piscataway, IEEE, pp 3267–3273

Martinie C, Navarre D, Palanque P (2014) A multi-formalism approach for model-based dynamic distribution of user interfaces of critical interactive systems. Int J Hum-Comput Stud 72(1):77–99

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

NTSB (2001) Runway Overrun During Landing, American Airlines Flight 1420, McDonnell Douglas MD-82, N215AA, Little Rock, Arkansas, June 1, 1999 Technical Report NTSB/AAR-01/02. National Transportation Safety Board, Washington, DC

NTSB (2015) Runway Overrun During Rejected Takeoff Gulfstream Aerospace Corporation G-IV, N121JM, Bedford, Massachusetts, May 31, 2014 Technical Report NTSB/AAR-15/03. National Transportation Safety Board, Washington, DC

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 Ltd., London, pp 189–212

Pan D, Bolton ML (2015) A formal method for evaluating the performance level of human-human collaborative procedures. In: Proceedings of HCI international. Springer, Berlin, pp 186–197

Pan D, Bolton ML (2016) Properties for formally assessing the performance level of human-human collaborative procedures with miscommunications and erroneous human behavior. Int J Ind Ergon (in press). doi:10.1016/j.ergon.2016.04.001

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

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 Ltd, London, pp 362–369

Perrow C (1999) Normal accidents: living with high-risk technologies. Princeton University Press, Princeton

Reason J (1990) Human error. Cambridge University Press, New York

Shankar N (2000) Symbolic analysis of transition systems. Proceedings of the international workshop on abstract state machines, theory and applications. Springer, London, pp 287–302

Sheridan TB, Parasuraman R (2005) Human-automation interaction. Rev Hum Factors Ergon 1(1):89–129

Syncro Soft (2016) Relax NG schema diagram. In: User Manual of Oxygen XML Editor 17.1. https://www.oxygenxml.com/doc/versions/17.1/ug-editor/index.html#topics/relax-ng-schema-diagram.html

van Paassen MM, Bolton ML, Jimenez N (2014) Checking formal verification models for human-automation interaction. 2014 IEEE international conference on systems, man and cybernetics (SMC). IEEE, Piscataway, pp 3709–3714