This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Automatic Detection of Potential Automation Surprises for ADEPT Models

Sébastien Combéfis, *Member, IEEE*, Dimitra Giannakopoulou, and Charles Pecheur, *Member, IEEE*

***Abstract*—This paper describes how to automatically detect *potential automation surprises* in interactive systems, within a rapid automation interface design tool named ADEPT. The proposed analysis method in this paper is based on a conformance relation, called *full-control*, between the model of the actual system and a mental model of it, that is, its behavior as perceived by the operator. The method can, among other things, automatically generate a so-called *minimal full-control mental model* for a given system. Systems are well designed if they can be described by relatively simple mental models for their operators, which can be assessed with the minimal full-control mental model generation algorithms. During the generation, potential automation surprises are detected and highlighted with execution examples that may lead to confusion. The analysis methods are based on an enriched version of labeled transition systems to describe the system and mental models. In order to be able to integrate the analysis method within ADEPT, a semantics for ADEPT models makes it possible to translate them into enriched LTSs. The proposed translation is automated for a specified class of ADEPT models that are characterized and defined in this paper. A case study demonstrates the proposed analysis framework and informs how the integration with ADEPT can be improved.**

***Index Terms*—ADEPT toolset, formal methods, human factors, human–machine interaction.**

I. Introduction

With complex systems, accidents may in part be due to *automation surprise* [1], [2] or *mode confusion* [3], which is a particular type of automation surprise. In those situations, the operator is surprised during the interaction because the system is not reacting as expected, that is, it is doing something not foreseen by the operator.

Approaches using *formal methods* to tackle various problems in *human–machine interaction* (HMI) have been developed, exploiting the rigorous and systematic analyses brought by formal methods [4]–[7]. In order to be usable and adopted by system designers, formal-method-based techniques must be supported by tools. In addition to increase use, formal models manipulated by system designers should be compatible with the way they think about systems and interactions.

Manuscript received May 1, 2014; revised October 5, 2014 and January 29, 2015; accepted March 17, 2015. This paper was recommended by Associate Editor D. Gillan.

S. Combéfis is with the École Centrale des Arts et Métiers, 1200 Bruxelles, Belgium (e-mail: s.combefis@ecam.be).

D. Giannakopoulou is with the NASA Ames Research Center, Mountain View, CA 94035 USA (e-mail: dimitra.giannakopoulou@nasa.gov).

C. Pecheur is with the ICT, Electronics and Applied Mathematics Institute, Université Catholique de Louvain, 1348 Louvain-la-Neuve, Belgium (e-mail: charles.pecheur@uclouvain.be).

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

Digital Object Identifier 10.1109/THMS.2015.2424851

This paper addresses integrating a given analysis method based on formal methods with an existing tool used by system designers. The considered method is used for the automatic detection of potential automation surprises and is implemented as a prototype command-line tool. The method is based on models described with enriched labeled transition systems called HMI-LTS [8]. The considered tool with which the formal-method-based analysis is integrated is *Automatic Design and Evaluation Prototyping Toolset* (ADEPT) [9].

The HMI-LTS models are conceptually far from the way system designers think about systems. This paper introduces a new formalism that extends HMI-LTSs to support a direct mapping from ADEPT models. Models in the new formalism can be expanded into HMI-LTSs, which enables the application of all formal analysis techniques defined for HMI-LTSs. The paper proposes a formal semantics for ADEPT models; this semantics forms the basis for an automated translation from ADEPT models into the new proposed formalism.

The remainder of this paper is structured as follows. Section II presents the related work about applying formal methods to the analysis of HMI, with a focus on techniques supported by tools. Section III presents the analysis method on which the work of this paper relies. Section IV describes the ADEPT tool and proposes a formal semantics for ADEPT models, that is used to support formal-method-based analysis. Section V presents an enriched version of HMI-LTS that is used as an intermediary step for the translation of ADEPT models. Section VI presents a case study, and Section VII discusses the work.

II. Related Work

Researchers have been working on the use of techniques and algorithms based on formal methods in order to analyze various aspects of HMI [4]–[7], [10].

Campos *et al.* [11], [12] propose a model-checking framework for the analysis of HMI. They define a set of generic parameterized usability properties in the *computation tree logic* temporal logic. The *Symbolic Model Verifier* model checker is used to check properties against systems modeled with the MAL modal logic. Their analysis framework is supported by the IVY tool [13]. The framework presented in this paper works with the full-control property, which is more general than the usability properties supported by their work.

Thimbleby *et al.* [14]–[16] use graphs to model the system with a focus on user interface (UI). Their analyses focus on structural properties and measures such as the maximum degree or the value of centrality. In contrast with our work, there is little focus on the dynamic aspects of interactions.

2168-2291 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

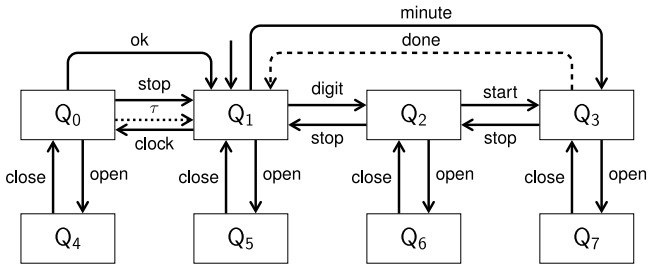


Fig. 1. HMI-LTS model of a simple model of a microwave that shows the current time (that can be changed) and that can cook food with two cooking modes (1-min or specified duration cooking). Commands are depicted with solid lines (⟶), observations with dashed lines (– – –⩾), and internal actions with dotted lines (·····⩾).

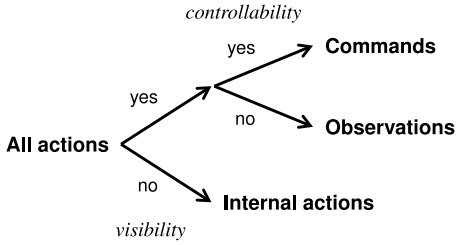


Fig. 2. Classification of actions for HMI-LTS into three sets according to the visibility and the controllability criteria.

Navarre *et al.* [17], [18] have developed a formal analysis framework based on Petri nets and supported by the Petshop tool [19]. Their focus is on the combination of user task models and system models. The present work does not focus on user task models, although the possibility has been explored in [20].

Bolton *et al.* [21]–[23] also consider user task models. More precisely, they have developed a framework to analyze and predict the impact of human errors and system failures. Their analysis is based on task-analytic models and taxonomies of erroneous human behavior. In their work, models are merged with the properties to be checked into a single model that is verified with the SAL model checker.

Concerning automation surprises, and in particular mode confusion, Rushby *et al.* [24], [25] and Bredereke and Lankenau [26], [27] have worked on formalizing mode confusion and have proposed techniques to reduce it. The former used the Murϕ model-checker, while the latter worked on specification/implementation refinement relations.

The aforementioned research proposes formal frameworks for the analysis of HMI and develops corresponding automated tools, such as IVY and Petshop, for example. In our work, a formal framework has been developed starting from the work of Degani and Heymann [28]–[30]. However, whereas Shiffman *et al.* [31] proposed a simple tool for those techniques, this work focuses on usability. Instead of developing a new custom tool, the approach proposed in this paper integrates the analysis techniques of the proposed formal framework within an existing tool developed for and used by system designers. System designers are not accustomed to manipulating mathematical objects and constructs used by formal methods, but are using models that are supported by existing modeling tools. The main motivation of this work is to bring the two worlds closer by making formal-methods-based analysis available within the environments that are familiar to system designers.

III. Modeling and Analyzing Human–Machine Interaction Systems

This section provides an overview of the formal framework illustrated with a simple model of a microwave (see Fig. 1). The door of the microwave can be opened or closed. When the door is closed, the user can perform two kinds of tasks. The clock of the microwave can be changed and set to a new value. The microwave can also be programmed to cook, either during 1 min with the 1-m cooking option or for a specified amount of time, to be entered by the user.

A. Human–Machine Interaction Labeled Transition Systems

We model HMI systems with an enriched version of labeled transition systems called HMI-LTSs [8]. An HMI-LTS is a directed graph whose edges are labeled with actions. The difference with (classical) LTSs is that three kinds of actions are considered.

1) *Commands* are actions executed by the operator on the system; they correspond to inputs to the system.
2) *Observations* are actions autonomously triggered by the system and observed by the operator; they correspond to outputs produced by the system.
3) *Internal actions* are neither controlled nor observed by the operator; they correspond to actions belonging to the internal behavior of the system.

Fig. 2 shows the three types of actions. The first classification level is related to whether or not the action is visible by the operator. The second classification level is related to whether or not the action is controlled by the operator.

Examples of commands include open that corresponds to the opening of the microwave door and digit which is the action of entering the cooking time. There is only one observation, namely done. It corresponds to a signal that the microwave emits when the cooking is done. There is only one internal action that represents the fact that the microwave does not stay in the $\mathbb{Q}_0$ state if the operator has not entered any number for a certain period of time. An arrow without source state is used to point to the initial state of the system, in this case, $\mathbb{Q}_1$.

The distinction between commands and observations plays a crucial role when studying HMI [30], [32]. While commands are under the control of the operator, observations are autonomously triggered by the machine. Any unexpected observation may surprise the operator and lead to a future erroneous interaction that could consequently lead to an accident.

Formally, an HMI-LTS is defined as a five-tuple $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow \rangle$, where S is the set of states, $\mathcal{L}^c$ and $\mathcal{L}^o$ are, respectively, the sets of commands and observations, $s_0 \in S$ is the initial state, and $\rightarrow \subseteq S \times (\mathcal{L}^c \cup \mathcal{L}^o \cup \{\tau\}) \times S$ is the transition relation. Since internal actions cannot be distinguished by the operator, they are all denoted with the same symbol τ. Finally,

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

commands and observations are both visible to the operator, and their set is denoted $\mathcal{L} = \mathcal{L}^c \cup \mathcal{L}^o$.

When a transition with the action $\alpha \in \mathcal{L}$ exists between states s and s', which is denoted $s \xrightarrow{\alpha} s'$, the action α is said to be *enabled* in state s. The set of commands (respectively, observations) that are enabled in a given state s is denoted $\Gamma^c(s)$ (respectively, $\Gamma^o(s)$). The intuition behind enabled commands (respectively, observations) is that they can be directly executed (respectively, observed) by the operator in the current state of the system. This set is important for commands since it represents the commands that, when entered by the user on a system, are accepted by the system.

Internal actions may take place between observable actions. A *weak transition* between states s and s' with the action α is a sequence of transitions $s \xrightarrow{\tau} ... \xrightarrow{\alpha} ... \xrightarrow{\tau} s'$ that is denoted $s \xrightarrow{\alpha} s'$. The set of commands (respectively, observations) that are *possible* in a given state s is denoted $A^c(s)$ (respectively, $A^o(s)$) and correspond to the set of $\alpha \in \mathcal{L}^c$ (respectively, $\mathcal{L}^o$) such that there exists a s' such that $s \xrightarrow{\alpha} s'$. The intuition behind possible commands (respectively, observations) is that they may be executed (respectively, observed) after some interval that cannot be predicted through observable behavior. Indeed, some internal actions may be executed before a possible command (respectively, observation) is accepted (respectively, is produced) by the system. An action that is enabled is by definition also a possible one, that is, $\Gamma^c(s) \subset A^c(s)$ and $\Gamma^o(s) \subset A^o(s)$.

The state $\mathtt{Q}_0$ in Fig. 1 has three enabled commands and six possible commands. The enabled commands are open, ok, and stop. The possible commands additionally include clock, digit, and minute. Those three latter commands become enabled when the system has transitioned to the $\mathtt{Q}_1$ state following the $\mathtt{Q}_0 \xrightarrow{\tau} \mathtt{Q}_1$ internal action.

A *trace* $\sigma = \langle \alpha_1, ..., \alpha_n \rangle$ is a sequence of actions such that there exists an *execution* $s_0 \overset{\alpha_1}{\Longrightarrow} s_1 ... s_{n-1} \overset{\alpha_n}{\Longrightarrow} s_n$ starting from the initial state of the HMI-LTS. The set of traces of an HMI-LTS $\mathcal{M}$ is denoted $\mathbf{Tr}(\mathcal{M})$. Intuitively, a trace represents a sequence of actions that can be observed during an interaction between a machine and its operator. Only the visible behavior is contained in the trace since internal actions are hidden.

Finally, an HMI-LTS is said to be *deterministic* if and only if for any state, there is at most one state that can be weakly reached for every action. Consequently, deterministic HMI-LTSs do not contain any τ-transition, and the execution of a given trace from the initial state always leads to the same state, no matter what path is taken. For example, the system model of Fig. 1 is not deterministic since, starting from the initial state $\mathtt{Q}_1$, after the execution of the $\langle \mathtt{clock} \rangle$ trace, the system can either be in state $\mathtt{Q}_0$ or back in state $\mathtt{Q}_1$ if the $\mathtt{Q}_0 \xrightarrow{\tau} S_1$ transition took place after the $\mathtt{Q}_1 \xrightarrow{\mathtt{clock}} \mathtt{Q}_0$ transition.

B. Full-Control Property

Systems and mental models are both modeled with HMI-LTSs, but mental models are always considered deterministic in this work. This hypothesis means that, whenever the operator performs an action, he/she knows his/her next (mental) state. A mental model for a given system captures a simplified view of the system, as perceived by the operator. The *full-control* property relates a mental model with a system model. It captures the fact that the mental model is complete enough to enable a complete control of the system, avoiding automation surprises. The full-control property requires that, at any time during the interaction, the two following conditions hold.

1) The operator knows precisely what are the possible commands on the system; the set of commands that are possible on the system must be exactly the same as the set of commands that are possible in the mental model.
2) The operator is aware of at least all the observations that could occur on the system; the set of observations that are possible according to the mental model must contain all the possible observations on the system.

Definition 1. Given $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ and $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$, the mental model $\mathcal{H}$ allows full-control of the system model $\mathcal{S}$, which is denoted $\mathcal{H}\mathbf{fc}\mathcal{S}$, if and only if $\forall \sigma \in \mathcal{L}^*$ such that $s_{0_S} \overset{\sigma}{\Longrightarrow} s_S$ and $s_{0_H} \xrightarrow{\sigma} s_H$:

$$A^c(s_S) = A^c(s_H) \text{ and } A^o(s_S) \subseteq A^o(s_H). \tag{1}$$

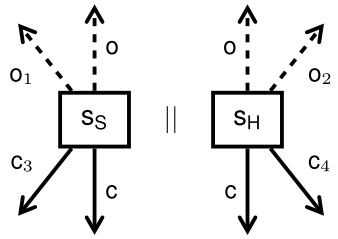


Fig. 3. Four potentially surprising situations that may occur during the interaction between a human $\mathcal{H}$ and a system $\mathcal{S}$ being used, in states of the interaction model.

Fig. 3 shows the four potentially surprising situations that may arise during any interaction between an operator and a system. During the interaction, the system is in a state S_S according to the system model, and the system is in the state S_H for the operator according to the mental model. In both states, there are commands and observations that are possible. The full-control property guarantees that three of those situations are avoided.

1) There is an observation o_1 that can be produced by the system but not foreseen by the operator. It can be dangerous if the system is producing an unexpected hazard signal.
2) There is a command c_3 that is available on the system but not present in the mental model. It is not precisely a surprising situation, but it prevents the operator to use all the functionalities of the system.
3) There is a command c_4 that the operator can execute according to his/her mental model but that is not possible on the system. That can surprise the operator since he/she will expect some feedback from the system that will never happen.

The last situation, that is, an observation o_2 present in the mental model, but that will never occur according to the system, is permitted by the full-control property. It is precisely the second condition of the full-control property.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

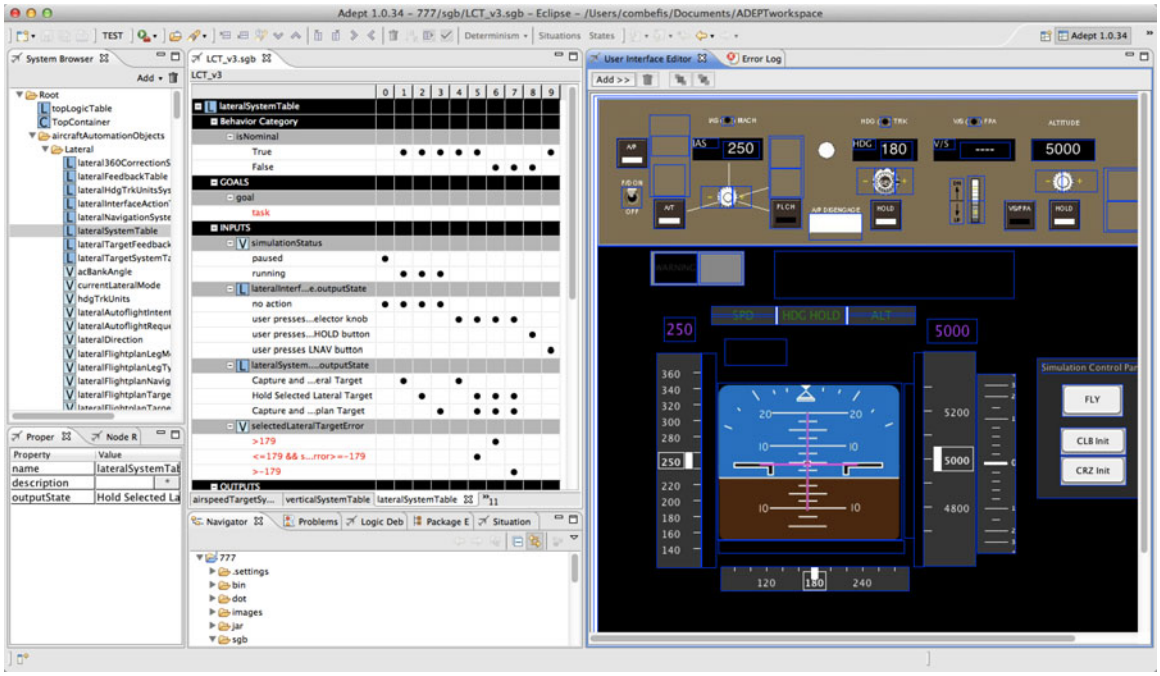


Fig. 4. Model of the autopilot of the B777 loaded in the ADEPT tool. There are three parts in the main window: 1) the left pane is used to navigate through the model, 2) the right pane shows a usable prototype of the UI, and 3) and the middle pane shows tables encoding the description of the behavioral aspects of the model.

The full-control property guarantees that all the functionalities offered by the system (the commands) are known by the operator. This is a strong requirement since an operator is not always interested in all the functionalities. A variant of the full-control property where the first condition is replaced by $A^c(s_S) \supseteq A^c(s_H)$ can be considered. However, this work aims at generating the minimal full-control mental model for a given system. In such a setting, considering the proposed variant of the full-control property would lead to a minimal mental model that does not allow any commands.

Generally speaking, for a given system model, there is no guarantee that there exists a full-control mental model. As developed in [33], the existence of a full-control mental model for a given system model is only guaranteed if the system model satisfies the *fc-determinism* property. That property states that, for any trace, all the states that can be reached with that trace starting from the initial state must have exactly the same set of possible commands. If a system model is not fc-deterministic, an execution trace can be found that may lead to a potential automation surprise. If such a situation appears, the system model should be redesigned in such a way that a full-control mental model exists for it.

C. Detection of Automation Surprises

A formal framework to analyze system models has been developed based on the full-control property [8]. The main analysis proposed in the framework consists in the automatic generation of a minimal full-control mental model for a given system model. Two algorithms to perform that generation are proposed in [33] and [34]. The first one is based on a variant of a bisimulation relation between the states of the system, allowing to merge together those which exhibit a similar behavior for the operator. The second one uses an active learning algorithm that iteratively builds mental model candidates until reaching a minimal one allowing full-control of the system model. Each algorithm has its strengths and weaknesses, and one or the other may work better for different systems. In the remainder of the paper, "the algorithm" refers to either of the two algorithms.

The algorithm, once run on a given system model, either succeeds and produces a minimal full-control mental model, or fails if the system is not controllable according to the full-control property. More precisely, the algorithm builds an abstraction of the system model that can be considered as a "perfect" mental model for the system, in the sense of the full-control property. If the algorithm fails, it highlights an error trace as an example of a potential automation surprise.

This work only focuses on the analysis of a given system model, even if the used formal framework proposes many other analyses. The reason for this focus is the ADEPT toolset is used by system designers to model systems and to check properties on them.

IV. Automatic Design and Evaluation Prototyping Toolset

ADEPT [9] is a Java-based tool developed at NASA Ames Research Center that supports designers in the early prototyping stages of interface design. ADEPT additionally offers a set of basic analyses that can be applied to the model being developed. Fig. 4 shows the main window of the tool where a model of the autopilot of the B777 has been loaded. The models developed

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

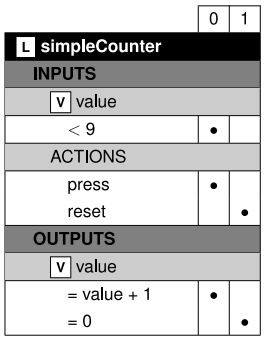


Fig. 5. ADEPT model of a simple counter system, whose value ranges between 0 and 9.

within ADEPT can be directly executed and simulated by the designers, but can also be analyzed by systematic and rigorous techniques.

A. Automatic Design and Evaluation Prototyping Toolset Models

An ADEPT model consists of two elements: a set of *logic tables* is coupled with an *interactive UI*. The logic tables are used to describe the dynamic aspects of the system. Their role is to describe how the state of the system evolves in reaction to actions by the user and events coming from the environment or internal to the system. The state of the system includes the state of the UI. The UI is a set of components encoded as Java objects representing graphical widgets, going from simple buttons to more complex widgets such as the primary flight display used in avionics and visible on Fig. 4. Finally, an ADEPT model also contains other elements such as timers, system variables, and functions, each of them related to a specific Java construction, such as instance variables and methods, for example.

The logic tables refer to the elements of the UI and to the other components of the ADEPT model via their Java instances variables. The interactions are performed through calls to the methods of those components, using the Java syntax. In particular, UI events can be seen as Boolean variables indicating whether the event occurred. Those variables can then be used in the logic tables.

B. Automatic Design and Evaluation Prototyping Toolset Logic Tables

Fig. 5 shows an example of an ADEPT model representing a simple counter system that counts from 0 to 9 in response to successive presses on a button. It can also be reset to the value 0 with another button. The model is composed of a single logic table.

Logic tables are 2-D tables structured with two parts: INPUTS and OUTPUTS (identified by the dark gray rows). Each of those parts is further structured into a two-level hierarchy of elements and values. Values in the input part represent conditions, whereas they correspond to statements in the output part. In addition to that vertical organization, logic tables are also structured horizontally: the left-part contains the element-values pairs and the right part consists of a sequence of columns.

Each block of the input and output parts is related to one variable of the system (identified by the light gray rows) and includes a sequence of possible values for the variable. A special variable named ACTIONS represents the action performed by the user. Each column on the right part of the table represents a possible scenario of operation.

For example, the input part of the first scenario (column 0) corresponds to the following condition: value $< 9 \wedge$ ACTIONS $=$ press. If that condition is satisfied in the current state of the system, the corresponding output part is executed. For the first scenario, the output corresponds to the following variable assignment statement: value $\leftarrow$ value + 1.

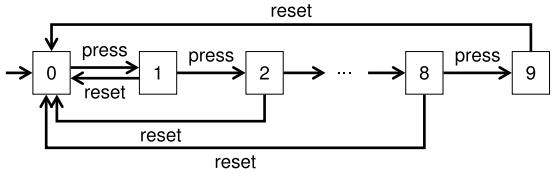


Fig. 6. Representation of the behavior of the ADEPT model of Fig. 5.

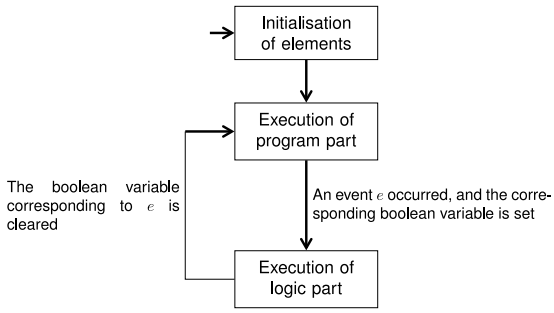


Fig. 7. Behavior of an ADEPT model.

Fig. 6 shows a representation of the behavior of the ADEPT model example of the simple counter. Values inside the boxes correspond to the value variable and the ACTIONS are put on the arrows linking the boxes. This representation is an LTS, with additional information on the states, which motivates the translation to HMI-LTS presented in this work.

C. Formal Semantics of Automatic Design and Evaluation Prototyping Toolset

This section proposes a formal semantics for ADEPT models. Such a semantics is necessary to perform formal-method-based analysis of ADEPT models. We use the example of Fig. 5 to illustrate different elements of the semantics.

An ADEPT model has two parts: the *logic part* is described by the logic tables and the *program part* is defined by the UI and other components. Fig. 7 summarizes the behavior of an ADEPT model. The elements of the model are first initialized. After initialization, the behavior consists of a loop alternating between the logic and program parts as events are occurring.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

1) Logic Part: A *row header* is a sequence of *switches*, which are used to associate an element with a sequence of values. Input and output headers are different, and a *table header* is a pair consisting of one input header and one output header. The simple counter ADEPT table has two input switches: (value, (<9)) and (ACTIONS, (press, reset)) and one output switch: (value, (= value + 1, = 0)).

$$Header = IHeader \times OHeader$$

$$IHeader = ISwitch^* \qquad OHeader = OSwitch^*$$

$$ISwitch = IElem \times IVal^* \; OSwitch = OElem \times OVal^*.$$

The *columns* define elementary fragments of the dynamic behavior and are referred to as *situation-automation behavior pairs* in [35]. One column can be seen as one possible execution scenario for the behavior of the system.

A *row index* is a pair $y = (i, j) \in \mathbb{N} \times \mathbb{N}$, denoted $i.j$ that represents the jth value of the ith element. For example, the row index 2.1 represents the first value of the second element of the INPUTS part; therefore, it corresponds to the press value of the ACTIONS element

$$Index = \mathbb{N} \times \mathbb{N}.$$

An *index set* is a set $Y \subset \mathbb{N} \times \mathbb{N}$ of row indices. An index set is an *index range* if its first indices cover the range of consecutive natural numbers $1, \ldots, n$ and its second indices cover the range a consecutive natural numbers $1, \ldots, n_i$ for each first index i. Finally, the subset $Y|_i = \{i.j \in Y\}$ contains row indices whose first index is i. For example, the set $Y = \{1.1, 2.1, 2.2\}$ is an index range that corresponds to the whole INPUTS part and $Y|_2 = \{2.1, 2.2\}$.

An (input or output) header H is a mapping from row indices to $(element, value)$ pairs. For a given $H = (H_1, ..., H_n)$, with $H_i = (E_i, (V_{i,1}, ..., V_{i,n_i}))$, $H(i.j) = (E_i, V_{i,j})$ provided that $1 \leq i \leq n$ and $1 \leq j \leq n_i$. For example, for the OUTPUTS part, $H(1.2) = (value, = 0)$.

A column gives a binary value for every row of the logic table that corresponds to a value. In the proposed semantics, columns are defined as two sets C^I and C^O so that $i.j \in C^I$ if and only if row $i.j$ is marked in that column, and similarly for C^O. Those two sets are included in the domains of the input and output headers of the logic table. For example, $C^I = \{1.1, 2.1\}$ is the column corresponding to the scenario 0, for the INPUTS part

$$Col = 2^{Index} \times 2^{Index}.$$

A *logic table* is a structure $T = (H^I, H^O, CC)$, where $H^I \in IHeader$ is an input header, $H^O \in OHeader$ is an output header, and $CC \in Col^*$ is a list of columns $C = (C^I, C^O)$ where $C^I \subseteq$ dom (H^I) and $C^O \subseteq$ dom (H^O)

$$Table = IHeader \times OHeader \times Col^*.$$

The logic table corresponding to the simple counter example of Fig. 5 can be defined as (H^I, H^O, CC) with

1) H^I = ((value, (< 0)), (ACTIONS, (press, reset)))
2) H^O = ((value, (= value + 1, = 0)))
3) $CC = ((\{1.1, 2.1\}, \{1.1\}), (\{2.2\}, \{1.2\}))$.

Finally, an ADEPT program M (the logic part of it) is a collection of named logic tables (given a set of names $Name$), such that all the names are different and with one distinguished name top: $M : Name \rightarrow Table$.

2) Program Part: The program part captures the behavior related to the interaction of the user with the UI. All the elements of the program parts result, directly or indirectly, into the generation of executable Java code. Five different types of entities are used to build an ADEPT model.

- 𝕍 *System variables* correspond to Java variables. They are used in input and output switches and can appear as elements or as values.
- ℂ *UI components* correspond to Java GUI widgets from the UI. GUI events are used in input switches for the ACTIONS element. GUI component attributes are used in output switches as elements whose values can be changed. Finally, arbitrary GUI components methods can be invoked in output switches for the PRIMITIVES element.
- 𝕆 *Timers* are used to schedule repetitive events. Timer events can be used in input switches for the ACTIONS element and timer methods can be invoked in output switches for the PRIMITIVES element.
- 𝔽 *Functions* correspond to Java methods which return a value. They are used in output switches as values.
- 𝕃 *Logic tables* appear in output switches for the LOGIC element. Furthermore, each logic table has an associated variable T.outputState that can appear in both input and output switches as element.

Input switches define conditions that correspond to Java Boolean expressions. They are derived from input element-value pairs, that is, $B = cond(E^I, V^I)$:

$$cond : IElem \times IVal \mapsto JavaExpr.$$

The $cond$ function is defined differently according to the type of the E^I element, as described hereafter:

1) $cond$ (𝕍 var, expr) = var == expr
2) $cond$ (𝕍 var, str) = var str
3) $cond$ (ACTIONS, ℂ component.event) = component.event
4) $cond$ (ACTIONS, 𝕆 timer.event) = timer.event
5) $cond$ (𝕃table.outputState, value) = table.outputState == value

For example, the $cond$ function applied to the (value, < 9) input element-value pair results in the following Java Boolean expression: value < 9.

Output switches define actions that correspond to Java statements. They are derived from output element-values pairs, that is, $S = stmt(E^O, V^O)$:

$$stmt : OElem \times OVal \mapsto String.$$

The $stmt$ function also depends on the type of the E^O element and is defined as follows:

1) $stmt$ (LOGIC, table) = call (table)
2) $stmt$ (PRIMITIVES, 𝕆 timer.method) = timer.method()
3) $stmt$ (PRIMITIVES, statement) = statement
4) $stmt$ (𝕍 var, expr) = var = expr
5) $stmt$ (𝕍 var, 𝕍 var') = var = var'

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

6) $stmt$ (𝕍 var, 𝔼fun) = var = fun()
7) $stmt$ (𝕍 var, str) = var str
8) $stmt$ (𝕃 table.outputState, value) = table.outputState = value
9) $stmt$ (ℂ component.field, value) = component.field = value

For example, the $stmt$ function applied to the (value, = value + 1) output element-value pair results in the following Java statement: value = value + 1.

3) Execution Semantics: The state of an ADEPT model is composed of two elements: the values of the system variables and the individual states of the different components including UI components and timers. Those two elements belongs to the program part, which, therefore, defines the set of possible states $q \in Q$ of the ADEPT model.

For a logic table to be executable without ambiguity, it must be *well formed*. If a logic table is well-formed, it is guaranteed that there is exactly one column (at most one for ACTIONS) that will be applied for any invocation of a table and that only one condition is true for any input switch in any state.

The well-formed property can be formally defined given the following definitions. Given an index set Y, the *choices* over Y are defined as the set $choices(Y) = Y|_1 \times \cdots \times Y|_n$, where $n = \max\{i | i.j \in Y\}$. Given an index set C (from an input column) and an index range Y (from the input header), the *don't care expansion* of C with respect to Y is defined as $expand(C, Y) = C \cup \bigcup \{Y|_i \mid C \cap Y|_i = \emptyset\}$.

Property 2. Two requirements have to be satisfied for a logic table (H^I, H^O, CC) to be well-formed:

1) (***R1***) Let $CC = (C_1, \ldots, C_m)$ and $C_i = (C_i^I, C_i^O)$. For any choice $C^* \in choices(\text{ dom } (H^I))$, there is exactly one $1 \leq i \leq m$ such that $C^* \subseteq expand(C_i^I, \text{ dom } (H^I))$.
2) (***R2***) For any input switch $(E^I, (V_1^I, \ldots, V_k^I))$, where $E^I \neq$ ACTIONS (respectively, $E^I =$ ACTIONS), for any state q, there is a unique (respectively, at most one) i such that $1 \leq i \leq k$ and $eval(cond(E^I, V_i^I), q) = T$.

Intuitively, ***R2*** ensures that only one condition is true for any input switch in any state, and *R1* ensures that any combination of such conditions is covered in any table. Note that *R1* can be checked algorithmically, and such a check is available within ADEPT. On the other hand, *R2* is much harder to check since it involves arbitrary Java code; it is even undecidable in general. For example, for the (value, (< 9)) input switch of the logic table of Fig. 5, *R2* means that the Java expression value < 9 must be *true* for any state q. It is generally not possible to check it algorithmically, the only way is to explore all the possible states q and check the value of the condition.

It is possible to define the execution of an ADEPT model using the $eval$ and $exec$ functions. The first function is used to evaluate the value of a condition B in a state q and is defined as $eval(B, q) \in \{T, F\}$. The second function, defined as $exec(S, q) \in Q$, is used to compute the state in which the system enters after execution of a statement S.

The proposed execution semantics provided in Fig. 8 defines semantic mapping $[\![\alpha]\!](q)$ denoting the semantics of the syntactic construct α in a state q of the model, for different constructs

$$
\begin{array}{rcl}
[\![M]\!] & : & Q \mapsto Q \\
[\![M]\!](q) & = & [\![M(top)]\!](q) \\
\\
[\![T]\!] & : & Q \mapsto Q \\
[\![(H^I, H^O, CC)]\!](q) & = & [\![(H^I, H^O)/CC]\!](q) \\
\\
[\![((H_1^I, H_1^O), \cdots, (H_n^I, H_n^O))]\!] & : & Q \mapsto Q \\
[\![((H_1^I, H_1^O), \cdots, (H_n^I, H_n^O))]\!](q) & = & \begin{cases} [\![H_i^O]\!](q) & \text{if } \exists i \cdot [\![H_i^I]\!](q) \\ q & \text{otherwise} \end{cases} \\
\\
[\![H^I]\!] & : & Q \mapsto \{T, F\} \\
[\![(S_1^I, \cdots, S_n^I)]\!](q) & = & [\![S_1^I]\!](q) \wedge \cdots \wedge [\![S_n^I]\!](q) \\
\\
[\![S^I]\!] & : & Q \mapsto \{T, F\} \\
[\![(E^I, (V_1^I, \cdots, V_k^I))]\!](q) & = & [\![(E^I, V_1^I)]\!](q) \vee \cdots \vee [\![(E^I, V_k^I)]\!](q) \\
\\
[\![(E^I, V^I)]\!] & : & Q \mapsto \{T, F\} \\
[\![E^I, V^I]\!](q) & = & eval(cond(E^I, V^I), q) \\
\\
[\![H^O]\!] & : & Q \mapsto Q \\
[\![(S_1^O, \cdots, S_n^O)]\!](q) & = & ([\![S_n^O]\!] \circ \cdots \circ [\![S_1^O]\!])(q) \\
\\
[\![S^O]\!] & : & Q \mapsto Q \\
[\![(E^O, (V_1^O, \cdots, V_k^O))]\!](q) & = & ([\![(E^O, V_k^O)]\!] \circ \cdots \circ [\![(E^O, V_1^O)]\!])(q) \\
\\
[\![(E^O, V^O)]\!] & : & Q \mapsto Q \\
[\![E^O, V^O]\!](q) & = & exec(stmt(E^O, V^O), q)
\end{array}
$$

Fig. 8. Execution semantics of an ADEPT model $M \in Model$ for a given state $q \in Q$. The distinguished table $M(top)$ is the entry point of the execution.

α. The result is a Boolean value for input constructs and a new state for output constructs. The execution semantics uses a *projection* operator of headers on columns $(H^I, H^O)/C$. Given a header $H = (H_1, ..., H_n)$ and an index set $Y \subseteq$ dom (H), the projection of H on Y is defined as

$$H/Y = (H_1/Y, ..., H_n/Y)$$

where, for each $H_i = (E_i, (V_{i,1}, ..., V_{i,n_i}))$,

$$H_i/Y = (E_i, (V_{i,j} \mid i.j \in Y, 1 \leq j \leq n_i))$$

The projection is extended to a column $C = (C^I, C^O)$ as

$$(H^I, H^O)/C = \left(H^I, expand\left(C^I, H^I\right), H^O/C^O\right)$$

and then to a list of columns $CC = (C_1, ..., C_m)$ as

$$\left(H^I, H^O\right)/CC = \left(\left(H^I, H^O\right)/C_1, ..., \left(H^I, H^O\right)/C_m\right).$$

The two requirements for well-formed logic tables ensure that there is at most one i such that $[\![H_i^I]\!](q)$ for any projected input column $(H_1^I, ..., H_{n_i}^I)$. Moreover, the second requirement guarantees that there is at most one i such that $[\![E^I]\!](V_i^I)$, and exactly one except if $E^I =$ ACTIONS.

V. State Event Models

HMI-LTS are event-based models, that is, they do not carry any explicit information on their states, except information about what are the possible actions for each state. ADEPT models combine state with transition information. In particular, the state of an ADEPT model consists of the values of a set of system variables. Among those variables, there are some *visible variables* whose values can be observed by the operator through the UI. The ADEPT logic tables describe how the system evolves

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

from one state to another, reacting to an action of the operator, while the observable state changes.

As in [36], a direct translation from ADEPT models to HMI-LTSs proved challenging. In order to tackle this translation problem, HMI-LTSs have been extended to support information on states in the style of Kripke structures [37].

First, *HMI state valued system models* (HVSs) enrich HMI-LTSs with a set of *state values* and a mapping function associating every state of the model to one state value. Formally, an HVS is a tuple $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v, \mathcal{O}\rangle$, where $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow\rangle$ is an HMI-LTS, $\mathcal{L}^v$ is the set of state values, and $\mathcal{O} : S \mapsto \mathcal{L}^v$ is the state value mapping function. HVSs are used to model systems and possess two different kinds of observations that are interpreted differently according to an HMI point of view. Observations of the system state are optional, whereas observations labeling transitions have to be observed for the interaction to proceed. Note that an operator may miss an observation that is on a transition.

Going back to the microwave example of Fig. 1, it can be enriched with the addition of two state-variables mode and opstate (operational state). The mode state-variable has two possible values: operational (when the door is closed) or disabled (when the door is opened). The opstate state-variable has four different values: settime, idle, program, and cook. Each state value corresponds to a valuation of the two system variables. For example, the state $\mathtt{Q}_1$ has the state value $\langle \mathtt{mode} = \mathtt{operational}, \mathtt{opstate} = \mathtt{idle}\rangle$ and the state $\mathtt{Q}_7$ has the state value $\langle \mathtt{mode} = \mathtt{disabled}, \mathtt{opstate} = \mathtt{cook}\rangle$.

Second, *HMI state valued mental models* (HVMs) are enriched HMI-LTSs that also include state values and are used to model mental models. State values are taken into account with the addition of *action guards* to the model. The action guards are put on the transitions, and for the transition to be fired, the state value of the current state of the system must conform with the action guard. Formally, an HVM is a tuple $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v\rangle$, where $\mathcal{L}^v$ is the set of state values, $\rightarrow \subseteq S \times \mathcal{L}^v \times \mathcal{L} \times S$ and $\langle S, \mathcal{L}^v \times \mathcal{L}^c, \mathcal{L}^v \times \mathcal{L}^o, s_0, \rightarrow\rangle$ is a deterministic HMI-LTS without τ-transition.

Different kinds of models are used for the system and mental models. More precisely, HVSs and HVMs differ regarding how they handle observations on states. A system model should expose to the operator the observable part of its current state, even though the operator may not find it useful and, therefore, may choose to ignore it. As a consequence, state values are attached to states. In the mental model, the behavior of the operator depends on the current state of the system, and therefore, transitions must contain guards on the system state. Moreover, in a given state, the mental model can have different transitions with different action guards. Therefore, state values are attached to the transitions.

The enriched models allow for even more compact mental models. Depending on what state variables are visible, thus changing the set of state values, the minimal full-control mental models vary. Fig. 9 shows an example of a minimal full-control mental model for the microwave system of Fig. 1 considering that only the opstate system variable is visible, and removing the τ transition from the system.

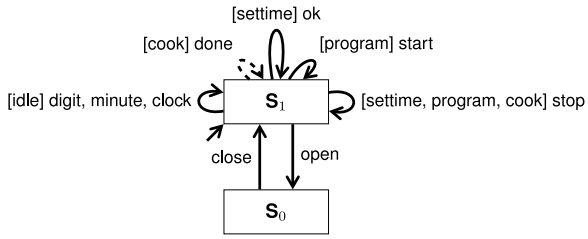


Fig. 9. Full-control mental model for the microwave oven example, considering only the opstate state-variable as visible.

A. Interaction Model and Full-Control Property

Given an HVS $\mathcal{S}$ and HVM $\mathcal{H}$, respectively, representing a system and a mental model, and given that they share the same alphabet of action and state values, their interaction is an LTS $\mathcal{I} = \mathcal{S} \parallel_I \mathcal{H}$ corresponding to the tuple $\langle S_I, \mathcal{L}^c \cup \mathcal{L}^o, s_{0_I}, \rightarrow_I\rangle$ with $S_I \subseteq (S_S \times S_H)$, $s_{0_I} = (s_{0_S}, s_{0_H})$, and $\rightarrow_I$ is defined so that:

1) $(s_{S_1}, s_{H_1}) \xrightarrow{\alpha} (s_{S_2}, s_{H_2})$ if and only if $s_{S_1} \xrightarrow{\alpha} s_{S_2}$ and $s_{H_1} \xrightarrow{[v]\alpha} s_{H_2}$ with $v = \mathcal{O}(s_{S_1})$;
2) $(s_{S_1}, s_{H_1}) \xrightarrow{\tau} (s_{S_2}, s_{H_2})$ if and only if $s_{S_1} \xrightarrow{\tau} s_{S_2}$.

The intuition is that the operator can perform a visible action (command or observation) if the state value of the system agrees with the action guard that is present on the mental model. The full-control property for enriched models is defined based on this interaction model. The difference with the full-control property for HMI-LTSs is that the set of possible commands and observations have to be considered with the associated state value.

Definition 3. Given a system model $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S, \mathcal{L}^v, \mathcal{O}\rangle$ and a mental model $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H, \mathcal{L}^v\rangle$, the mental model $\mathcal{H}$ allows full-control of the system model $\mathcal{S}$, if and only if, for all reachable $(s_S, s_H) \in \mathcal{S} \parallel_I \mathcal{H}$:

$$A^c(s_S) = A^c(s_H) \text{ and } A^o(s_S) \subseteq A^o(s_H) \tag{2}$$

and where $A^c(s) = \{(v, c) \mid \exists s \xrightarrow{\tau^*} s' \xrightarrow{c} s'' \wedge v = \mathcal{O}(s') \wedge c \in \mathcal{L}^c\}$ for HVSs and $A^c(s) = \{(v, c) \mid s \xrightarrow{[v]c} s' \wedge c \in \mathcal{L}^c\}$ for HVMs. The $A^o(s)$ sets are defined similarly.

B. Expanded Models

HVSs and HVMs can be expanded into HMI-LTSs so that the full-control property is preserved. Consequently, it is possible to check the full-control property between HVSs and HVMs using the algorithms developed for HMI-LTSs. The idea of the expansion is to move the state value that is either on the state for HVSs or on the transition for HVMs on a transition.

Fig. 10 illustrates the two mappings. The idea is similar for enriched systems and mental models. One transition of the enriched model is translated in a sequence of two transitions, by adding one intermediate state marked with the state value for HVSs or with the action guard for HVMs. The first transition of the translation corresponds to the observation of the state value or action guard and the second transition has the action that is on the original enriched transition. Note that the enriched system

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

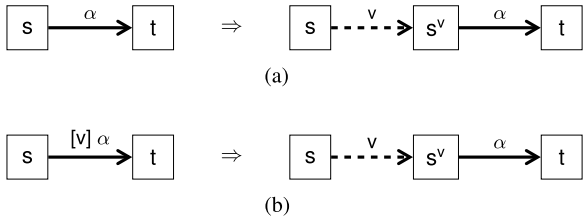


Fig. 10. Transition mapping from HVSs and HVMs enriched models to HMI-LTS, for non-τ transitions. (a) Transition from the original system model (on the left) induces two transitions in the expanded system model (on the right), where $v = \mathcal{O}(s)$. (b) Transition from the HVM (on the left) induces two transitions in the expanded HMI-LTS (on the right).

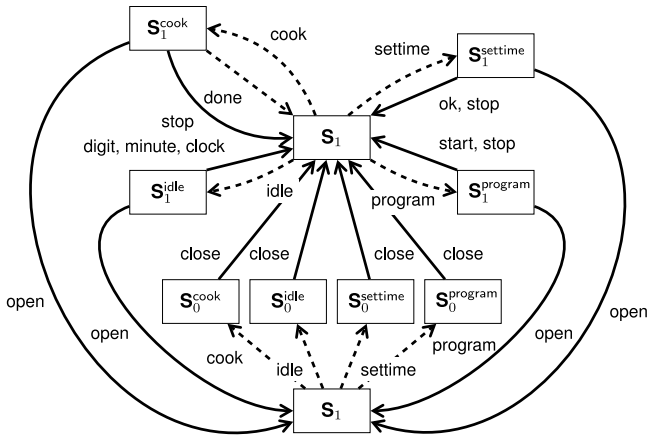


Fig. 11. Expansion of the mental model of Fig. 9.

model may contain τ-transitions; these transitions are simply kept without any change in the expanded HMI-LTS.

Both expanded models can be defined formally.

Definition 4. The expansion of an HVS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, [4] \rightarrow_S, \mathcal{L}^v, \mathcal{O}\rangle$ is an HMI-LTS $\langle S_E, \mathcal{L}^c, \mathcal{L}^o_E, s_{0_S}, \rightarrow_E\rangle$, denoted $exp(\mathcal{S})$ and where $\mathcal{L}^o_E = \mathcal{L}^o \cup \mathcal{L}^v$ and:

1) $S_E = S_S \cup \{s^v \mid s \in S_S, v = \mathcal{O}(s) \text{ and } \Gamma(s) \neq \emptyset\}$
2) $\rightarrow_E = \{(s,\tau,t) \mid (s,\tau,t) \in \rightarrow_S\} \cup \{(s,v,s^v), (s^v,\alpha,t) \mid (s,\alpha,t) \in \rightarrow_S \text{ and } v = \mathcal{O}(s)\}$.

Definition 5. The expansion of an HVM $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, [4]\ s_{0_H}, \rightarrow_H, \mathcal{L}^v\rangle$ is also an HMI-LTS $exp(\mathcal{H}) = \langle S_E, [4]\ \mathcal{L}^c, \mathcal{L}^o_E, s_{0_S}, \rightarrow_E\rangle$, where $\mathcal{L}^o_E = \mathcal{L}^o \cup \mathcal{L}^v$ and

1) $S_E = S_H \cup \{s^v \mid s \in S_H, (s,v,\alpha,t) \in \rightarrow_H\}$;
2) $\rightarrow_E = \{(s,v,s^v), (s^v,\alpha,t) \mid (s,v,\alpha,t) \in \rightarrow_H\}$.

Fig. 11 shows the expansion of the mental model of Fig. 9. The two states of the original mental model ($\mathbb{Q}_0$ and $\mathbb{Q}_1$) are still present and eight additional states have been added (one for each state value, for both states of the original model).

The following theorem, proven in [38], shows that it is possible to reduce the check of the full-control property for the enriched models to the equivalent check on the expanded HMI-LTS.

Theorem 1. Given an HVS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S, \mathcal{L}^v, \mathcal{O}\rangle$ and an HVM $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H, \mathcal{L}^v\rangle$:

$$\mathcal{H} \mathbf{fc}\ \mathcal{S} \iff exp(\mathcal{H})\ \mathbf{fc}\ exp(\mathcal{S}). \tag{3}$$

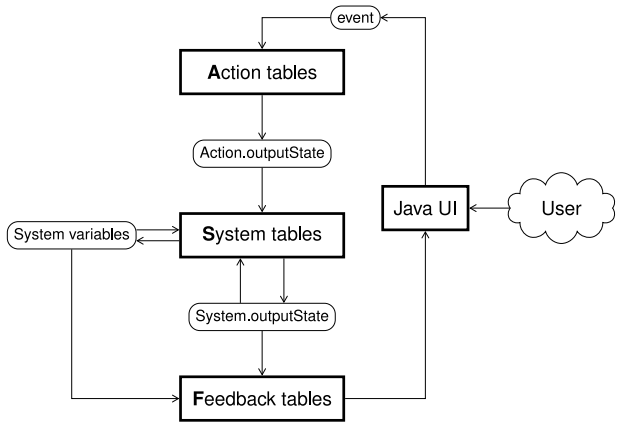


Fig. 12. ASF structure of ADEPT models, with its execution loop.

C. Automatic Design and Evaluation Prototyping Toolset Model Translation

We have considered the translation of one specific class of ADEPT models as not all elements of a complete ADEPT model are relevant for the analyses proposed by the formal framework. For example, aspects related to the GUI components are not relevant. ADEPT models following the *ASF structure* (action-system-feedback) have their logic tables split into three groups.

1) *Action tables* manage the interaction between the operator and the system, through the components of the UI. The inputs of the action tables come from the Java UI (actions from the user on interface components or automatic events triggered by timers). The outputs of the actions tables are summarized in the outputState variable.
2) *System tables* correspond to the internal decision logic of the system. The system tables take as input the outputState variables of action tables and combine them with the values of the system variables to update the state of the system. The output of system tables is also summarized in their outputState variables.
3) *Feedback tables* characterize what information is showed to the operator. That information is rendered through display components on the UI. The feedback tables take as input the outputState variables of the system tables.

Fig. 12 illustrates the ASF structure of ADEPT models and shows the execution loop of such models, along with the information that is shared between the different tables. An important assumption about the execution of the ADEPT model is that each loop through all the logic tables is supposed to be always associated with exactly one event.

For the translation of an ADEPT model following the ASF structure into an HVS, only the logic tables belonging to the system table category are considered. It is exactly those tables that contain the decision logic of the system; the two other kinds of tables play an indirect role in the translation process. The action tables are used to identify the alphabet of the HVS and the feedback tables are used to define the visible system variables that define the set of state values. An ADEPT model following the ASF structure is translated into an HVS according to the following rules.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

1) A state $s \in S$ of the HVS corresponds to a unique assignment of values to all the system variables of the ADEPT model (including the outputState variables).
2) The set of commands $\mathcal{L}^c$ is the union of the domains of all the outputState variables of the action tables, except the distinguished "no action." If that particular value is true in all the tables of the model, the transition corresponds to an internal τ-transition in the HVS.
3) The set of observations $\mathcal{L}^o$ is empty.
4) The initial state $s_0 \in S$ is based on the initial values of the system variables, as configured in the ADEPT model.
5) The transition relation $\rightarrow$ is defined according to the formal semantics proposed in this paper.
6) The set of state values $\mathcal{L}^v$ is defined by the product of domains of visible system variables. The state value of a particular state $s \in S$ corresponds to the assignment of values to the visible system variables.

VI. Evaluation

The formal analysis of ADEPT models proposed in this work has been applied to an autopilot model of a Boeing 777 aircraft. The full autopilot ADEPT model is composed of 38 logic tables. The model follows the ASF structure and has 12 action tables, 15 system tables, and nine feedback tables. The remaining tables are not considered in this work because they represent user tasks, an experimental feature of ADEPT. Three groups of tables can be identified in the model, namely one for the lateral aspect, one for the vertical aspect and one for airspeed. Tables of each group can then be classified according to the ASF structure. Action tables determine actions from UI events, system tables update the state of the system according to the performed action, and, finally, feedback tables reflect the state of the system to UI elements. Our case study focuses on the system tables.

After cleaning the system tables, the considered model has a total of 12 logic tables. There is also a total of 20 commands and no observations. The commands correspond to the manipulation of knobs, thumbwheels, buttons, and switches. There is a total of 25 system variables, some with a finite domain and the others being integer of floating point numbers.

The first lesson from our analysis of this reduced ADEPT model is that the extension from simple HMI-LTSs to enriched ones makes it possible for an automatic translation of ADEPT models, which would have been very hard to perform reliably by hand, given the size of the system. However, computing all states of the enriched HMI-LTS does not scale with the full size of such a large and complex model, in particular because the domains of the numeric variables are too large.

In order to scale, abstraction or model reduction has to be performed on the ADEPT model. Several options can be used alone, or in combination. The first possible reduction is to only consider a subset of the system tables, or to only take into account a subset of the system variables. Another reduction consists of limiting the domains of the system variables, by restricting the values to a small range. Another typical technique in formal methods is abstraction [37], which consists of replacing infinite domains of the system variables with abstract domains. One subpart of the full autopilot model that has been analyzed after model reduction, based on the first two presented approaches, was an HVS with 7680 states and 66 242 transitions, among which 57 545 are labeled with commands and 8697 are internal τ-transitions. The obtained minimal mental model has 25 states and 180 transitions. The time for model generation and analysis was 4.5 s on a 2.2-GHz Intel iCore7 MacBook Pro, with 8-GB RAM.

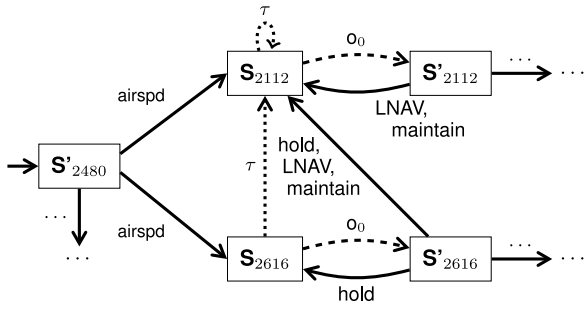


Fig. 13. Situation exhibiting the fc-determinism issue indicating a potential mode confusion situation. States S'_{2112} and S'_{2616} both also have the following commands, leading to the same blocks: lattgt, lattgt↻, lattgt↻, latHOLD, airspd, airspd↻, and airspd↻.

A. Analyzing Mode Confusion

Using the proposed formal framework, it is also possible to analyze mode confusion, which is a particular case of automation surprise. ADEPT models implicitly contain information about operating modes. In fact, the outputState system variable can be used as a mode indicator. Experiments have been performed with one of the system tables of the autopilot model (namely the airspeedSystemTable table). This experiment highlighted a potential mode confusion.

The considered subsystem has four different operating modes. After translating the ADEPT table into an HMI-LTS and running the minimal full-control mental model generation algorithm on it, an error is produced indicating that the system model is not fc-deterministic. Fig. 13 shows the part of the system model exhibiting the fc-determinism issue. By analyzing the error trace manually, it is possible to trace back the error in the ADEPT logic table. The issue is that the state S'_{2480} can lead, with the same action airspd, to two states (S_{2112} and S'_{2616}) that have different behaviors. From those two states, after having observed o_0, the sets of possible commands are not the same. In particular, from S'_{2112}, the maintain command is possible (representing the maintain mode) and from the S'_{2616} state, the maintain and hold commands are possible. Therefore, the operator is faced with potential mode confusion.

VII. Discussion

This paper addresses the integration of a formal framework that can be used to automatically detect potential automation surprises into ADEPT, a tool used by system designers to design automation interfaces. The formal framework uses HMI-LTSs, a low-level formalism based on labeled transition

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

systems, whereas the ADEPT tool uses a higher level, more intuitive description of a target system. In order to be able to go from ADEPT models to HMI-LTSs that are used by the formal framework, this paper proposes a formal semantics for ADEPT models and a systematic translation methodology for ADEPT models with the so-called ASF structure.

The semantics and translation algorithms proposed in this paper are only a first step toward effectively integrating our techniques with ADEPT. While we have worked on translating ADEPT models to HVSs, we still need to be able to directly relate the results of our analyses with the original ADEPT models. In the future, we, therefore, plan on working on a better integration of our analysis tools with ADEPT, better visualization of the analysis results, and on increasing scalability of our analysis algorithms.

A fully automated translation of any ADEPT model following the ASF structure is generally possible. However, in the current version, some basic configuration information is still provided manually after understanding the considered ADEPT model. More precisely, the steps that are still manual are the extraction of the alphabet of the HVS and the identification of the visible system variables.

For very large ADEPT models, the size of the generated HMI-LTS is expected to be too large for our approach. Moreover, for an ADEPT model with many state variables, some of these with potentially large domains, the number of states of the generated HMI-LTS and the time taken to execute the tables to compute the transition relation will also dramatically increase. This problem is even more amplified when considering system variables of floating-point type. Typical state reduction techniques such as slicing or abstraction could be used to deal with this state-explosion problem. An alternative technique would be compositional verification [37]. In this work, we reduced the ADEPT model manually, but support for this task should be provided.

REFERENCES

[1] N. B. Sarter, D. D. Woods, and C. E. Billings, "Automation surprises," in *Handbook of Human Factors & Ergonomics*, G. Salvendy, Ed. New York, NY, USA: Wiley, 1997, ch. 57, pp. 1926–1943.

[2] E. Palmer, "Oops, it didn't arm—A case study of two automation surprises," in *Proc. 8th Int. Symp. Aviation Psychol.*, Apr. 1995, pp. 227–232.

[3] N. G. Leveson, L. D. Pinnel, S. D. Sandys, S. Koga, and J. D. Reese, "Analyzing software specifications for mode confusion potential," in *Proc. Workshop Human Error Syst. Develop.*, Mar. 1997, pp. 132–146.

[4] M. Harrison and H. Thimbleby, *Formal Methods in Human-Computer Interaction*. Cambridge, U.K.: Cambridge Univ. Press, Feb. 1990.

[5] P. Palanque, Ed., *Formal Methods in Human-Computer Interaction*. New York, NY, USA: Springer, 1997.

[6] J. C. Campos and M. D. Harrison, "Formally verifying interactive systems: A review," in *Proc. 4th Int. Eurographics Workshop Design, Specification Verification Interactive Syst.*, Jun. 1997, pp. 109–124.

[7] M. Bolton, E. Bass, and R. Siminiceanu, "Using formal verification to evaluate human-automation interaction, a review," *IEEE Trans. Syst., Man, Cybern. A, Syst. Humans*, vol. 43, no. 3, pp. 488–503, May 2013.

[8] S. Combéfis, D. Giannakopoulou, C. Pecheur, and M. Feary, "A formal framework for design and analysis of human-machine interaction," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Oct. 2011, pp. 1801–1808.

[9] M. S. Feary, "A toolset for supporting iterative human–automation interaction in design," NASA Ames Res. Center, Mountain View, CA, USA, Tech. Rep. 20100012861, Mar. 2010.

[10] A. J. Dix, "Formal methods," in *The Encyclopedia of Human-Computer Interaction*, 2nd ed., M. Soegaard and R. F. Dam, Eds. Aarhus, Denmark: The Interaction Design Foundation, 2013, ch. 29.

[11] J. C. Campos, G. Doherty, and M. D. Harrison, "Analysing interactive devices based on information resource constraints," *Int. J. Human-Comput. Studies*, vol. 72, no. 3, pp. 284–297, Mar. 2014.

[12] M. Sousa, J. C. Campos, M. C. B. Alves, and M. D. Harrison, "Formal verification of safety-critical user interfaces," in *Proc. Formal Verification Model. Human-Mach. Syst. AAAI Spring Symp.*, Mar. 2014, pp. 62–67.

[13] J. C. Campos and M. D. Harrison, "Interaction engineering using the IVY tool," in *Proc. ACM SIGCHI Symp. Eng. Interactive Comput. Syst.*, Jul. 2009, pp. 35–44.

[14] H. Thimbleby, *Press On: Principles of Interaction Programming*. Cambridge, MA, USA: MIT Press, Jan. 2010.

[15] A. Gimblett and H. Thimbleby, "User interface model discovery: Towards a generic approach," in *Proc. 2nd ACM SIGCHI Symp. Eng. Interactive Comput. Syst.*, 2010, pp. 145–154.

[16] A. Gimblett and H. Thimbleby, "Applying theorem discovery to automatically find and check usability heuristics," in *Proc. 5th ACM SIGCHI Symp. Eng. Interactive Comput. Syst.*, 2013, pp. 101–106.

[17] D. Navarre, P. Palanque, and R. Bastide, "Engineering interactive systems through formal methods for both tasks and system models," in *Proc. RTO Human Factors Med. Panel Spec. Meeting*, Jun. 2001, pp. 20.1–20.17.

[18] R. Bastide, D. Navarre, and P. Palanque, "A tool-supported design framework for safety critical interactive systems," *Interacting Comput.*, vol. 15, no. 3, pp. 309–328, Jun. 2003.

[19] D. Navarre, P. Palanque, J.-F. Ladry, and E. Barboni, "ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability," *ACM Trans. Comput.-Human Interaction*, vol. 16, no. 4, pp. 18:1–18:56, Nov. 2009.

[20] S. Combéfis, "Operational model: Integrating user tasks and environment information with system model," in *Proc. 3rd Int. Workshop Formal Methods Interactive Syst.*, Nov. 2009, pp. 83–86.

[21] M. Bolton, R. Siminiceanu, and E. Bass, "A systematic approach to model checking human-automation interaction using task analytic models," *IEEE Trans. Syst., Man, Cybern. A, Syst. Humans*, vol. 41, no. 5, pp. 961–976, Sep. 2011.

[22] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, "Generating phenotypical erroneous human behavior to evaluate human-automation interaction using model checking," *Int. Human-Comput. Stud.*, vol. 70, no. 11, pp. 888–906, May 2012.

[23] M. Bolton and E. Bass, "Generating erroneous human behavior from the strategic knowledge in task analytic models and evaluating its impact on system safety with model checking," *IEEE Trans. Syst., Man, Cybern. A, Syst. Humans*, vol. 43, no. 3, pp. 1314–1327, May 2013.

[24] E. J. Bass, K. M. Feigh, E. Gunter, and J. Rushby, "Formal modeling and analysis for interactive hybrid systems," in *Proc. 4th Int. Workshop Formal Methods Interactive Syst.*, Jun. 2011, pp. 1–16.

[25] G. Gelman, K. M. Feigh, and J. Rushby, "Example of a complementary use of model checking and human performance simulation," *IEEE Trans. Human-Mach. Syst.*, vol. 44, no. 5, pp. 576–590, Oct. 2014.

[26] J. Bredereke and A. Lankenau, "A rigorous view of mode confusion," in *Proc. 21st Int. Conf. Comput. Safety, Rel. Security*, Sep. 2002, pp. 19–31.

[27] J. Bredereke and A. Lankenau, "Safety-relevant mode confusions—Modelling and reducing them," *Rel. Eng. Syst. Safety*, vol. 88, no. 3, pp. 229–245, Jun. 2005.

[28] A. Degani and M. Heymann, "Formal verification of human-automation interaction," *J. Human Factors Ergonomics Soc.*, vol. 44, no. 1, pp. 28–43, Spring 2002.

[29] M. Heymann and A. Degani, "On the construction of human-automation interfaces by formal abstraction," in *Proc. 5th Int. Symp. Abstr., Reformulation Approx.*, Aug. 2002, pp. 99–115.

[30] M. Heymann and A. Degani, "Formal analysis and automatic generation of user interfaces: Approach, methodology, and an algorithm," *Human Factors: J. Human Factors Ergonomics Soc.*, vol. 49, no. 2, pp. 311–330, Apr. 2007.

[31] S. Shiffman, A. Degani, and M. Heymann, "Uiverify—A web-based tool for verification and automatic generation of user interfaces," in *Proc. 8th Annu. Appl. Ergonomics Conf.*, Mar. 2005, pp. 8–11.

[32] D. Javaux, "A method for predicting errors when interacting with finite state systems. How implicit learning shapes the user's knowledge of a system," *Rel. Eng. Syst. Safety*, vol. 75, pp. 147–165, Feb. 2002.

[33] S. Combéfis and C. Pecheur, "A bisimulation-based approach to the analysis of human-computer interaction," in *Proc. ACM SIGCHI Symp. Eng. Interactive Comput. Syst.*, Jul. 2009, pp. 101–110.

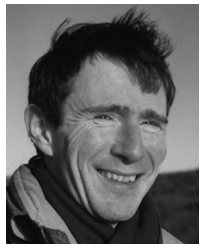
This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

[34] S. Combéfis, D. Giannakopoulou, C. Pecheur, and M. Feary, "Learning system abstractions for human operators," in *Proc. Int. Workshop Mach. Learning Technol. Softw. Eng.*, Nov. 2011, pp. 3–10.
[35] M. Feary, "Automatic detection of interaction vulnerabilities in an executable specification," in *Proc. 7th Int. Conf. Eng. Psychol. Cognitive Ergonomics*, Jul. 2007, pp. 487–496.
[36] S. Combéfis, D. Giannakopoulou, and C. Pecheur, "State event models for the formal analysis of human-machine interactions," in *Proc. Formal Verification Model. Human-Mach. Syst. AAAI Spring Symp.*, Mar. 2014, pp. 15–20.
[37] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: The MIT Press, Jan. 1999.
[38] S. Combéfis, "A formal framework for the analysis of human-machine interactions," Ph.D. dissertation, Univ. Catholique Louvain, Louvain-la-Neuve, Belgium, 2013.

Sébastien Combéfis (M'09) received the Ph.D. degree in engineering from the Université catholique de Louvain, Louvain-la-Neuve, Belgium, in November 2013.

He is currently a Lecturer in computer science with the École Centrale des Arts et Métiers. He worked under the supervision of Prof. C. Pecheur on a thesis entitled "A Formal Framework for the Analysis of Human-Machine Interactions." His research interests include the application of formal method-based techniques for the analysis of various problems in the human–machine interaction field.

Dimitra Giannakopoulou received the Ph.D. degree in distributed computing from Imperial College, University of London, London, U.K. She is currently a Research Scientist with the NASA Ames Research Center, Mountain View, CA, USA, since 2000. As a member of the Robust Software Engineering group, her research focuses on scalable software specification and verification techniques.



Charles Pecheur (M'86) received the Ph.D. degree from the Université de Liège, Liège, Belgium, in 1996. He has been a Professor with the Université Catholique de Louvain, Louvain-la-Neuve, Belgium, since 2004. His research focuses on theory, algorithms, and tools for the design, analysis, and validation of reactive computer systems, including fault analysis, human–computer interaction, and other aspects related to partial observability of a system.

Dr. Pecheur is a Member of the ACM.