Learning System Abstractions for Human Operators

Sébastien Combéfis*, Dimitra Giannakopoulou†, Charles Pecheur*, Michael Feary†

*Computer Science and Engineering Department
ICT, Electronics and Applied Mathematics Institute
Université catholique de Louvain, Belgium
{Sebastien.Combefis,Charles.Pecheur}@uclouvain.be

†NASA Ames Research Center
Moffett Field, CA 94035, USA
{Dimitra.Giannakopoulou,Michael.S.Feary}@nasa.gov

ABSTRACT

This paper is concerned with the use of formal techniques for the analysis of human-machine interactions (HMI). The focus is on generating system abstractions for human operators. Such abstractions, once expressed in rigorous, formal notations, can be used for analysis or for user training. They should ideally be *minimal* in order to concisely capture the system behaviour. They should also contain enough information to allow *full-control* of the system.

This work addresses the problem of automatically generating abstractions, based on formal descriptions of system behaviour. Previous work presented a bisimulation-based technique for constructing minimal full-control abstractions. This paper proposes an alternative approach based on the use of the L^* learning algorithm. In particular, minimal abstractions are generated from learned three-valued deterministic finite-state automata. The learning-based approach is applied on a number of examples and compared to the bisimulation-based approach. The result of these comparisons is that there is no clear winner. However, the proposed approach has wider applicability since it can handle more types of systems than the bisimulation-based technique. Moreover, if no full-control abstraction can be generated due to a form of non-determinism in the system, the learning-based approach provides counterexamples that allow to detect and analyze that non-determinism. We also discuss how the well-known HMI issue of mode confusion can be analyzed through this approach.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods; I.2.6 [**Artificial Intelligence**]: Learning; H.1.2 [**Models and Principles**]: User/Machine Systems—*Human factors*

General Terms

Verification, Human Factors, Algorithms

Keywords

Formal methods, Learning, Human-Machine Interaction (HMI), Verification, 3DFA, Model-checking

1. INTRODUCTION

Most complex computer systems involve some amount of interaction between humans and machines. Ensuring correct perception, understanding and control of the machine by its human operators is an important part of the safety requirements of a system. There are numerous examples of failures caused by an inappropriate interaction between the operator and the machine. A well-known class of problems is known as *automation surprises*, that occur when the system behaves differently than its operator expects. For example, if a car driver unknowingly engages the cruise-control system, she could be surprised by the car's behaviour, leading into hazardous situations. Automation surprises can lead to *mode confusion* [14, 21] and sometimes to critical failure, as testified by real accidents [6, 15, 17].

This paper reports on collaborative work between the Human Factors and Robust Software Engineering (RSE) groups at the NASA Ames Research Center on the use of formal techniques for the analysis of HMI systems. In particular, we address here the problem of automatic generation of system abstractions for human operators. Such abstractions, once expressed in rigorous, formal notations, can be used for analysis or for user training. System abstractions should concisely capture the behaviour of the system from the point of view of an operator. They should also contain enough information to allow proper control of the system, that is the operator can control the system by knowing what he can perform on the system and what he can expect to observe. This problem has already been stated and studied in [5] using bisimulation-based relations. A suitable similarity relation is defined on the states of the system model. Provided that some determinacy restrictions are met, a variant of the Paige-Tarjan refinement algorithm [16] can be used to merge those equivalent states and get a suitable abstraction as a minimal abstraction of the system.

In this paper, we investigate an alternative approach for addressing the same problem: a variant of the L^* learning algorithm is used to construct the abstraction. The framework is implemented on top of the Java Pathfinder model-checker [23], building on previous work in [9] where a similar approach is used to infer component interfaces. However, the setting of HMI systems presents some new challenges as compared to interface generation. These challenges are described in Section 2, where we also explain why existing techniques are not satisfactory for abstraction of HMI systems. At a high level, the proposed framework first uses L^* to build a 3DFA, that is, a deterministic finite automaton with accepting, refusing and "don't-care" states. Sequences to don't-care states correspond to situations that are not needed for properly operating the system but may be included in the abstraction without compromising operation. They allow to obtain a simpler, smaller abstraction at the end, by merging together compatible sequences. The framework subsequently minimizes the 3DFA into a standard transition system, which is the desired minimal abstraction.

Moreover, the partially built model is still usable in cases where the algorithm fails due to resource limitations or inadequate system models. The contributions of this work can be summarized as follows. First, we provide a learning framework for abstraction generation; this consists of defining a teacher to be used in a variant of the L^* algorithm to learn 3DFAs, which characterize the set of possible abstractions satisfying the full-control property, this is the proper control criterion used in this work. Second, we use our implementation of the framework in the Java Pathfinder model checker to provide evaluation results on a number of examples: some benchmarks from previous work, and some examples provided by our NASA human factors collaborators. Third, we show how failed abstraction generation provides counter-examples that support detecting and analyzing violations of required determinism properties.

The remainder of this paper is organized as follows. Sections 2 and 3 provide related work, motivation and background. Section 4 is the core of the paper and explains the proposed learning-based framework. Section 5 describes the prototype implementation and discusses the results of the experiments. Finally, Section 6 concludes the paper with discussion and plans for future work.

2. RELATED WORK AND MOTIVATION

Analysis of human-machine interaction (HMI) is a field that has been studied extensively by researchers in psychology, human factors and ergonomics, but formal methods can also contribute to the analysis and design of the behavioural aspects of HMI. Indeed, since the mid-1980s, several researchers have investigated the application of formal methods to HMI analysis, but most of the work so far has focused on specific target applications or on the system and its properties [20, 3, 22]. More recently, Heymann and Degani [11] pioneered a more generic automata-based approach for checking and generating adequate formal abstractions of a user's knowledge about a given system. Their work distinguishes between commands, observations and internal actions, and their abstraction algorithm is based on the definition of compatible system states and merger tables. Our terminology and general framework are based on the work of Heymann and Degani.

In [5], Combéfis and Pecheur extended that work by proposing a formal definition of *full-control* to characterize good system's abstractions, using labelled transition systems (LTSs) as formal models. This definition induces a similarity relation on the states of the system, such that similar states can be confounded in the abstraction. When that relation is an equivalence for a specific system, they develop a minimization algorithm for calculating the quotient of the system modulo the relation. However, the relation may fail to be transitive, so their algorithm is not applicable to some models as illustrated in Figure 5. The proposed approach overcomes the latter problem and will be able to produce a minimal model for any LTS of an HMI system.

Abstraction generation is related to the problem of component interface generation. A component interface summarizes all possible correct usages of the component [10]. A large body of research has used learning for interface generation [9, 1]. In this work, we therefore decided to investigate and evaluate the applicability of learning for abstraction generation. The distinction between the two domains is that, for interface generation, one does not typically differentiate between controllable and observable actions. Moreover, the notion of a precise interface is defined as a safe and permissive one, which is different from the notion of "full-control" associated with abstractions. Note that the notion of controllable and observable actions is similar to a notion of inputs and outputs, respectively. A distinction between input and output actions of a system is made in the learning-based approach for interface automata of Emmi et al. [7], but that work is performed in the context of showing, in a compositional way, component compatibility.

The framework that we present in this paper has a fundamental difference from all of the above frameworks: the notion of full-control allows to *optionally* accept some sequences in the learned language. In our work, we investigated whether it would be possible to apply the framework presented in [9] on LTSs of system models modified to capture the distinction between observable and controllable actions in the full-control property. Observations may or may not be added, as driven by the need to merge states that are equivalent with respect to the full-control property. In fact, the correct interpretation for missing observations is a "don't care" one, where the decision on whether to add them or not is driven by the needs of minimization. This motivated us to look into an alternative version of L^*, which learns 3DFAs, rather than simple DFAs [4]. However, Chen et al. [4] defined their algorithm in the context of computing minimal assumptions for compositional verification, as opposed to abstraction generation for HMI systems, which will be presented here.

We would like to stress that, although learning frameworks have a similar overall structure, the challenge for any specific problem is in the definition of an appropriate Teacher to represent the language that is being learned. None of the existing learning frameworks that we are aware of could have been used for generating HMI abstractions.

3. BACKGROUND

We use the example of a semi-automatic vehicle transmission system (VTS) to illustrate some of the concepts in this section. The example is taken from [11]. The model of the system is shown on Figure 1. The system has eight states and two kinds of actions. The initial state is low-1. The actions push-up and pull-down are triggered when the user operates the transmission lever. The actions up and down are triggered autonomously by the system, without the intervention of the user, and corresponds to automatic internal gear shifting as speed changes. The user can just hear the occurrence of those two last actions. Note that this system has a particular behaviour: while the system is in the low level (low-1, low-2 or low-3), if the user triggers a push-up action, the system can either transition to the medium level or high level depending on the current low state the system is in.

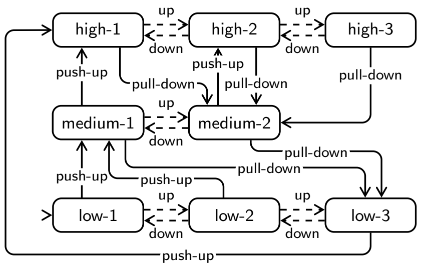


Figure 1: Vehicle transmission system (VTS): the system model.

3.1 Labelled Transition Systems

Both system models and abstractions are formally represented with an enriched version of labelled transition system. *Act* is the universal set of actions, τ is the unobservable action, and Π is a special state which denotes an error. An HMI LTS $\mathcal{M}$ is a tuple $\mathcal{M} = \langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \delta\rangle$ where S is the finite set of states and $s_0 \in S$ is the initial state. The set $\mathcal{L}^c$ of *commands* contains the actions that are controlled by the user, and the set $\mathcal{L}^o$ of *observations* contains actions controlled by the system but observed by the user. We use $\mathcal{L}^{co} = \mathcal{L}^c \uplus \mathcal{L}^o$ to denote the actions that are visible to the operator, i.e., commands and observations. All unobservable and uncontrollable actions are represented by τ. The transition function is $\delta : S \times (\mathcal{L}^{co} \cup \{\tau\}) \to 2^S$. We say that $\mathcal{M}$ is *deterministic* if it contains no τ-transitions and if $\delta(s, a)$ contains at most one element.

Figure 1 shows the graphical representation of the HMI LTS of the VTS system model. Commands are depicted as solid lines $\longrightarrow$ and observations as dashed lines $\dashrightarrow$.

The notation $s \xrightarrow{\alpha} s'$ is a shortcut for $s' \in \delta(s, \alpha)$ and corresponds to a strong transition. It is extended for a sequence $\sigma = \alpha_1\alpha_2 \dots \alpha_n \in \mathcal{L}^*$ in the usual way, that is $s \xrightarrow{\sigma} s'$ is a shortcut for $s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s'$. The notation $s \overset{\alpha}{\Longrightarrow} s'$ represents a weak transition and is a shortcut for $s \xrightarrow{\tau^*\alpha\tau^*} s'$, that is the transition α can be preceded and followed by zero or more τ-transitions. The $\Rightarrow$ notation is extended for a sequence in a similar way.

The set of all sequences belonging to a model is denoted $\mathbf{Tr}(\mathcal{M})$ and defined as $\{\sigma \in \mathcal{L}^{co*} \mid \exists s_0 \overset{\sigma}{\Longrightarrow} s'\}$. Given a state $s \in S$, the set of *enabled* commands (resp. observations), denoted $A^c(s)$ (resp. $A^o(s)$) is defined by $\{\alpha \in \mathcal{L}^c \mid s \overset{\alpha}{\Longrightarrow} s'\}$ (resp. $\{\alpha \in \mathcal{L}^o \mid s \overset{\alpha}{\Longrightarrow} s'\}$).

In this work, system models and abstractions are represented as HMI LTSs $\mathcal{M}_M$ and $\mathcal{M}_U$ which are defined on the same alphabet $\mathcal{L}^c, \mathcal{L}^o$. Moreover, abstractions are deterministic HMI LTSs without τ transitions, so that $s_{U_0} \overset{\alpha}{\Longrightarrow} s'$ reduces to $s_{U_0} \xrightarrow{\alpha} s'$ in $\mathcal{M}_U$.

The possible interactions between a system $\mathcal{M}_M$ and a user following a deterministic abstraction $\mathcal{M}_U$ for that system are represented by the *parallel composition* between the models and denoted by $\mathcal{M}_M \parallel \mathcal{M}_U$. States of $\mathcal{M}_M \parallel \mathcal{M}_U$ are pairs of states $(s_M, s_U) \in S_M \times S_U$; in particular, the initial state is (s_{0_M}, s_{0_U}). There is a transition $(s_M, s_U) \xrightarrow{\alpha} (s'_M, s'_U)$ with $\alpha \in \mathcal{L}^{co}$ if there is both $s_M \xrightarrow{\alpha} s'_M$ and $s_U \xrightarrow{\alpha} s'_U$, and there is a transition $(s_M, s_U) \xrightarrow{\tau} (s'_M, s_U)$ if there is $s_M \xrightarrow{\tau} s'_M$.

For simplicity, in the remainder of the paper, an HMI LTS will simply be referred to as LTS.

3.2 Full-Control Abstraction

In this work, both systems and their associated abstractions are represented with LTS. As formalized by some of the authors in previous work [5], a desirable property for an abstraction is that of full-control, defined as follows. An abstraction $\mathcal{M}_U$ allows *full-control* of a system $\mathcal{M}_M$ iff:

$$\forall \sigma \in \mathcal{L}^{co*} \text{ such that } s_{0_M} \overset{\sigma}{\Longrightarrow} s_M \text{ and } s_{0_U} \xrightarrow{\sigma} s_U : \\ A^c(s_M) = A^c(s_U) \quad \wedge \quad A^o(s_M) \subseteq A^o(s_U). \qquad (1)$$

Intuitively, a full-control abstraction allows a user that follows it to know exactly what commands can be executed in the current state of the system, and to be prepared to receive at least any observation that the system may produce, and possibly others too.

Achieving full-control is only possible if the user can know the allowed commands after any sequence σ. Otherwise, the user has no way to know whether a command is available or not. A system is *full-control-deterministic* iff:

$$\forall \sigma \in \mathcal{L}^{co*} \text{ such that } s_{0_M} \overset{\sigma}{\Longrightarrow} s_M \text{ and } s_{0_M} \overset{\sigma}{\Longrightarrow} s'_M : \\ A^c(s_M) = A^c(s'_M) \qquad (2)$$

Full-control-non-determinism is detected and reported during the generation of abstractions (see Section 4.4).

Figure 2 shows the minimal full-control abstraction for the vehicle transmission system example. It is easily seen that for every composite state (s_M, s_U) of the parallel composition between the models of Figures 1 and 2, the full-control conditions $A^c(S_M) = A^c(s_U)$ and $A^o(s_M) \subseteq A^o(s_U)$ are indeed satisfied.

The full-control requirement allows an abstraction to have more observations than those possible on the system. To capture and represent such optional behaviour, we use Three-Valued Deterministic Finite Automata (3DFAs).

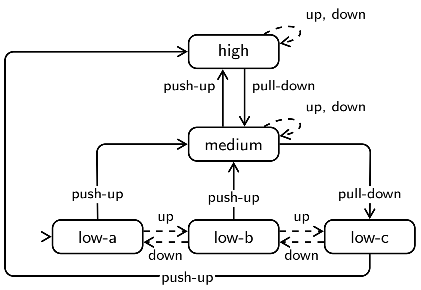


Figure 2: VTS example: the minimal full-control abstraction.

3.3 DFAs and Three-Valued DFAs (3DFAs)

A *Deterministic Finite Automaton* (DFA) $\mathcal{A}$ is a tuple $\langle \Sigma, S, s_0, \delta, Acc\rangle$, where Σ is an alphabet, S is the finite set of states, s_0 is the initial state, $\delta : S \times \Sigma \to S$ is the transition function, and $Acc \subseteq S$ is a set of *accepting states*. The transition function is extended in the usual way to sequences, so that for $\sigma = \sigma_0 \cdots \sigma_n \in \Sigma^*$, $\delta(s_0, \sigma) = \delta(\ldots \delta(\delta(s_0, \sigma_0), \sigma_1) \ldots, \sigma_n)$. A sequence $\sigma \in \Sigma^*$ is accepted by the automaton if and only if $\delta(s_0, \sigma) \in Acc$.

A *Three-Valued Deterministic Finite Automaton* (3DFA) $\mathcal{C}$ is a tuple $\langle \Sigma, S, s_0, \delta, Acc, Rej, Dont \rangle$, where Σ, S, s_0, δ are as defined in a DFA. However, S is partitioned into three disjoint sets *Acc* (accepting states), *Rej* (rejecting states), and *Dont* (don't care states). Given a 3DFA $\mathcal{C} = \langle \Sigma, S, s_0, \delta, Acc, Rej, Dont \rangle$, a sequence $\sigma \in \Sigma^*$ is accepted if $\delta(s_0, \sigma) \in Acc$, rejected if $\delta(s_0, \sigma) \in Rej$, and is a don't care sequence if $\delta(s_0, \sigma) \in Dont$.

Let $\mathcal{C}^+$ denote the DFA $\langle \Sigma, S, s_0, \delta, Acc \cup Dont \rangle$, where all don't care states become accepting states, and $\mathcal{C}^-$ denote the DFA $\langle \Sigma, S, s_0, \delta, Acc \rangle$, where all don't care states become rejecting states. By definition, we have that $\mathcal{L}(\mathcal{C}^-)$ is the set of accepted sequences in $\mathcal{C}$ and $\overline{\mathcal{L}(\mathcal{C}^+)}$ is the set of rejected sequences in $\mathcal{C}$.

A DFA $\mathcal{A}$ is consistent with a 3DFA $\mathcal{C}$ if and only if $\mathcal{A}$ accepts all sequences that $\mathcal{C}$ accepts, and rejects all sequences that $\mathcal{C}$ rejects. It follows that $\mathcal{A}$ accepts sequences in $\mathcal{L}(\mathcal{C}^-)$ and rejects those in $\overline{\mathcal{L}(\mathcal{C}^+)}$, or equivalently, $\mathcal{L}(\mathcal{C}^-) \subseteq \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{C}^+)$. A DFA $\mathcal{A}$ is the minimal consistent DFA of a 3DFA $\mathcal{C}$ if it is consistent with $\mathcal{C}$ and has the smallest number of states among all DFAs which are consistent with $\mathcal{C}$.

3.4 The L^* Learning Algorithm

The learning algorithm L^* of Angluin [2] learns an unknown regular language and produces a DFA that accepts it. Let U be an unknown regular language over some alphabet Σ. In order to learn U, L^* interacts with a *Teacher* that must answer correctly two types of questions. The first type is a *membership query*, consisting of a sequence $\sigma \in \Sigma^*$; the answer is *true* if $\sigma \in U$, and *false* otherwise. The second type is a *conjecture*, that is, a candidate DFA C whose language $\mathcal{L}(C)$ the algorithm believes to be identical to U. The answer is *true* if $\mathcal{L}(C) = U$. Otherwise the Teacher returns a counterexample, which is a sequence σ in the symmetric difference of $\mathcal{L}(C)$ and U. Let M be the minimal (in terms of number of states) automaton such that $\mathcal{L}(M) = U$. L^* is guaranteed to terminate with M as its last conjecture. The conjectures made by L^* strictly increase in size; each conjecture is smaller than the next one, and all incorrect conjectures are smaller than M.

4. LEARNING FRAMEWORK FOR FULL-CONTROL ABSTRACTION GENERATION

Similarly to Chen et al [4], we use L^* to learn 3DFAs, but in the context of abstraction generation for HMI systems. The challenge therefore lies in providing a correct Teacher that captures the notion of full-control. Our framework uses L^* for 3DFAs to learn a minimal full-control abstraction for a system model $\mathcal{M}$. The high-level structure of the framework is presented in Figure 3. The framework first uses L^* to learn a minimal 3DFA $\mathcal{C}$, with $\Sigma_{\mathcal{C}} = \mathcal{L}^{co}$. $\mathcal{C}$ must exhibit the full-control property, meaning that any DFA consistent with $\mathcal{C}$ is a full-control abstraction for $\mathcal{M}$. The framework subsequently generates a minimal DFA consistent with the 3DFA that was produced by L^*.

In what follows, we present our implementation of a Teacher for L^*. Sections 4.1 and 4.2 presents the two parts of a Teacher (membership query and conjecture) and provide brief justifications about their correctness which induces the correctness of the Teacher. We also discuss the minimization phase of the framework, as well as complexity and overall correctness issues. Note that the learning always succeeds when the system model is full-control deterministic. At the end of this section, we also discuss the case where the system is not full-control deterministic.

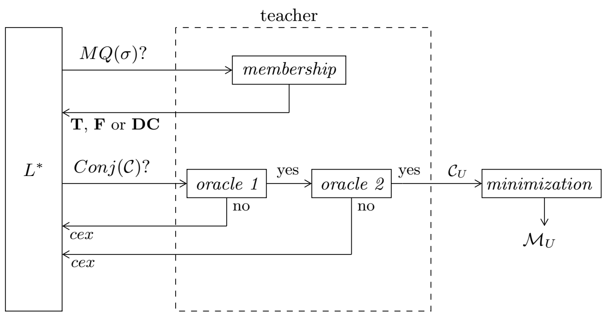


Figure 3: Global view of the proposed learning-based approach to generate a full-control abstraction from a given system model.

The notion of correctness for HMI system abstractions is captured by the full-control property as presented in Section 3.2. The definition of full-control involves weak transitions in the model, and therefore our framework Teacher uses $\mathcal{M}_{\text{weak}}$, which is obtained from $\mathcal{M}$ by transforming its transition relation into the corresponding weak transition relation. This is performed with the standard τ^*-completion construction, which computes the reflexive and transitive closure with respect to the τ relation [12].

Moreover, let $\mathcal{A} = \langle \Sigma, S, s_0, \delta, Acc \rangle$ be a DFA such that $\Sigma_{\mathcal{A}} = \mathcal{L}^{co}$. We define $lts(\mathcal{A}) = \langle Acc, \mathcal{L}^c, \mathcal{L}^o, s_0, \delta' \rangle$, where δ' is obtained from projecting δ onto the accepting states *Acc*. In other words, $lts(\mathcal{A})$ is a deterministic LTS obtained by removing the rejecting states and their corresponding transitions from the DFA. This transformation is well defined because the DFAs that are generated by our approach are always prefix-closed, meaning that it is not possible to reach an accepting state after a rejecting state has been reached.

4.1 Membership query

A membership query (MQ) determines whether a sequence $\sigma \in \mathcal{L}^{co*}$ should be an accepting, rejecting, or don't care sequence in the learned 3DFA $\mathcal{C}$. There are therefore three possible outcomes to such queries: yes (**T**), no (**F**) and don't care (**DC**). Membership queries are answered based on the full-control property requirement.

Our membership query algorithm operates on $\mathcal{M}_{\text{weak}}$, completed on commands by adding transitions leading to an error state Π. That is for each state s, a transition $s \xrightarrow{\alpha} \Pi$ is added for each $\alpha \in \mathcal{L}^c \setminus A^c(s)$. The sequence σ is then simulated on the completed system and there are different possible outcomes giving rise to three different answers:

1. σ *may* lead to the error state: $MQ(\sigma) = \mathbf{F}$;
2. σ can be simulated entirely and *never* leads to an error state: $MQ(\sigma) = \mathbf{T}$;
3. σ cannot be simulated entirely: $MQ(\sigma) = \mathbf{DC}$.

Notice that for any sequence σ such that $MQ(\sigma) = \mathbf{F}$ (resp. $\mathbf{DC}$), it always follows that $MQ(\sigma\sigma') = \mathbf{F}$ (resp. $\mathbf{DC}$) for all sequences σ'. That property is used in the implementation to speed up the membership queries algorithm by using a memoized table that stores the results of all previously queried sequences.

Intuition and Justification. **Case 1:** Since the error state is reached during simulation of σ, it means that there exists a sequence σ' and a command c such that $\sigma'c$ is a prefix of σ which leads to the error. Therefore, c is not available after σ' in the system model. The full-control property requires that c not be available after σ' in the learned 3DFA either. As mentioned above, if a sequence is not accepted by the language that we are trying to learn, then any extension of that sequence will not be accepted either. It follows that any extension of sequence $\sigma'c$ is not accepted either, and therefore the answer to query σ must be $\mathbf{F}$. For example, $MQ(\textsf{pull-down}) = \mathbf{F}$, because $\textsf{low-1} \xrightarrow{\textsf{pull-down}} \Pi$.

Case 2: Let σ be $\sigma'a$, where a is either an observation or a command. The full-control property requires for a to be available after σ' in the full-control model (since it requires equality of commands and a superset for observations), so the answer must be $\mathbf{T}$. For example, $MQ(\textsf{push-up}) = \mathbf{T}$, because the sequence exists and never leads to error.

Case 3: Since σ cannot be simulated entirely and it does not lead to the error state, it must block on some observation (since the system model is completed with respect to commands). Therefore, there exists a sequence σ' and an observation o such that $\sigma'o$ is a prefix of σ, σ' belongs to $\mathbf{Tr}(\mathcal{M})$ and o is not available after σ' in the system model. Such observations are optional in the abstraction according to the full-control property, which explains the $\mathbf{DC}$ answer. For example, $MQ(\textsf{push-up}, \textsf{down}) = \mathbf{DC}$, because $MQ(\textsf{push-up}) = \mathbf{T}$ and $\textsf{down}$ is not possible on the states reached after executing $\langle\textsf{push-up}\rangle$. $\square$

4.2 Conjectures

A Conjecture (*Conj*) establishes whether a candidate 3DFA $\mathcal{C}$ has the full-control property, meaning that *any* DFA that is consistent with the conjectured 3DFA has the full-control property. The algorithm may reply with a *yes*, in which case learning terminates, or with a *no*, in which case a counterexample *cex* is provided that exhibits the fact that the conjectured 3DFA does not provide full-control. In other words, *some* consistent DFA does not have the full-control property. Checking this property on $\mathcal{C}$ is established in two steps, represented by Oracle 1 and Oracle 2.

Note that in this work, we omit a completeness check for the 3DFA defined in Chen et al [4]. The completeness check requires a potentially expensive determinization of the system model. By omitting this check, we may miss smaller abstractions, but this case did not occur in our case studies. Chen et al. also omit this check in their experiments.

Oracle 1 operates on $\mathcal{M}_{\text{weak}}$ and $\mathcal{C}^+$. It first completes $\mathcal{M}_{\text{weak}}$ with respect to commands by adding transitions leading to the error state Π: for each state s, a transition $s \xrightarrow{\alpha} \Pi$ is added for each $\alpha \in \mathcal{L}^c \setminus A^c(s)$. It then computes the parallel composition of the resulting LTS with $lts(\mathcal{C}^+)$. If the error state Π is not reachable in the composition, then the Oracle 1 check passes, and Oracle 2 is invoked. Otherwise, the answer is *no*, and the counterexample *cex* obtained is returned to L^*, which will start a new iteration of membership queries in order to produce a refined conjecture.

Intuition and Justification. Oracle 1 establishes whether $\forall \mathcal{C}_i$ such that $\mathcal{C}_i$ is consistent with $\mathcal{C}$ the following holds: $\forall \sigma \in \mathcal{L}^{co*}$ such that $s_{0_M} \overset{\sigma}{\Longrightarrow} s_M, s_{0_{\mathcal{C}_i}} \xrightarrow{\sigma} s_{\mathcal{C}_i}$ and $s_{\mathcal{C}_i} \in Acc_{\mathcal{C}_i} : A^c(s_M) \supseteq A^c(s_{\mathcal{C}_i})$.

In other words, after any string, the set of commands available in the system model should be a superset of the set of commands available in the abstraction. We use $\mathcal{C}^+$ to represent all consistent DFAs of $\mathcal{C}$ for this check because $\mathcal{C}^+$ accepts the largest language among them. When Π is reachable in the parallel composition of $lts(\mathcal{C}^+)$ with $\mathcal{M}_{\text{weak}}$ completed with Π on commands, the obtained counterexample exposes a sequence σ and a command c where σ can be executed in both models, but after σ, command c is enabled in the abstraction but not enabled in the system model (hence it leads to Π). Therefore, the conjectured 3DFA represents at least one consistent DFA ($\mathcal{C}^+$) that is not a full-control model of the system. Since the system model is complete with respect to commands, there is no way of missing any counterexamples that are relevant to Oracle 1. $\square$

Oracle 2 operates on $\mathcal{M}_{\text{weak}}$ and $\mathcal{C}^-$. It first completes $lts(\mathcal{C}^-)$ with respect to all observable actions (both commands and observations) so that any missing transitions are replaced with transitions to the error state Π. It then computes the parallel composition of $\mathcal{M}_{\text{weak}}$ with the completed $lts(\mathcal{C}^-)$. If state Π is unreachable in the composition, then the answer is *yes*, and L^* concludes producing $\mathcal{C}$ as a representative of all full-control abstractions for $\mathcal{M}$. Otherwise, the answer is *no*, and a counterexample *cex* is produced, which exhibits the fact that the candidate $\mathcal{C}$ represents some abstractions that are missing transitions on some command or some observation (the last action in *cex*). Based on *cex*, L^* starts a new iteration of membership queries in order to produce a refined conjecture.

Intuition and Justification. Oracle 2 establishes whether $\forall \mathcal{C}_i$ such that $\mathcal{C}_i$ is consistent with $\mathcal{C}$ the following holds: $\forall \sigma \in \mathcal{L}^{co*}$ such that $s_{0_M} \overset{\sigma}{\Longrightarrow} s_M, s_{0_{\mathcal{C}_i}} \xrightarrow{\sigma} s_{\mathcal{C}_i}$ and $s_{\mathcal{C}_i} \in Acc_{\mathcal{C}_i} : A^c(s_M) \subseteq A^c(s_{\mathcal{C}_i}) \wedge A^o(s_M) \subseteq A^o(s_{\mathcal{C}_i})$.

In other words, after any string, the set of commands available in the system model should be a subset of the set of commands available in the abstraction, and the same should hold for observations. We use $\mathcal{C}^-$ to represent all consistent DFAs of $\mathcal{C}$ for this check because $\mathcal{C}^-$ accepts the smallest language among them. When the error state of the completed $lts(\mathcal{C}^-)$ is reachable in the parallel composition of $lts(\mathcal{C}^-)$ with $\mathcal{M}_{\text{weak}}$, the obtained counterexample exposes a sequence σ and an observable action *obs* (*obs* may be a command or an observation) such that, after σ, action *obs* is enabled in the system model but not enabled in the abstraction. Since $lts(\mathcal{C}^-)$ is complete, there is no way of missing counterexamples that are relevant to Oracle 2. $\square$

Figure 4 shows a 3DFA which is an intermediate candidate produced by the learning algorithm. The candidate is checked by the two oracles and fails during Oracle 2 with the following counterexample: $\langle\textsf{up}, \textsf{up}, \textsf{down}, \textsf{down}\rangle$. Although this sequence is a trace of the system, in the 3DFA, it leads to a don't care state (state 1), which corresponds to a non-accepting state in $\mathcal{C}^-$ and therefore an error state in the completed $lts(\mathcal{C}^-)$. The error state is therefore reachable in the parallel composition of $\mathcal{M}_{\text{weak}}$ with the completed $lts(\mathcal{C}^-)$, and the above counterexample is returned to L* to refine the conjecture.

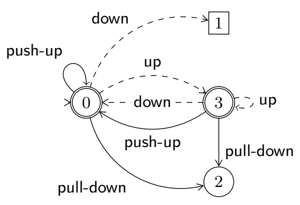


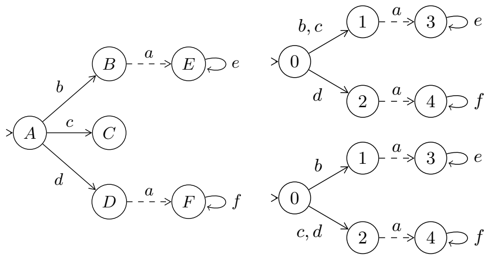
Figure 4: Intermediate 3DFA candidate for the VTS example. State 1 is the don't care state and 2 is the rejecting state.

4.3 Minimizing 3DFAs

The minimization step consists of computing a minimal (in terms of numbers of states) DFA consistent with the 3DFA $\mathcal{C}_U$ produced by L^*. We use the algorithm proposed in [18] to perform this step. The algorithm computes a set *Comp* of sets of *compatible states* called *compatibles* for short. We refer the reader to [18] for a detailed definition, but intuitively, compatibles identify states that could be merged because they exhibit compatible behaviour. Two behaviors are compatible if the two states to which they lead are both accepting, or both rejecting, or at least one of them is a don't care, in which case it could be interpreted either way. Note that some states may belong to multiple compatibles, which means that there exist several choices when computing a consistent DFA for $\mathcal{C}_U$.

In generating an abstraction based on $\mathcal{C}_U$, the states of the 3DFA are merged according to the compatibles, but each state should be assigned to a single compatible among all the choices. In order to guarantee a minimal abstraction, all the possible matching functions should be explored. That amounts to solving the set covering problem, known to be NP-complete [13]. The search can be improved by using heuristics, as proposed in [19] for example.

Figure 5(a) shows an example of a system for which the maximal compatibles overlap. States B and C are in the same compatible, and so are states C and D. The algorithm will output one of the two minimal full-control abstractions depicted in Figures 5(b).



(a) The system model. (b) The two abstractions.

Figure 5: A system for which the full-control similarity is not transitive, with two possible minimal full-control abstractions.

4.4 Non-Determinism

As mentioned, system models used throughout this work are expected to be full-control deterministic, as defined in Section 3.2, i.e., the same observable sequence will always lead to states where the same set of commands is available. Indeed, if a sequence may lead to different commands, then the user cannot possibly know what commands are allowed after that sequence. Assume, for example, that we modify our running example of Figure 1 by replacing the "up" transition from low-2 to low-3 with a τ (unobservable) transition. The modified system is no longer full-control deterministic. If we use our learning framework to compute an abstraction for this modified example, the learning process fails on the following sequence:

$\langle$up, down, down, down, up, push-up, up, pull-down, pull-down$\rangle$

In general, the learning algorithm fails when it cannot use a counterexample to refine its current candidate. This happens whenever counterexamples obtained from conjectures disagree with information obtained through membership queries. In our example, the above sequence is returned from a failed conjecture check as a trace that should be included in the abstraction but is not. However, a query on that sequence returns **F** as a result, because there exist executions of that sequence where the last pull-down is not allowed.

Such conflicting information by the teacher confuses the learner. The algorithm fails and reports the contradiction, along with the abstraction built so far, which may still be a useful partial result. Furthermore the failing trace precisely points to the source of non-determinism that prevented successful generation. In contrast, the bisimulation-based algorithm of [5] simply fails on systems that are non full-control deterministic, without providing any tangible information. Note that the sequence obtained is not minimal because the model checking Oracle used a depth-first search strategy; breadth-first search could be used alternatively for getting shortest counterexamples.

5. IMPLEMENTATION AND EVALUATION

We have implemented the presented learning framework within JavaPathfinder (JPF), as a new project (jpf-hmi). JPF[1] [23]) is an open-source, extensible verification framework developed by the RSE group at NASA Ames. It is a software model checker that handles Java bytecode directly. Details about jpf-hmi are provided in a paper submitted in the JPF workshop associated with ASE 2011.

The approach developed in this paper has been evaluated on the following six different models:

- **VTS** is a simple model of a vehicle transmission system from [11], used as illustration in Section 3.
- **AirConditioner** comes from [5] and is derived from the user manual of an air conditioner.
- **TimedVCR** is based on a model of a video-cassette recorder (VCR) developed in ADEPT, a toolset for analyzing HMI [8]. Its large number of states is due to a float-type variable *tapeRemaining*. In **SimpleVCR**, the *tapeRemaining* variable is omitted: internal states of the system with different values of remaining tape length are not distinguished in the system model. **FullVCR** refines SimpleVCR with different tape speeds.
- **AlarmClock** is a partial model of an alarm clock and **AlarmClock2** is a version with larger ranges for time values, which result in many internal τ transitions.

We have attempted to generate abstractions for all these models using both the bisimulation-based technique from [5]

[1]http://babelfish.arc.nasa.gov/trac/jpf/

	System	Abstraction	Bisim.	Learning	
	States / Trans.	States / Trans.		3DFA states	Total
VTS	8 / 20	5 / 14	10 ms	10	92 ms
AirConditionner	154 / 885	27 / 150	177 ms	51	6 271 ms
TimedVCR	3 352 / 15 082	2 / 9	1 031 ms	6	614 ms
SimpleVCR	20 / 110	2 / 9	65 ms	6	250 ms
FullVCR	24 / 261	4 / 24	45 ms	11	432 ms
AlarmClock	42 / 215	5 / 14	–	14	512 ms
AlarmClock2	1 734 / 67 535	5 / 15	–	14	30 831 ms

Table 1: Experimental results.

and the learning-based approach proposed here. Table 1 summarizes the results of the experiments. Execution times were measured on the same machine, and represent the total time needed to compute the minimal full-control abstraction. For the learning-based approach, this comprises generating the τ^*-closure, learning the 3DFA and minimizing to the LTS of the abstraction. τ^*-closure is also needed in the bisimulation-based approach. It accounts for 10 seconds on the AlarmClock2 model and is negligible for all other models, which have very few τ transitions. Minimization accounts for less than 10% of the time in all cases. No algorithm is uniformly better than the other in terms of execution time. As expected, the learning-based algorithm performs better on the larger TimedVCR model.

When both approaches work, they produce the same models. The AlarmClock model has a similar structure to that of Figure 5(a). As a consequence, the bisimulation-based approach fails while the learning-based approach generates one possible full-control abstraction. In the TimedVCR and SimpleVCR models, all commands are permanently enabled once the system is turned on, which results in a rather trivial two-state abstraction. The FullVCR yields a larger, more informative 24-state model because commands are not always enabled.

6. DISCUSSION AND CONCLUSION

We proposed a framework to automatically learn an abstraction for a given system model, as an alternative to an existing bisimulation-based approach [5]. We have demonstrated, through a number of experiments, that the learning framework can be more efficient in some cases, but that it also waives some restrictions of the existing approach, making our framework applicable to a larger number of systems. Moreover, the learning framework can provide useful diagnostic messages when it detects violation of full-control determinism.

Note that the latest feature of our framework can be used to detect another important problem in HMI systems, namely *mode confusion*. Mode confusion happens when the user believes that the system is in a different mode of operation than it actually is, which may lead into incorrect and hazardous maneuvers. The complex flight guidance systems found on modern civil aircrafts constitute a prominent target for this kind of analysis.

In the bisimulation-based approach of [5], modes are handled by enriching models with mode assignments on system model and abstraction states, and refining the algorithm to preserve mode consistency. The same result can be achieved within the current framework, by adding self-loop transitions $s \xrightarrow{m} s$ on system states to indicate that s is within mode m. Treating these mode actions as commands ensures that the abstraction "knows" in which mode the system is at any time. Conversely, mode confusion will occur if the same observable sequence leads to different modes. Since modes are treated as commands, mode confusion can be detected as an instance of violation of full-control determinism, where the last action in the failing sequence is a mode action.

In terms of performance, our experiments showed that, when both approaches are applicable, there is no clear winner between them. One could therefore include them both in an HMI analysis environment, apply them in parallel, and use the results of the one that terminates first. In the future, we plan on working on optimizations to the current algorithms. Moreover, we are working on connecting jpf-hmi to the ADEPT tool for the specification and generation of user-interfaces for HMI systems [8]. We have almost completed an automatic translation of ADEPT models into statecharts as supported by the JPF tool. This will allow us to have access to additional realistic examples that have been developed in the domain of HMI systems. Such examples will be used to thoroughly evaluate but also evolve our techniques for practical use in the real world. In particular, scalability is a major direction that we need to pursue; most systems in the HMI domain, such as autopilots, are large and complex and would challenge any formal analysis technique.

Acknowledgments

This work is partly supported by project MoVES under the Interuniversity Attraction Poles Programme — Belgian State — Belgian Science Policy.

7. REFERENCES

[1] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 98–109, New York, NY, USA, Jan. 2005. ACM.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.

[3] J. C. Campos and M. D. Harrison. Systematic analysis of control panel interfaces using formal tools. In *Proceedings of the 15th International Workshop on the Design, Verification and Specification of Interactive Systems*, number 5136 in Lecture Notes in Computer Science, pages 72–85. Springer-Verlag, July 2008.

[4] Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFAs for compositional verification. In S. Kowalewski and

A. Philippou, editors, *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *Lecture Notes in Computer Science*, pages 31–45, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] S. Combéfis and C. Pecheur. A bisimulation-based approach to the analysis of human-computer interaction. In G. Calvary, T. N. Graham, and P. Gray, editors, *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'09)*, pages 101–110, New York, NY, USA, 2009. ACM.

[6] A. Degani. *Taming HAL: Designing Interfaces Beyond 2001*. Palgrave Macmillan, Jan. 2004.

[7] M. Emmi, D. Giannakopoulou, and C. S. Păsăreanu. Assume-guarantee verification for interface automata. In J. Cuellar and T. Maibaum, editors, *Proceedings of the 15th International Symposium on Formal Methods (FM'08)*, volume 5014, pages 116–131, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] M. S. Feary. A toolset for supporting iterative human – automation interaction in design. Technical Report 20100012861, NASA Ames Research Center, Mar. 2010.

[9] D. Giannakopoulou and C. S. Păsăreanu. Interface generation and compositional verification in JavaPathfinder. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interface. In *Proceedings of the 10th European Software Engineering Conference (ESEC'05)*, pages 31–40, New York, NY, USA, Sept. 2005. ACM.

[11] M. Heymann and A. Degani. Formal analysis and automatic generation of user interfaces: Approach, methodology, and an algorithm. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 49(2):311–330, Apr. 2007.

[12] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC'93)*, pages 228–240, New York, NY, USA, 1983. ACM.

[13] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[14] N. G. Leveson, L. D. Pinnel, S. D. Sandys, S. Koga, and J. D. Reese. Analyzing software specifications for mode confusion potential. In *Workshop on Human Error and System Development*, pages 132–146, 1997.

[15] N. G. Leveson and C. S. Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[16] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, Dec. 1987.

[17] E. Palmer. Oops, it didn't arm. – a case study of two automation surprises. In *Proceedings of the 8th International Symposium on Aviation Psychology*, pages 227–232, 1996.

[18] M. C. Paull and S. H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, EC-8(3):356–367, Sept. 1959.

[19] J. M. Pena and A. L. Oliveira. A new algorithm for the reduction of incompletely specified finite state machines. In *Proceedings of the 9th IEEE/ACM International Conference on Computer-Aided Design (ICCAD'98)*, pages 482–489, New-York, NY, USA, Nov. 1998. ACM.

[20] J. Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75(2):167–177, Feb. 2002.

[21] N. B. Starter and D. D. Woods. How in the world did we ever get into that mode ? Mode error and awareness in supervisory control. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 37(1):5–19, Mar. 1995.

[22] H. Thimbleby and J. Gow. Applying graph theory to interaction design. In J. Gulliksen, editor, *Engineering Interactive Systems 2007/DSVIS 2007*, number 4940 in Lecture Notes in Computer Science, pages 501–518. Springer-Verlag, 2008.

[23] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 3–12, 2000.