Guaranteeing Consistency in Text-Based Human-Computer-Interaction

Bernhard Beckert [1]

Department of Computer Science
University Koblenz-Landau
Koblenz, Germany

Gerd Beuster [2,3]

Department of Computer Science
University Koblenz-Landau
Koblenz, Germany

Abstract

Wrong assumptions about the state of the computer system are a main source of error in human-computer interaction. We show how consistency requirements between the state of a computer system and the user's assumptions about the state can be formally defined. The definition of HCI consistency is used to show correctness of a methodology to ensure consistency for TTY-based applications.

Key words: Formal Methods, Security, HCI

1 Introduction

Security of interactive systems critically depends on correct display of the system's state. Only if the user's assumptions about the state of the system are correct, can he make informed decisions about the further course of action. Informally, a system is consistent if the user's assumptions about the system correspond to the actual system state whenever he interacts with the system. There are two main sources for wrong assumptions about system state leading to inconsistent systems:

[1] Email: beckert@uni-koblenz.de
[2] Email: gb@uni-koblenz.de
[3] This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the authors. See http://www.verisoft.de for more information about Verisoft.

This paper has been presented at the
1st International Workshop on Formal Methods for Interactive Systems
URL: fmis.iist.unu.edu

- Inconsistency during updates

  Human-Computer Interaction is inherently asynchronous. Execution of user commands and updates of the data displayed by the output device take time. Due to the inherently asynchronous character of Human-Computer Interaction, the user may err about the system state; either because commands have not been executed yet, or because the screen has not been updated.

- Insufficient information or wrong interpretation of data

  The system may not provide enough information to determine the system sate, or the user may interpret application output wrongly. A large part of the specification of interactive applications is concerned with the relation between user input and the information shown to the user. For example, when editing a text, the current (internal) state of the text should be shown to the user, and user input should cause changes to the text. Usually, the specification of user input and system output is rather informal. Specifications declare that something "is shown on the screen" and the user "enters a text." In most cases, this informal description is sufficient. However, in security-critical applications, a precise and formal definition is desirable.

The second point, wrong interpretation of data, is addressed in a number of works. Reeder and Maxion [21] analyzed the problem of representing NTFS file permissions on Windows XP systems and developed the design principle of "anchor-based subgoaling" in order to mitigate the problem.

In this work, we address the first source of errors, inconsistencies during updates. Most user interface security requirements are highly application specific. However, there are also some generic requirements. We show that for a large class of applications, it is possible to define generic requirement in a formal way. In this paper, we focus on one of these generic requirements: The user should always be aware of the system state when he issues a command. We show how consistency during updates can be guaranteed for text-based applications.

In Section 4 we develop a formal definition of consistency based on the generic computer security requirement of application Integrity. In Section 5, we show that the common approach to model interactive applications does not guarantee consistency. We provide an alternative model for which consistency can be guaranteed. In Sections 6 and 7, a generic specification template for interactive applications that guarantee consistency in HCI is presented. In this paper, we consider text-based user interfaces only, but our methods can be extended for handling graphical user interfaces. In the Verisoft project (`http://www.verisoft.de`), our methods are applied to specify and verify an email system.

2 Related Work

We build upon work on formal methods for developing computing systems, human-computer interaction (HCI) research, and secure system design.

Formal methods, human computer interaction, and security are established fields of computer science research. There is also work combining each two of these fields. Formal methods have been used to specify human-computer interaction. User interfaces have been designed and evaluated under security aspects. System security has been treated with formal methods. In this work, we combine all *three* fields.

Abowd et al. [1] and Jain [15] give a survey of formal languages for the description of user interfaces. More overviews are given in two (different) books called *Formal Methods in Human-Computer Interaction* [13,19]. An early contribution to formal methods for the description of user interaction is the PIE model, developed by Dix and Runciman [9]. In this model, system behavior is defined as a function from commands issued by the user to effects produced by the system. In case of a text-based user interface, the input is a sequence of keystrokes and the output are characters displayed on the screen [8].

Carr's Interaction Object Graphs (IOG) are an extension of statecharts for modeling elements of graphical user interfaces and their interactions [4]. It allows a description both on the pixel-level and on an aggregated level. IOGs focus on graphical user interfaces, and the language used to describe them is directly executable. The formalism of IOG allows basic reasoning tasks like testing for reachability of all states. Statecharts are also used by Degani et al. for specification of the interaction interfaces between human users and machines, addressing the question what information about a machine's state is required in order to operate it safely [6].

Sucrow [23] uses graph grammars to describe graphical user interface elements. Changes in the GUI are modeled by re-write rules. Palanque et al. [18] use hierarchical Petri nets to combine user models and a system models of interactive systems. Berstel et al. developed "Visual Event Grammars" (VEG), a formal method for the specification and validation of graphical user interfaces [2]. They describe complex graphical user interface as communicating automata.

PIE and similar formalisms put an emphasis on describing the I/O behavior of a computer system and are suitable for automated reasoning, e.g. with model checkers. Rushby uses model checking in order to detect potential discrepancies between system behavior and the mental models of system users [22].

Other approaches like Task Knowledge Structures (TKS) [12], (Extended) Task Action Grammar ((E)TAG) [5], and Goals Operators Methods Selection-rules (GOMS) [16] focus on providing cognitive models of the user. TKS provides an explicit representation of the cognitive model of the user. TAG

allows a precise formal description of the user actions, the user's knowledge and the user's internal representation of the system (what the user thinks about the system). GOMS is more oriented towards psychological analysis of user behavior and timed measurement of user activity. A major weakness of GOMS is that it is limited to sequential user plans, and that it does not provide means to generate application specifications from user models. ConcurTaskTrees [20], developed by Paterno et al., provides a richer formalism for the description of user behavior and generation of application specifications. Harrison et al. [27] use formal methods to derive requirements for human-error tolerance from task descriptions.

A general weakness of these formal HCI models is that they require detailed models of the user behavior in order to model the interaction between a computer system and a user. While computer systems can (and should) be formally specified, a formal user model is always based on assumptions about the user which may or may not be true. The approach presented in this we paper make no unnecessary assumptions about the user.

In [13], Dix and Harrison develop the concept of "State Display Conformance" which is closely related to the consistency requirements developed in this paper. It should be noted that Grudin's argument *against* user interface consistency requirements [11] does not apply to the work presented in this paper. He argues that consistency defined as having similar user interface elements for similar functionality can not be generalized, because similarity depends on context. Our work however does not address consistency within a user interface, but consistency between a user's mental representation of a system state and the actual system state.

In a number of works, formal specification methods like Z have been applied to user interface design. One of the first formal specifications of interactive components was the specification of a text editor in Z in Sufrin's paper *Formal specification of a display editor* [24]. Based on Sufrin's specification, Booth and Jones implemented an editor in the Miranda functional programming language [3]. Goldson [10] and Hussey/Carrington [14] provide more case studies in using Z for user interface specification.

3 Notation

We specify the abstract behavior of system components by Input Output Labeled Transition Systems (IOLTS) and Linear Temporal Logic (LTL). Below, we define these concepts and some related notions used throughout this paper.

Definition 3.1 A *Labeled Transition System* (LTS) is a tuple $L = (S, \Sigma, s_0, \rightarrow)$ where S is a set of *states*, $s_0 \in S$ is an *initial state*, Σ is a set of *labels*, and $\rightarrow \subseteq S \times \Sigma \times S$ is a *transition relation*. We use the notation $p \xrightarrow{\sigma} q$ for $(p, \sigma, q) \in \rightarrow$.

Definition 3.2 An *Input Output Labeled Transition System* (IOLTS) is an

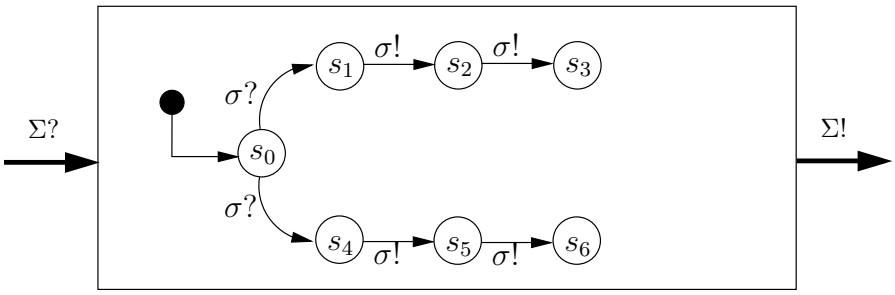

Fig. 1. State Transition Diagram representation of an IOLTS.

LTS $L = (S, \Sigma, s_0, \rightarrow)$ with $\Sigma = \Sigma? \cup \Sigma! \cup \Sigma I$. We call $\Sigma?$ the *input alphabet*, $\Sigma!$ the *output alphabet*, and ΣI the *internal alphabet*.

We use state transition diagrams to visualize IOLTS. An example is shown in Figure 1.

The combination of two IOLTSs L_a and L_b where the output alphabet of L_a is the input alphabet of L_b is called a *composition*:

Definition 3.3 Let $L_a = (S_a, \Sigma_a, s_{0a}, \rightarrow_a)$, $L_b = (S_b, \Sigma_b, s_{0b}, \rightarrow_b)$ be two IOLTS with $\Sigma!_a = \Sigma?_b$. The *composition* $(L_a||L_b) = (S, \Sigma, s_0, \rightarrow)$ of L_a and L_b is defined by:

$$
\begin{aligned}
S &= S_0 \times S_1 \\
\Sigma? &= \Sigma?_a \\
\Sigma! &= \Sigma!_b \\
\Sigma I &= \Sigma I_a \cup \Sigma I_b \cup \Sigma!_a \\
s_0 &= (s_{0a}, s_{0b}) \\
\rightarrow &= \{((s_a, s_b), \sigma, (s'_a, s_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ with } \sigma \in \Sigma?_a \cup \Sigma I_a\} \cup \\
&\quad \{((s_a, s_b), \sigma, (s_a, s'_b)) \mid s_b \xrightarrow{\sigma}_b s'_b \text{ with } \sigma \in \Sigma!_b \cup \Sigma I_b\} \cup \\
&\quad \{((s_a, s_b), \sigma, (s'_a, s'_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ and } s_b \xrightarrow{\sigma}_b s'_b \text{ with } \sigma \in \Sigma!_a = \Sigma?_b\}
\end{aligned}
$$

Often, components are combined by *mutual* composition. In mutual composition, the output of L_a serves as input for L_b, and the output of L_b serves as input of L_a (this is illustrated in Figure 2).

Definition 3.4 Let $L_a = (S_a, \Sigma_a, s_{0a}, \rightarrow_a)$ and $L_b = (S_b, \Sigma_b, s_{0b}, \rightarrow_b)$ be IOLTS.

We assume the input and output alphabets of L_a and L_b to consist of internal and external subsets, where the internal input is denoted with $\Sigma?I$, the external input with $\Sigma?I$, the internal output with $\Sigma!I$, and the external output with $\Sigma!E$. And we demand that these subsets are chosen such that $\Sigma!I_a = \Sigma?I_b$ and $\Sigma!I_b = \Sigma?I_a$.

Then, the *mutual composition* $(L_a||_m L_b) = (S, \Sigma, s_0, \rightarrow)$ of L_a and L_b is defined by:

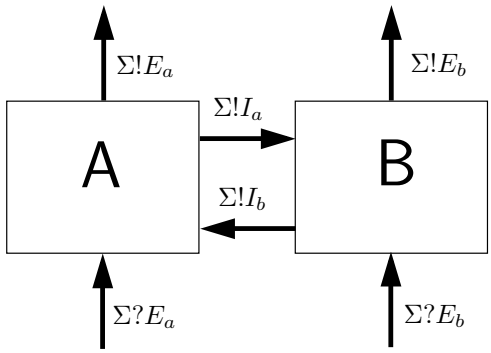

Fig. 2. Mutual composition of IOLTSs.

$$
\begin{aligned}
S &= S_0 \times S_1 \\
\Sigma? &= \Sigma?E_a \cup \Sigma?E_b \\
\Sigma! &= \Sigma!E_a \cup \Sigma!E_b \\
\Sigma I &= \Sigma I_a \cup \Sigma I_b \cup \Sigma!I_a \cup \Sigma!I_b \\
s_0 &= (s_{0a}, s_{0b}) \\
\rightarrow &= \{(s_a, s_b), \sigma, (s'_a, s_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ with } \sigma \in \Sigma?E_a \cup \Sigma!E_a \cup \Sigma I_a\} \cup \\
&\quad \{(s_a, s_b), \sigma, (s_a, s'_b)) \mid s_b \xrightarrow{\sigma}_b s'_b \text{ with } \sigma \in \Sigma?E_b \cup \Sigma!E_b \cup \Sigma I_b\} \cup \\
&\quad \{(s_a, s_b), \sigma, (s'_a, s'_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ and } s_b \xrightarrow{\sigma}_b s'_b \text{ with} \\
&\qquad\qquad \sigma \in \Sigma!I_a \cup \Sigma!I_b = \Sigma?I_b \cup \Sigma?I_a\}
\end{aligned}
$$

The input/output behavior of a component is described by *traces*, which are (possibly infinite) sequences of elements from the alphabet Σ, and *paths*, which are corresponding sequences of states.

Definition 3.5 Let $L = (S, \Sigma, s_0, \rightarrow)$ be an IOLTS. Then, a *path* is a sequence $\langle s_0, s_1, \ldots \rangle$ of states from S with $s_i \rightarrow s_{i+1}$ for all $i \geq 0$. A *trace* (of L) is a sequence $\langle \sigma_0, \sigma_1, \ldots \rangle$ of elements of Σ such that there is a path $\langle s_0, s_1, \ldots \rangle$ with $s_i \xrightarrow{\sigma_i} s_{i+1}$ $(i \geq 0)$.

We use Linear Temporal Logic (LTL) to describe properties of components. The syntax of LTL is defined as usual, i.e., given a set P of atomic propositions, LTL formulae ϕ are constructed inductively by:

$$\phi ::= p | \phi \vee \phi | \phi \wedge \phi | \neg\phi | X\phi | \phi U \phi | G\phi | F\phi \qquad (p \in P)$$

Now, we can use IOLTSs to interpret LTL formulas—in combination with valuations λ that map atomic propositions to the states in which they are true. The satisfaction relation is extended to more complex formulae as usual.

Definition 3.6 Given an IOLTS $L = (S, \Sigma, s_0, \rightarrow)$ and a set P of atomic propositions, a *valuation* λ is a mapping from P to S. An atom p is said to be true in $s \in S$ iff $s \in \lambda(P)$.

Given a path $c = \langle s_0, s_1, \ldots \rangle$, by c^i we denote the sub-path of c starting at s_i.

Whether an LTL formula ϕ is satisfied by a path c and a valuation λ, denoted by $L, \lambda, c \models \phi$, is inductively defined as follows:

- $L, \lambda, c \models \top$
- $L, \lambda, c \models \phi$ if $\phi \in P$ and $s_0 \in \lambda(\phi)$
- $L, \lambda, c \models \neg\phi$ if not $L, \lambda, c \models \phi$
- $L, \lambda, c \models \phi \wedge \psi$ if $L, \lambda, c \models \phi$ and $L, \lambda, c \models \psi$
- $L, \lambda, c \models \phi \vee \psi$ if $L, \lambda, c \models \phi$ or $L, \lambda, c \models \psi$
- $L, \lambda, c \models \mathbf{X}\phi$ if $L, \lambda, c^1 \models \phi$
- $L, \lambda, c \models \phi \mathbf{U} \psi$ if (a) $L, \lambda, c \models \psi$ or (b) there is some $i \geq 1$ s.t. $L, \lambda, c^i \models \psi$ and $L, \lambda, c^k \models \phi$ for all $0 \leq k < i$
- $L, \lambda, c \models \mathbf{G}\phi$ if $L, c^i \models \phi$ for all $i \geq 0$
- $L, \lambda, c \models \mathbf{F}\phi$ if $L, c^i \models \phi$ for some $i \geq 0$

An LTL formula ϕ is said to be satisfied by a valuation λ, denoted by $L, \lambda \models \phi$, iff $L, \lambda, c \models \phi$ for all paths c of L. And ϕ is said to be satisfied by L, denoted by $L \models \phi$ iff $L, \lambda \models \phi$ for all valuations λ.

4 Formal Definition of User Interface Integrity

The aim of computer security is to guarantee access to services and resources to authorized persons, while preventing access and manipulation by unauthorized parties. The basic security threats are *Data Leaking*, *Data Manipulation*, and *Program Manipulation* [7]. These are countered by the core security requirements, usually abbreviated as *CIA*:

Confidentiality: Information is available to authorized parties only.

Integrity: Both the assumptions of the user about the application, and the assumptions of the application about the user are correct.

Availability: Accessibility of services and data is guaranteed.

Adapting these concepts to user interface security is straightforward:

HCI Confidentiality: No secret information is leaked via the user interface.

HCI Integrity: There is a correspondence between the configuration of the application (defined by its internal state and data), and the user's assumption about the data and the state.

HCI Availability: The user interface must guarantee reachability of desirable states, and it must prevent user interactions that lead to transitions into undesirable states.

In the following, we concentrate formalizing the integrity requirement. Informally, we define HCI Integrity as follows:

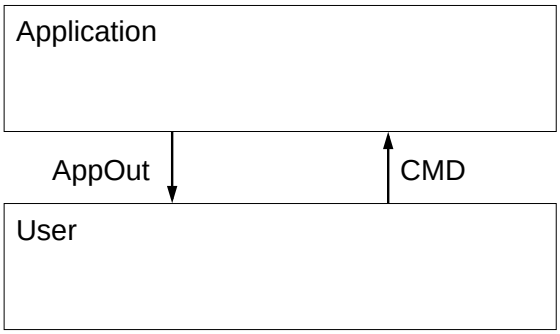

Fig. 3. Basic System (User + Application) Model

Definition 4.1 HCI Integrity: Whenever the user makes a critical decision, all critical properties are the same in the application and the user's assumption about the application.

In order to maintain a most general view on all possible applications and user models, we do not define *what* constitutes critical properties and their correct interpretation by the user. We only assume that there *are* critical properties, and that the user may or may not have correct assumptions about them. We assume that in all critical states atomic proposition *critical* holds, and that there are atomic propositions $a_0, \ldots, a_n$ representing critical properties of the application, and $u_0, \ldots, u_n$ representing the user's assumptions about these properties. With these definitions, HCI Integrity is defined by the LTL formula

$$\mathbf{G}(\mathit{critical} \rightarrow ((a_0 \leftrightarrow u_0) \wedge (a_1 \leftrightarrow u_1) \wedge \cdots \wedge (a_n \leftrightarrow u_n))) \quad (1)$$

The definition of $a_0, \ldots, a_n$ and $u_0, \ldots, u_n$ depends on the actual application and user models. For given user and application models, automated reasoning techniques (e.g., model checking) can be used to check if the HCI Integrity formula holds.

5 Guaranteeing Integrity

In the last Section, we developed a formal definition of integrity. In order to apply the definition, suitable user and application models and definitions of a valuation function λ must be provided. In this Section, we use the methodology to deduce required properties of a generic class of text-based user interface. Based on this, a specification of a main execution loop for this class of applications is developed in Section 6.

In a generic model of a TTY application, one user interacts with one application. A keyboard is used as the input device and a TTY screen as the output device. This model is depicted in Figure 3. Two types of messages are used to exchange information between the user and the application: *AppOut* is the data type for information shown on the screen. *CMD* is the data type for input given by the user. This model can be further structured without losing

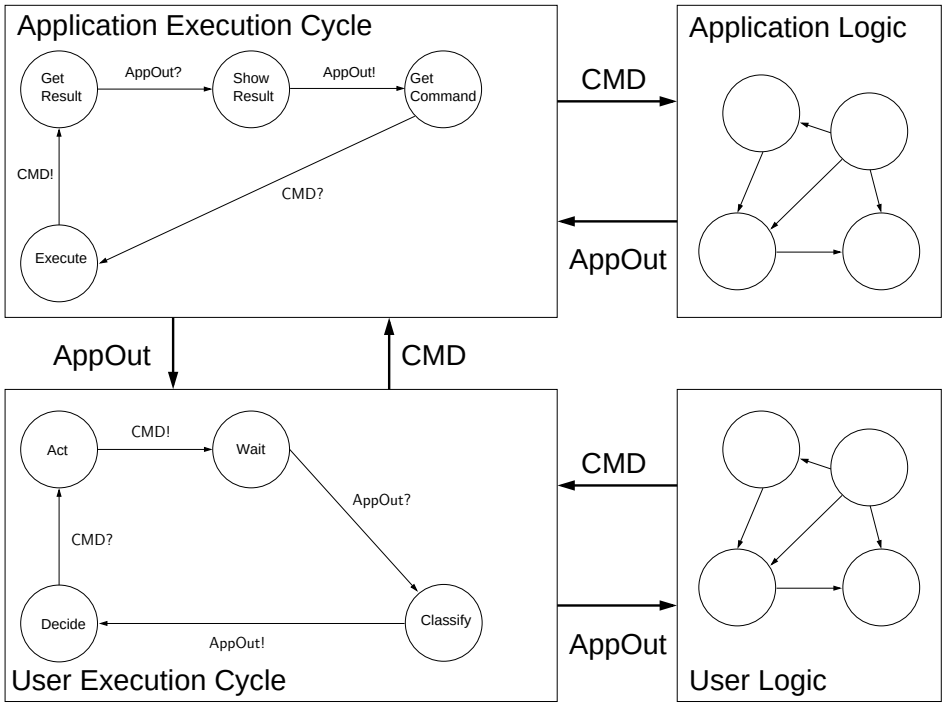

Fig. 4. Basic generic model of user and application.

generality. All well-designed applications (and all reasonable models of user behavior) split up the components into a generic execution loop, governing the general behavior of the application (or the user), and an application (task) specific component. The separation of a generic execution loop and a task specific component serves two purposes: It follows established system design practice and therefore allows realistic modeling of applications. Secondly, the separation in a generic and an application specific component allows to deduce properties that hold for all applications following this design, independent of the concrete application's task.

A basic model following this approach is shown in Figure 4. In this model, *AppOut* and *CMD* are variables representing all possible command input and application output. Question marks after variable names indicate reading of an input value, and exclamation marks indicate writing of an output value. Thus, in one cycle of application execution the following steps are taken: The application waits for the user to enter a command (GetCommand $\xrightarrow{\text{CMD?}}$ Execute). The command is processed by application logic (Execute $\xrightarrow{\text{CMD!}}$ GetResult, GetResult $\xrightarrow{\text{AppOut?}}$ ShowResult), and the result of the computation is forwarded to the output device (ShowResult $\xrightarrow{\text{AppOut!}}$ GetCommand). In the same way, the user reads application output, evaluates which command should be issued next, and enters the command into the input device.

This basic model already allows to deduce interesting properties in respect to Integrity constraints. A reasonable assumption is that all decisions made

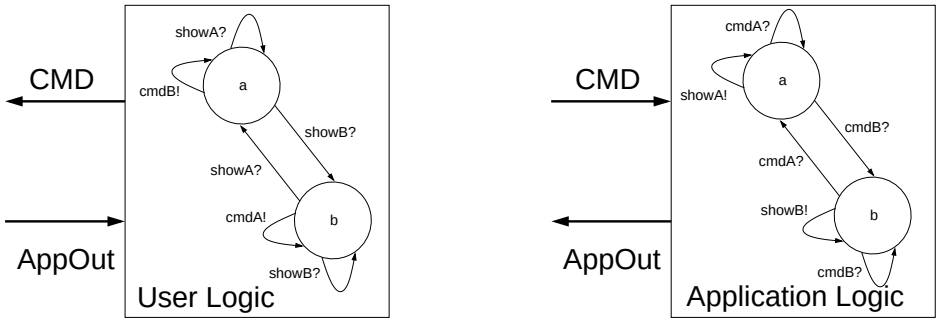

Fig. 5. Simple Logic Modules.

AppExec	**UserExec**	**AppLogic**	**UserLogic**	**AppOut**	**CMD**
ShowResult	Wait	a	b	-	-
GetCommand	Wait	a	b	showA	-
GetCommand	Classify	a	b	showA	-
GetCommand	Decide	a	a	showA	-
GetCommand	Act	a	a	showA	-
GetCommand	Wait	a	a	showA	cmdB
Execute	Wait	b	a	showA	-
Execute	Classify	b	a	showA	-
Execute	Decide	b	a	showA	-

Fig. 6. Refutation of naïve model (excerpt)

by the user are critical::

$$\lambda(critical) = \{\text{UserExeCyc.Decide}\}$$

Critical properties of an application, and user assumptions about critical properties, always depend on the user and application model. We provide most simple definitions of these components in order to show a general weakness of the naïve application and user execution cycle model. In this most simple component definition, there are only two commands, two application outputs, and two states in both the user and application logic model. These logic models are shown in Figure 5. As the security relevant property, we define the question whether the application is in state "a":

$\lambda(a_0) = \{\text{AppLogic.a}\}$
$\lambda(u_0) = \{\text{UserLogic.a}\}$

Integrity is not guaranteed for this model. The problem lies in the lack of consistency, as the trace given in Figure 6 shows: When the user decides about

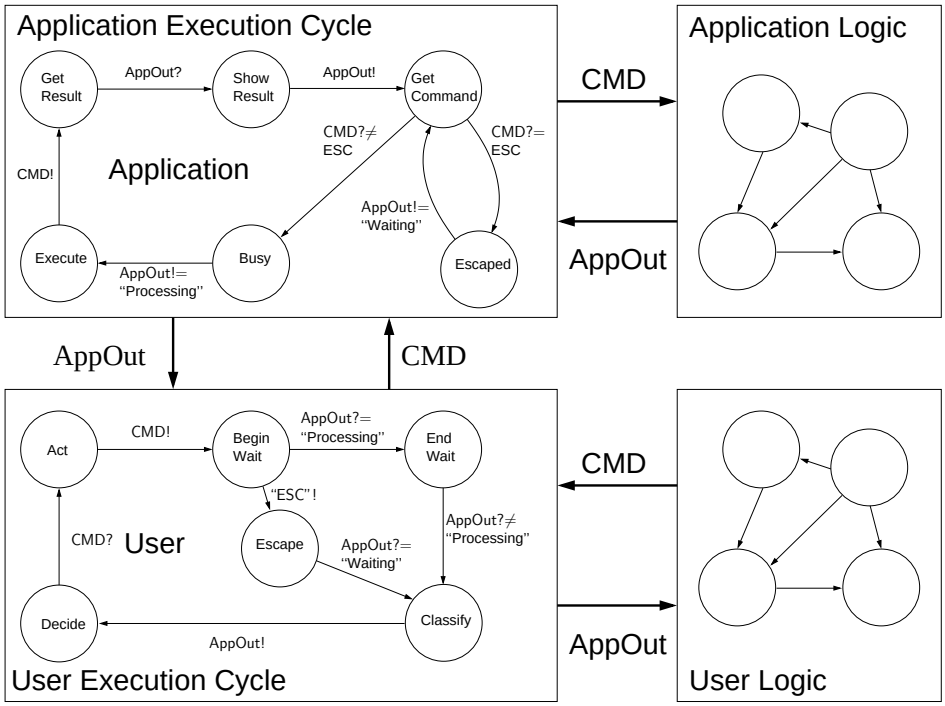

Fig. 7. Refined Model

the next command for the second time, he does not recognize that execution of the first command has not been completed.

The system model is not a model for Formula 1, because the application configuration may change while the user issues a command. This problem is well known from real-world computer systems: if the user does not know if a command has been executed, he may be tempted to re-issue the command, resulting in double execution of the command. In the worst case, this can lead to a security problem, for example when the user accidentally confirms a critical action twice. Next, we show a solution for the problem. In Section 6, we apply the solution to a real-world program specification.

The problem can be solved by introducing new states for synchronization, as shown in Figure 7. In this model, the system gives visual feedback indicating whether it is waiting for user input or processing user input. Once the application received a user command, it shows "processing" on the screen. When processing is finished, the new application status is shown. Just showing the message "processing" while executing user commands is not sufficient, however. Depending on execution speed, the user may not recognize the message "processing" at all (because it was shown for a very short amount of time), our it may take a long time before the message is shown (in case the system is slow). In order to give the user the ability to distinguish between the two cases, an escape-key is introduced. If the user pushes the escape-key, the message "waiting" is shown. This way, if the user does not know about the state of the input process, he can press "escape" and wait for the message "waiting" to show up. The model given in Figure 7 satisfies the integrity

constraint of Formula 1.

While it is perfectly fine to change the specification of the application, one may ask if it is acceptable to change the user model, i.e. our assumptions about the user. We do think this is acceptable. It is common practice to train the user on how to operate a system. For this, a formal user model allows to explicit state what a user has to know in order to operate the system.

6 Specification of Secure Interactive Applications

In Section 5 we showed that the naïve model of user and application interaction is not sufficient to guarantee consistency. While we showed that the refined model guarantees consistency for the given application logic and user logic components, we did not—and can not—show that the consistency constraint holds for all application and user logic components, because it does not solely rely on consistency between the user and the application model; the user must also have the right assumptions about the application model. He must have knowledge about the inner working of the application, and about the consequences of his actions. This knowledge is represented in the user logic module. Just like the user's and application's execution loops, the logic components of user and application are modeled as IOLTS. This requires a state-based representation of the application, and of the user's knowledge about the application.

In the last Section, basic application logic and user logic modules were used in the refutation of the naïve model. These example modules (given in Figure 5), had only two outputs: showA and showB. In actual applications, possible system configurations and outputs are much richer in detail. Even if considering TTY-based applications, we have screens with multiple rows and columns, where each cell can contain an alphanumeric character. Even on a moderately sized screen, the set of all possible combinations of output characters are too large to be modeled explicitly. Therefore, it is necessary to find a suitable abstraction of application states, application output and user assumptions about application states in order to make real-world applications suitable for automated model checking for consistency constraints. Of course, such security-relevant states are abstractions of the application's actual internal configuration, which is much richer in detail. Nevertheless, we assume that these states are the *right* abstraction in that the user has sufficient information about the internal configuration of the application if he or she knows the abstract state of the application.

In the following, our abstract model of an application assumes that an application can be in one of many states, and that the current state is represented in variable `applicConf`. User commands (usually corresponding to keystrokes entered by the user) trigger state transitions. Depending on the result of a command, the system transits into a new state. The actual specifications of command `execution` is application dependent. Pseudo code for

```
1: repeat
2:   {Show Result}
3:   updateScreen(confAsString(applicConf),applicConf)
4:   {Get Command}
5:   repeat
6:     cmd := getKeystroke()
7:     if cmd = ESC then
8:       {Escaped}
9:       updateScreen(confAsString(applicConf) +
                      ''Waiting'',applicConf)
10:    end if
11:  until cmd ≠ ESC
12:  {Busy}
13:  updateScreen(confAsString(applicConf) +
                  ''Processing'',applicConf)
14:  {Execute & Get Result}
15:  applicConf := execute(cmd,applicConf)
16: until cmd = QUIT
```

Algorithm 1. The main event loop

a main event loop implementing the Application Execution Cycle model from Figure 7 is given in Algorithm 1.

For the specification of the screen update function `updateScreen`, we use the following auxiliary functions:

- *confAsString*(`applicConf`) is a string that allows the user to identify the state of the application.
- *screenOutput*(`applicConf`) is a two-dimensional array of characters. It contains the correct screen output corresponding to `applicConf`. The actual definition of `screenOutput` is under the discretion of the application at hand.
- $stringAt(x, y)$ is the string shown on screen position (x, y).

We require that the current state of the application logic component plus optionally the additional information "waiting" or "processing" are shown in the first line of the screen. A specification for function `updateScreen` in OCL [4] is shown in Table 1.

It should be noted that we do not restrict ourselves to a certain application. The specification fits every applications requiring a secure, text-based user interface.

[4] The OCL specification should be understandable without deeper knowledge of OCL. See [25,26] for more information on OCL and [17] for the current language specification.

context	`updateScreen(status,conf)`
post	$stringAt(0,0) =$ `status` and
	$\forall k \in \{1,\ldots,screenHeight-1\}$:
	$stringAt(0,k) =$
	$screenOutput($`applicConf`$)[k-1]$

Table 1
Specification of the application's function for updating the screen contents

7 Verification

In order to verify that an implementation satisfies the consistency constraints, a number of assumptions about the user are necessary:

(i) The user observes the screen.

(ii) The user understands the output of *stateAsString*.

Under these assumptions, it is sufficient to show that the status string as provided by *stateAsString* is adequate, and that *updateScreen* is called as specified in the last Section. With these assumptions and the additional assumption that the operating system works correctly, verification of observability can be split into two parts:

(i) Proofs for the application's functions `execute` and `updateScreen`.

(ii) Proofs about the main event loop in respect to the application model.

The second part is generic, since the main event loop given in Algorithm 1 is applicable to all applications following our design methodology.

It should be noted that two calculi are integrated in our approach. The properties of the main execution loop need to be proven in some temporal calculus. Satisfaction of the requirements for the main execution loop can be proven by model checking. Proofs about the application's functionality can be executed in a calculus based on pre- and postconditions, e.g. Hoare logic. From the proofs about the application's functionality it follows that for each distinct system configuration, `updateScreen` produces a distinct and up-to-date screen representation. From the assumption that the user understands the chosen representation, it follows that observability is given immediately after every call of `updateScreen`.

In project Verisoft (`http://www.verisoft.de`), this approach is used to prove observability of an email client application in the context of a pervasively verified computer system.

8 Conclusions and Future Work

In this paper, we showed how formal methods can be used to guaranteed a fundamental requirement of user interface security.

In Section 4, we translated the generic security requirements of Confidentiality, Integrity, and Availability to human-computer interaction, and we gave a formal definition of Integrity for HCI: The user should always be aware of the current state of the system. In Chapter 5, we developed generic models for TTY-based application. We showed that a naïve approach leads to models that do not guarantee consistency. We provided a refined model that satisfies consistency constraints. In Section 6, we showed how the formal model can be transfered to actual applications, and what has to be shown about an application in order to ensure its security.

In project Verisoft (`http://www.verisoft.de`) our method is used to specify, implement and verify a secure email client. In Verisoft, both the operating system and the application program are formally verified based on that specification.

In the future, we plan to extend our work to other aspects of user interface security. Our goal is to create a systematic formal description of user interface security for interactive systems.

References

[1] G. D. Abowd, J. P. Bowen, A. J. Dix, M. D. Harrison, and R. Took. User interface languages: A survey of existing methods. Technical Report PRG-TR-5-89, Oxford University Computing Laboratory, October 1989.

[2] Jean Berstel, Stefano Crespi Reghizzi, Gilles Roussel, and Pierluigi San Pietro. A scalable formal method for design and automatic checking of user interfaces. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2):124–167, April 2005.

[3] Simon P. Booth and Simon B. Jones. A screen editor written in the miranda functional programming language. Technical Report TR-116, Department of Computing Science and Mathematics, University of Stirling, February 1994.

[4] David A. Carr. Interaction object graphs: an executable graphical notation for specifying user interfaces. In Philippe Palanque and Fabio Paternò, editors, *Formal methods in Human-Computer Interaction*, pages 141–155. Springer, 1997.

[5] Geert de Haan. *ETAG, A Formal Model of Competence Knowledge for User-Interface Design*. PhD thesis, Vrije Universiteit, Amsterdam, 2000.

[6] Asaf Degani, Michael Heymann, George Meyer, and Michael Shafto. Some formal aspects of human-automation interaction. Technical report, NASA, Moffett Field, CA: NASA Ames Research Center, 2000.

[7] Rüdiger Dierstein. Sicherheit in der Informationstechnik — der Begriff IT-Sicherheit. *Informatik Spektrum*, 27(4), August 2004.

[8] A. Dix and G. Abowd. Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, 11(6):334–346, 1996.

[9] A. J. Dix and C. Runciman. Abstract models of interactive systems. In P. Johnson and S. Cook, editors, *HCI'85: People and Computers I: Designing the Interface*, pages 13–22. Cambridge: Cambridge University Press, 1985.

[10] Doug Goldson. Formal modelling of interactive systems. In *Proceedings of APAQS 2OOO, the First Asia-Pacific Conference on Quality Software*, IEEE Conference Proceedings. IEEE Computer Society Press, 2000.

[11] Jonathan Grudin. The case against user interface consistency. *Communications of the ACM*, 32(Issue 10):1164–1173, October 1989.

[12] F. Hamilton. Predictive evaluation using task knowledge structures, 1996.

[13] Michael Harrison and Harold Thimbleby, editors. *Formal methods in human-computer interaction.* Cambridge Univ. Press, Cambridge, Mass., 1990.

[14] A. Hussey and D. Carrington. Specifying a web browser interface using Object Z. In Philippe Palanque, editor, *Formal methods in human computer interaction*, chapter 8. Springer, 1998.

[15] Vipul Jain. User interface description formalisms. Technical report, McGill University School of Computer Science, Montréal, Canada, 1994.

[16] Bonnie E. John and David E. Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction*, 3(Issue 4):320–351, December 1996.

[17] Object Modeling Group. *Unified Modelling Language Specification, version 1.5*, March 2003.

[18] Philippe Palanque, Remi Bastide, and Valerie Senges. Validating interactive system design through the verification of formal task and system models. In *Engineering for Human-Computer Interaction.* Chapman & Hall, August 1995.

[19] Philippe Palanque and Fabio Paternò, editors. *Formal methods in human computer interaction.* Springer, New York, London, 1998.

[20] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications.* Springer, 1999.

[21] Robert W. Reeder and Roy A. Maxion. User interface dependability through goal-error prevention. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 60–69, Washington, DC, USA, 2005. IEEE Computer Society.

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

[23] Bettina Sucrow. Formal specification of human-computer interaction by graph grammars under consideration of information resources. In *Automated Software Engineering*, pages 28–35, 1997.

[24] B. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, pages 157–202, 1982.

[25] Jos Warmer and Anneke Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, March 1999.

[26] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley Professional, 1998.

[27] Peter Wright, Bob Fields, and Michael Harrison. Deriving human-error tolerance requirements from task analysis. In *IEEE International Conference on Requirements Engineering*, 1994.