Formal Specification of Security-relevant Properties of User Interfaces[1]

Bernhard Beckert   Gerd Beuster

{beckert|gb}@uni-koblenz.de

University of Koblenz
Department of Computer Science

Abstract. When sensitive information is exchanged with the user of a computer system, the security of the system's user interface must be considered. In this paper, we show how security relevant properties of a user interface can be modelled and specified using the Object Constraint Language (OCL).

1 Introduction

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. In this paper, we show how security relevant properties of a user interface can be modelled, investigated, and ensured using formal methods.

2 Environment and Notation

In this paper, we model a text-based user interface. Input comes from the keyboard and output goes to a terminal with a fixed number of rows and columns for display of characters. Assuming no additional input from other sources (like a mouse or network card), the behavior of a text-based application can be described as a function from a (finite) sequence of keystrokes to a screen output. That is, the behavior is specified by what is supposed to appear on the screen after a particular sequence of keystrokes. We use *keyboard* to refer to keyboard input and *screenAt* to refer to screen output. When we want to refer to a specific

[1] 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.

screen position, we use the notation *screenAt*$[x, y]$ for the character shown at screen position (x, y).

To refer to keyboard input up to resp. screen output at a particular point t in time, we use *keyboard*(t) to denote the list of keystrokes entered up to time t, and *screenAt*(t) to denote the screen output at time t.

In a post-condition, t refers to the current time, i.e., the point in time when the function terminates, while $t@pre$ refers to the point in time when the function is entered. In this, we follow the common OCL syntax (though standard OCL does usually not contain explicit references to particular points in time).

3 Specifying Operating System Requirements

3.1 Overview

The operating system provides interfaces between application programs and the hardware. In the case of simple, text-based user interfaces, which we are examining in this paper, the operating system has to provide access to two resources: the keyboard and the screen. Most work on secure interface design assumes that the application runs in a safe and friendly environment. Although some work takes attacks on input/output facilities from the outside into account, interference with the input/output facilities from within the system (by trojans, worms, viruses, etc.) is usually not part of the attack scenarios. Here, we assume that the security critical application is running in a multi-process environment, where hostile processes may launch attacks on input/output facilities. We provide a formal method that *guarantees* security against software based man-in-the-middle attacks.

3.2 Specifying Screen Output Functions

Below, we give constraints specifying the operating system functions for accessing the screen.

```
context setChar(character, x,y)
post    if  ((x ≥ 0) and (x < screenWidth()) and
             (y ≥ 0) and (y < screenHeight()))
        then
            screenAt(t)[x, y] = character  and
            result = CHAR_SET_OK
        else
            result = POSITION_OUT_OF_BOUNDS
        endif
```

3.3 Specifying Keyboard Input Functions

Since user input comes from the keyboard, we can identify all user input during the lifetime of the application with a list of keystrokes, where a keystroke is a character (a character code) associated with a timestamp.

Usually, computer systems have an input buffer. This buffer is filled with the user's keystrokes independently of the current application's activity. When the application calls the operating system function for retrieving the next keystroke, the first keystroke of the keyboard buffer is returned. In the scenarios we are modeling, however, the use of a keyboard buffer is often not advisable. From the view of security, we want that the user approves or denies an activity only *after* he or she is aware of the options available. With a keyboard buffer, a user may enter commands that are executed at a later point in time. It could then happen that the user approves or denies an activity *before* the available options are shown to him or her. Therefore, we define the operating system function `getkeystroke` without using an input buffer. The function `getkeystroke` is specified to return the next character typed *after* its invocation.

```
context getkeystroke()
post    $\text{result} \in keyboard(t)$  and
        $timestamp(\text{result}) > t@pre$  and
        not $\exists k \in keyboard(t)$ :
              $(timestamp(k) > t@pre$ and
               $timestamp(k) < timestamp(\text{result}))$
```

3.4 Specifying Security-relevant Properties

Under security aspects, a key requirement for a system using keyboard input and screen output is the impossibility of man-in-the-middle attacks against the keyboard and the screen. If an attacker can get in between the legitimate application and its input/output facilities, the attacker can manipulate the user at will.

There is no easy way to prevent physical man-in-the-middle attacks like, for example, covering the real keyboard with a faked keyboard as described in [2]. However, the prevention of software-based attacks with trojans, worms, viruses etc. is possible if the operating system provides means to guarantee exclusive access to the keyboard and screen. We call the process of acquiring exclusive access "locking" and the release of the lock "unlocking."

We consider information on whether screen and keyboard are locked and by which process to be part of the current configuration (i.e., the status) of the operating system. In the specification of requirements for the operating system, one has to refer to this information and other configuration details. For that purpose, we assume the relevant parts of the operating system configuration to be stored in a data structure (a class) `OSConf` with the following class attributes:

```
OSConf.screenLocked
OSConf.keyboardLocked
OSConf.ioStatus
```

`OSConf.screenLocked` and `OSConf.keyboardLocked` contain the process IDs (PIDs) of the processes locking the screen resp. the keyboard. A PID of 0 means that the resource is not locked. The third attribute `OSConf.ioStatus` can have the values `busy` and `waiting`. It indicates whether the system is busy or is

`Conf.command`	Last issued command
`Conf.commandResult`	Result of last command
`Conf.applicConf`	Application-specific part of configuration

Table 1. Configuration of an application.

waiting for input. While it is busy, all input is discarded (see comment about input buffers in Chapter 3.3).

Locking a resource is not sufficient to guarantee security. The user must also *know* which process locks a resource and whether the system is busy or not. Therefore, the operating system configuration must be shown to the user represented by a string of characters. We assume this string representation to be given by the function *OSConfString* : `OSConf` → `String` , which we do not further specify here. It must return a string that allows the user to determine the exact operating system configuration. Its actual implementation depends, for example, on the language(s) the user is supposed to understand.

We assume that the first line of the screen is reserved for information on the operating system configuration, i.e., the first line should be identical to *OSConfString*(`OSConf`).

We specify the correct display of the operating configuration resources as an invariant of `OSConf`:

```
context OSConf
inv    $stringAt(t)[0,0] = OSConfString(\texttt{OSConf})$
```

In the following we give a constraint for the operating system call for locking the screen. The calls for locking the keyboard and unlocking the resources are equivalent. Here `PID` refers to the PID of the current application.

```
context lockscreen()
post   if OSConf.screenLocked@pre = 0
       then
         OSConf.screenLocked = PID  and
         result = SCREEN_LOCKED_OK
       else
         result = SCREEN_LOCKED_BY_OTHER
       endif
```

4 Security of Interactive Applications

4.1 Overview

In Chapter 3, we showed how to specify security-relevant properties of input/output functions provided by an operating system. By ensuring and verifying these properties, certain types of software-based man-in-the-middle attacks can be prevented.

There are, however, other essential aspects of a secure software system. In this chapter, we are going to introduce a method for specifying properties of applications that are desirable both for security and usability. Namely, the following properties are considered:

1. The user is always aware of the state of the system.
2. User input is only possible if the screen output is consistent.
3. Results of user actions are communicated to the user.

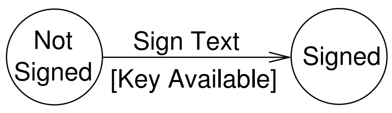

Fig. 1. State Chart Example

On an abstract level, the behavior of text-based interactive applications can be described using state charts. Edges are labeled with keystrokes, guard conditions, or both, as shown in Figure 1. In this example, the system transits from state $\boxed{\text{Not Signed}}$ to state $\boxed{\text{Signed}}$ if the command "Sign Text" is issued and the guard condition "Key Available" is satisfied. Of course, the states in such as state chart 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 in what abstract state the application is. Since, as said above, we also want the user to know what the result of the last issued command was, we define the configuration `Conf` of the application to contain—besides an application-dependend part `Conf.applicConf`—the last issued command `Conf.command`, and the result `Conf.commandResult` of that command, which can take the special valued `none` if the command is not yet completed (see Table 1).

Now, two aspects of the application have to be specified:

1. The way in which the configuration is related to screen output; and how keyboard input corresponds to commands.
2. The effect that the execution of a command has, which must implement the abstract behavior specified by the state chart.

4.2 Specification of Input/Output Behavior

For the specifiation of the first aspect (input and output), we assume the following to be given (see Table 2):

– *stateAsString*(`state`) is a string that allows the user to determine what the state of the application is.
– *resultAsString*(`commandResult`) is a string that allows the user to determine what the result of the last issued command is.
– *screenOutput*(`applicConf`) is a two-dimensional array of characters. It contains the correct screen output corresponding to `applicConf`. Its dimensions are $screenWidth()$ and $screenHeight() - 3$.
– *command*(`char`) is the command that is issued by entering `char` on the keyboard.

Name	**Description**
stateAsString	Textual representation of the state
resultAsString	Textual representation of a command result
screenOutput	Screen output for a configuration
command	Command issued by entering a character
state	State abstraction of a configuration
newState	Next state when a command is issued in a certain configuration
result	Result of a command in a certain configuration

Table 2. Functions specifying an application.

We demand that $\mathit{stateAsString}(\texttt{state})$ is shown on the second line of the screen, and $\mathit{resultAsString}(\texttt{commandResult})$ on the third line (remember that the first line is reserved for the operating system's status line), which is why $\mathit{screenOutput}(\texttt{applicConf})$ must have a height of $\mathit{screenHeight}() - 3$.

Thus, the function `updateScreen` can be specified as follows. It is the application's function for updating the screen contents (using the operating system function `setChar`).

```
context updateScreen()
post    $stringAt(t)[0,1] =$
            $stateAsString(state(\texttt{Conf.applicConf}))$  and
        $stringAt(t)[0,2] =$
            $resultAsString(\texttt{Conf.commandResult})$  and
        $\forall k \in \{3,\ldots,screenHeight()-1\}$ :
            $stringAt(t)[0,k] =$
              $screenOutput(\texttt{Conf.applicConf})[\mathrm{k}]$
```

4.3 Specification of Command Execution

For the specification of the second aspect (command execution), we assume the following to be given (see Table 2):

– $\mathit{state}(\texttt{applicConf})$ is the state abstraction of the application configuration.
– $\mathit{newState}(\texttt{applicConf}, \texttt{command})$ specifies the state transition. (It has `applicConf` as an argument and not, as one might expect, the abstraction $\mathit{state}(\texttt{applicConf})$, because it depends on guard conditions that can only be evaluated using the concrete application configuration.
– $\mathit{result}(\texttt{applicConf}, \texttt{command})$ is the result of executing `command` when the application is in configuration `applicConf`.

Now, the function `execute` can be specified. It executes a command and implements the state transition by changing the application configuration.

```
context execute()
post    state(Conf.applicConf) =
            newState(Conf.applicConf@pre,
                     Conf.command@pre)
        Conf.commandResult =
            result(Conf.applicConf@pre,
                   Conf.command@pre)
```

4.4 The Application's Main Algorithm

Now, we have everything at hand to describe how the main algorithm of the application works: First, screen and keyboard are locked. Then, in the main loop, commands are read and executed while keeping the screen updated. These steps are arranged in the following way:

– Screen and keyboard are locked immediately on program start and unlocked when the program quits. If locking the screen or the keyboard fails, the program terminates.
– Whenever the program is waiting for user input, the screen is up to date. Commands can be issued only when the system is waiting. All keystrokes entered during processing are discarded. By this we ensure that the user issues a command only when the current configuration of the system is visible on the screen.
– When processing is finished, the loop starts over again unless the user has issued the command "quit."

Pseudo for the main execution loop is given in Algorithm 1.

Algorithm 1 The application's main algorithm

```
1: if not (lockkeyboard() = KEYBOARD_LOCKED_OK) then
2:    Exit
3: end if
4: if not (lockscreen() = SCREEN_LOCKED_OK) then
5:    Exit
6: end if
7: {OSConf.screenLocked = PID and OSConf.keyboardLocked = PID}
8: repeat
9:    updateScreen()
10:   Conf.command = command(key(getkeystroke()))
11:   Conf.commandResult = none
12:   updateScreen()
13:   execute()
14: until Conf.command = QUIT
```

The consistency of the screen output follows from the algorithm and the specification of `getkeystroke`. The screen is up to date when the system is waiting for user input, and immediately after user input, and it may be inconsistent in between. Since the operating system displays status information “waiting” when the system is waiting for user input and “busy” when it is not, the user knows when the display must be consistent (whenever the system is waiting for user input). The situation would become more complicated if we used an input buffer. In that case, there is no longer a direct relationship between waiting/busy status and the consistency of screen output. It would be necessary to show an extra “consistency flag” on the screen.

5 Conclusions and Future Work

In Chapter 3 we gave a formal specification for text-based input/output functions of an operating system. This formalism can be extended to other input/output devices, e.g., card readers and graphical terminals. Additionally, we showed how to protect against software-based attacks on input/output resources. These security measurements require special functionality of the operating system. It must be able to grant processes exclusive access to input/output resources. Moreover, dedicated screen areas must be provided for information on who is locking the resources. This area must not be writable for anybody except the operating system.

The method we propose does not make any claims about what happens outside the realm of software. It cannot guarantee that an output device operates as intended, nor can it prevent tempering with the hardware of input/output devices.

In Chapter 4, we described a state-chart-based method for the formal specification of interactive applications. This formalism takes both security and usability aspects into consideration.

Our future work will go into two directions: As part of the Verisoft project (`http://www.verisoft.de`), the methods introduced in this paper are used to formally specify an email client. In Verisoft, both the operating system and the application program will be formally verified based on that specification.

The other direction of further work is to develop formal methods for the specification of applications that have richer user interfaces than a purely text based interface.

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. L. Bussard and Y. Roudier. Authentication in ubiquitous computing. In *UBICOMP 2002, Workshop on Security in Ubiquitous Computing*, Göteborg, Sweden, September 2002.
3. A. Dix and G. Abowd. Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, 11(6):334–346, 1996.
4. V. Jain. User interface description formalisms. Technical report, McGill University School of Computer Science, Montréal, Canada, 1994.
5. B. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, pages 157–202, 1982.