Real World Verification
Experiences from the Verisoft Email Client

Gerd Beuster, Niklas Henrich, Markus Wagner*
University Koblenz-Landau
{gb}|{nikhen}|{wagnermar}@uni-koblenz.de

Abstract

This paper reports our experiences developing a completely verified email client. The formal specification of the email client includes all informal requirements and security goals. Compliance to the formal specification has been proven for the complete source code. The email client is part of project Verisoft, where pervasively verified systems are developed.

1 Introduction

The goal of the Verisoft project is to create the tools and methods to allow the pervasive formal verification of computer systems, and to show that verification of real world systems is viable [Pau05]. In Verisoft, formal methods and verification technology are used throughout all aspects of system developing, including verified hardware, verified development tools, and verified operating systems and verified application programs. Four concrete systems are developed in Verisoft. Of these four systems, three are developed by or in cooperation with partners from the industry, and one is developed by the academic partners. The industry projects include an *Emergency Call System* developed in cooperation with the BMW group, a *Biometric Identification System* developed in cooperation with T-Systems, and *Hardware verification* developed in cooperation with Infineon Technologies. The academic project develops a secure email system. This paper reports our experiences developing a completely verified email client as part of the academic system.

1.1 The Academic System

The goal of the academic project is to show that common desktop technology can be formally specified and verified. For this reason, the technology used in the academic system tries to stay as close to "normal" systems, technologies, and standards as possible. The academic system is made up of different parts, as depicted in figure 1. The

*This work was 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.

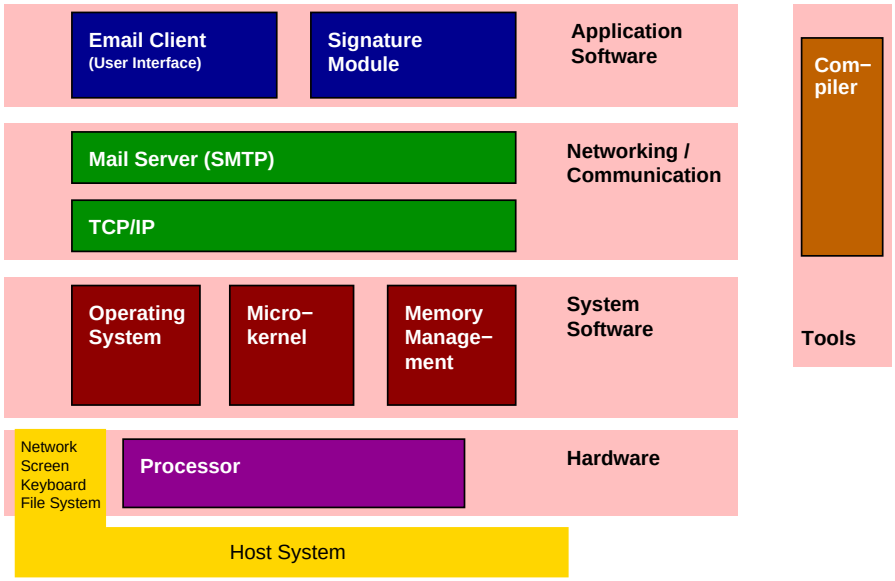

Figure 1: Components of the academic system

verified compiler compiles programs written in the C dialect C0 [LPP05]. The machine code is run on fully verified hardware (processor) [ABKS05]. Three layers of software build upon the hardware. The first layer consists of a fully verified micro-kernel, memory management unit and an accompanying operating system called *Simple Operating System* [GHLP05]. The networking and communication layer consists of a fully verified SMTP mail server using a fully verified TCP/IP stack. This allows the academic Verisoft system to interconnect with the "real world" like Intranets or the Internet. The application software sits on top of the system software and communication layer.

As part of the Academic Verisoft System, we developed a completely verified email client. The formal specification of the email client includes all informal requirements and security goals. Compliance to the formal specification has been proven for the complete source code. Each of the three main Sections 2–4 deals with one of the core results of our work on the Verisoft email client. Section 2 explains how we formally specified secure user interfaces. Section 3 describes the lessons learned from the early stages of specification, and why developing a prototype was important. Section 4 reports our experiences from verifying the Verisoft email client.

1.2 Related Projects

Another important fundamental research project in the area of verification and analysis is the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (AVACS), which is funded by the Deutsche Forschungsgemeinschaft (DFG). About 70 scientists of the Universities of Oldenburg, Freiburg and Saarbruecken as well as the Max-Planck-Institute of Computer Sciences of Saarbruecken are working on the improvement of techniques for mathematically precise verification, including the development of tools. The goal of their work is to automate safety analyses of critical embedded systems which are used for example in aircrafts, motor vehicles or

railway transportation [DHO04].

Significant prior projects are DAEDALUS and VERIFIX. The DEADALUS consortium was a research and technology development project in the Fifth Framework Programme (FP5) of the European Union. With the contributions of universities from France, Germany, Denmark, and Israel, the project developed methods and tools to support the industrial validation of critical concurrent software by static analysis and abstract testing [Gou01, CC02]. The goal of VERIFIX, another project funded by DFG, was the construction of mathematically correct compilers, which included the development of formal methods for specification and implementation of a compiler. One of the project's results was a fully verified LISP interpreter [GZ99].

2 User Interface Specification

Within the academic part of the Verisoft project, the Verisoft email client, for short *Vericlient*, provides the interface to the user. When a user accesses the academic system, he interacts with the email client. The email client itself has internal interfaces to four components: The I/O facilities (via the operating system), the SMTP server for delivery and reception of emails, and the signature module for generation and checking of signatures. For its internal operation, the email client makes use of data structures provided by the C library.

Providing a user interface is the core functionality of the email client. Vericlient provides a text-based user interface as shown in Figure 2. Since a general design goal of the Verisoft email system was to provide a secure environment for using email services, special emphasis was put onto the security of the user interface.

2.1 User Interface Security Requirements

The definition of a secure user interface is based on the common definition of secure computing as

Confidentiality	Information is available to authorized parties only.
Integrity	Neither the system nor services provided by and data processed by the system can be manipulated.
Availability	Accessibility of services and data is guaranteed.

We adapted these concepts to user interface security by restricting these definitions to the aspects involving the user interface and human-computer interaction. For Confidentiality, this means that eavesdropping on the input/output facilities must not be possible. Integrity of the user interface is guaranteed if manipulation of the user interface is not possible, i.e. if the user's assumptions about the state of the application, gained by observing and manipulating the application via the user interface, corresponds to the actual state of the application. Availability of the user interface means that an attacker can not get the user interface into a state where the full functionality is no longer accessible.

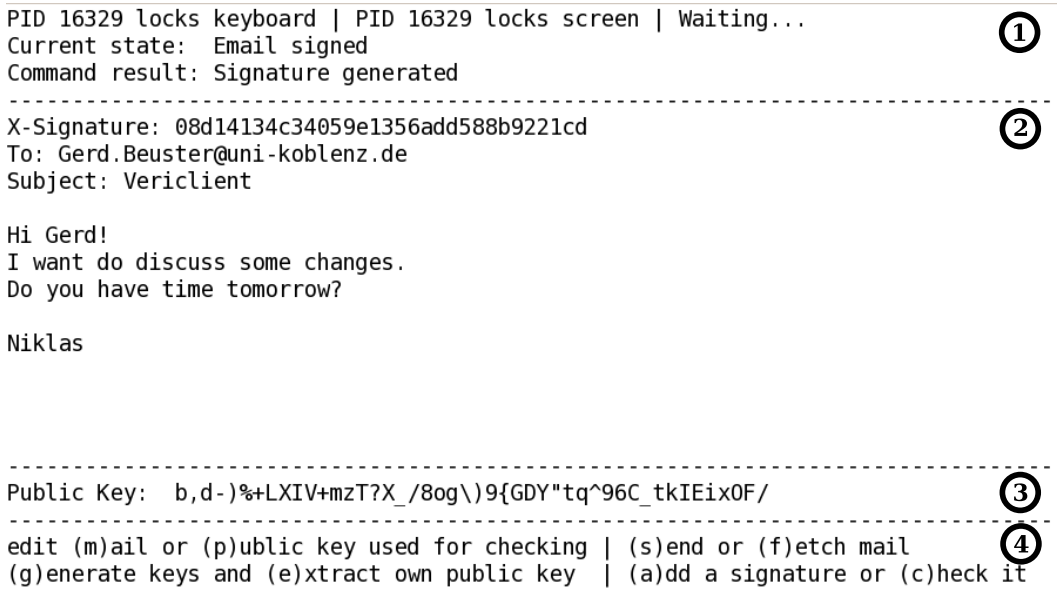

Figure 2: Vericlient prototype running: The numbers indicate the following screen areas: (1) Status / current state of the email client (2) Editing area (3) Public key (4) Commands available

Confidentiality	A third party can not gain information from observing human-computer interaction.
Integrity	Whenever the user issues a command, all relevant information, most notably the state of the program and the data processed, is shown on the screen correctly.
Availability	The functionality provided by the user interface is always accessible.

Translating these security constraints into a formal specification and writing an email client application satisfying the constraints posed a number of challenges. It turned out that a number of constraints raised by the email client development group required functionality from outside the email client. For example, neither Confidentiality, nor Integrity, nor Availability can be guaranteed if an attacker can manipulate the I/O devices. Therefore 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 [BR02]. However, the prevention of software-based attacks with Trojan horses, 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." 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. Providing (and verifying) this functionality is beyond the realm of a client application like Vericlient; it has to be provided by the operating system. Therefore, in the specification phase of the project a lot of communication with other development group was necessary. Most changes where requirement from the operating system group.

2.2 User Interface Specification

In order to formally specify and verify the security of a user interface, it was necessary to bring together formal methods, human computer interaction, and computer security. All three of them are established fields of research. There are also works combining each two of the 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. For the Verisoft email client, we had to combine all *three* fields.

Confidentiality relies mainly on the operating system, which has to ensure that eavesdropping on the application and the communication channel is not possible. Availability depends on all components involved. For the email client, it has to be shown that it is always possible to write, sign and send email, and that is always possible to receive email, check the signature, and read it. Integrity, defined as the requirement that data is displayed correctly whenever the user issues a command, is primarily the responsibility of the email client.

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, if we want to formally verify the integrity of a system, a formal definition is required.

In order to ensure integrity with formal methods, it is necessary that a) the output device provides the "right" information, b) the information is up-to-date whenever the user issues a command, and c) that the user is able to understand the information shown by the output device. For the latter, a formal user model is required. While there are some (semi-)formal methods for the description of user interfaces and human-computer interaction, these are usually not suited for automatic reasoning. Our approach was to formalize and extend the GOMS user model technique in a way that makes it suitable for modeling human-computer interaction, including potential human errors, and for automated reasoning [BB]. GOMS is a modeling technique (more specifically, a family of modeling techniques) for analyzing the complexity of interactive systems. The user's behavior is modeled in terms of *Goals*, *Operators*, *Methods* and *Selection rules*. Briefly, a GOMS model consists of methods that are used to achieve goals. A method is a sequential list of operators that the user performs and (sub)goals that must be achieved. If there is more than one method which may be employed to achieve a goal, a selection rule is invoked to determine what method to choose, depending on the context.

For the Vericlient, we assume that the user knows about the system state if he gets information about the last operation of the system ("email has been sent"), and the data on which the operation was performed (i.e. the actual email). Guaranteeing that screen output is up-to-date whenever the user issues a command is tricky, because user interfaces are inherently asynchronous. There will always be moments during the execution of the application, when the screen output does not reflect the actual state of

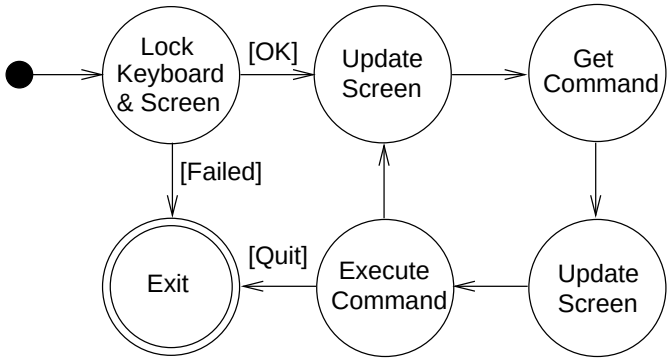

Figure 3: Statechart describing the main event loop.

the system, because only parts of the output screen have been updated, or because the system just finished an internal operation and the screen output had not been updated. The same is true for user input. Since keyboard input is usually buffered, it is possible that the user triggers actions without having seen the current screen output.

We solved the problem of asynchronous input/output by defining strict points in the main execution loop when the screen is updated, and by imposing restrictions on the input buffer. A statechart of the main execution loop is given in Figure 3.

After locking keyboard and screen (required to guarantee Confidentiality and Authenticity), the event loop receives keystrokes and executes the commands associated with the keystrokes. It also takes care of keeping the screen up-to-date. Since the screen may be inconsistent during state updates (i.e., the current screen display may not reflect the internal state of the system), the screen update function is called twice: Once before the system waits for the next keystroke, and again before command execution. In the second update, the screen area for displaying the current state shows the message "processing." When processing is finished, the loop starts over again, unless the user has issued the command "quit." Before the next command is accepted, the input buffer is emptied. This way, the user can be ensured that the screen display is consistent with the actual state of the application whenever the message "processing" is *not* shown.

The functional behavior of the email client, and thus the information to be displayed, is defined by the statechart shown in Figure 4. For example, the system transits from state Unsigned to state Signed if command "sign" was issued and the signing operation was successful. Of course, the states in such a statechart 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.

2.3 Example: Editing Mail

Not only the state of the system, but also the data has to be displayed correctly. Defining "correct" display of an email under security aspects is a challenging task. In the real world, "phishing" attacks are major form of electronic fraud [Bac05]. Many of

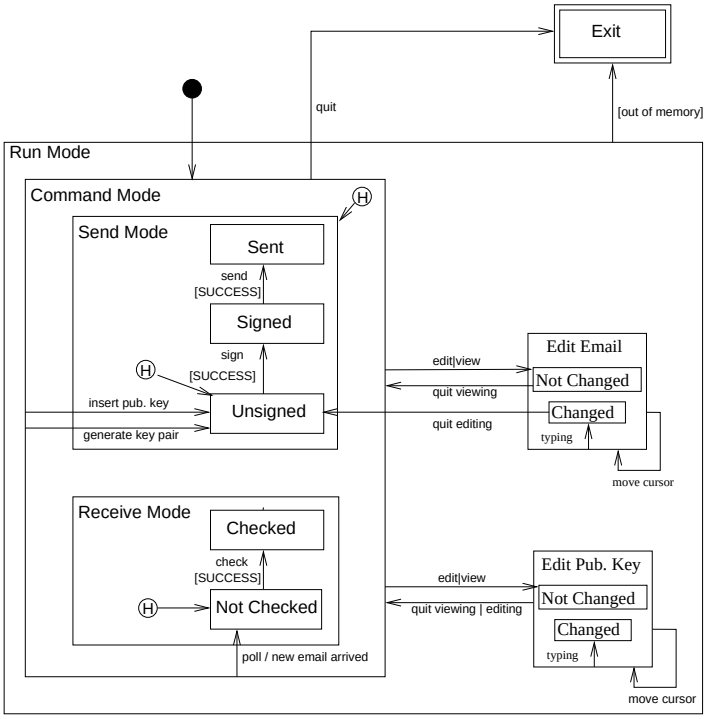

Figure 4: Statechart of email client applications. State transitions represent execution of program functions.

these attacks are based on exploitation of incorrect or ambiguous display of email messages. General concepts against these attacks are beyond the scope of this paper. For the Verisoft email client, these attacks are prevented by restricting the way emails are displayed. The Verisoft module for viewing and editing shows the pure ASCII representation of the email.

In the following, we present a short excerpt of the specification of the Verisoft email editing module. This example allows us to demonstrate how an interactive user interface component can be specified. The email viewing and editing component has the following characteristics: It is a full screen editor; the user can roam freely over the text using the cursor keys. The text edited may not fit the screen. In that case, the editor will scroll when the cursor reaches the screen borders.

The email message editing field is represented by a data structure $textEdit := (s, cx, cy, co, ro)$ with s a list of strings where each element represents a line of the text, (cx, cy) the cursor position and (ro, co) row and column offsets. If the text is larger than the size of the screen, the offsets indicate which part of the email are shown.

The part of the main execution loop's `updateScreen` responsible for showing the email (with (x, y) a position on the screen) is specified as:

$$updateScreen[y, x] = \begin{cases} s[y + ro][x + co] & \text{if } length(s) < y + ro \text{ and} \\ & \quad length(s[y + ro]) < x + co \\ \text{blank} & \text{otherwise} \end{cases}$$

The specification of main execution loop function `execute` for email editing is defined by the OCL specification given in Table 1. Note that `INSERT_CHAR` represents the set of all printable characters.

context `execute(cmd, textEdit)`
pre `cmd` $\in$ { `CURSOR_LEFT`, `CURSOR_RIGHT`, `CURSOR_UP`, `CURSOR_DOWN`, `INSERT_CHAR`, `DELETE_CHAR`, `QUIT` }
post **if** `cmd` = `CURSOR_LEFT` **then**
`textEdit` = $cursorLeft$(`textEdit@pre`) **and**
result = `CURSOR_MOVED`
else if `cmd` = `CURSOR_RIGHT` **then**
`textEdit` = $cursorRight$(`textEdit@pre`) **and**
result = `CURSOR_MOVED`
else if `cmd` = `CURSOR_UP` **then**
`textEdit` = $cursorUp$(`textEdit@pre`) **and**
result = `CURSOR_MOVED`
else if `cmd` = `CURSOR_DOWN` **then**
`textEdit` = $cursorDown$(`textEdit@pre`) **and**
result = `CURSOR_MOVED`
else if `cmd` $\in$ `INSERT_CHAR` **then**
`textEdit` =
$insertChar$(`cmd, textEdit@pre`) **and**
result = `CHAR_INSERTED`
else if `cmd` = `DELETE_CHAR` **then**
`textEdit` = $deleteChar$(`cmd, textEdit@pre`) **and**
result = `CHAR_DELETED`
else
result = `QUIT`
end if

Table 1: Command execution function

The auxiliary functions describing the effects of cursor movements and inserting/deleting characters are straightforward. As an example, we only provide a definition for $cursorRight$:

$$cursorRight(a) = (s, cx', cy', co', ro')$$

with

$$cx' = \begin{cases} cx+1 & \text{if } a.cx+1 < length(a.s[a.cy]) \\ 0 & \text{if } a.cx+1 \geq length(a.s[a.cy]) \text{ and} \\ & \quad a.cy+1 < length(a.s) \\ cx & \text{otherwise} \end{cases}$$

$$cy' = \begin{cases} cy+1 & \text{if } a.cx+1 \geq length(a.s[a.cy]) \text{ and} \\ & \quad a.cy+1 < length(a.s) \\ cy & \text{otherwise} \end{cases}$$

$$co' = \begin{cases} co+1 & \text{if } cx' = co + \texttt{screenWidth} \\ 0 & \text{if } cx' = 0 \\ co & \text{otherwise} \end{cases}$$

$$ro' = \begin{cases} ro+1 & \text{if } cy' = ro + \texttt{screenHeight} \\ ro & \text{otherwise} \end{cases}$$

The correctness of the specification was ensured in two ways: It has been shown that the editing component specification allows to enter an arbitrary text, and it has been shown that the order of characters is preserved when an arbitrary text is shown by the email client. While these two refinement proofs ensure basic correctness of the editing component, they do not capture the interactive behavior of the editing component. A prototypical implementation (see also Section 3) was used to ensure that the specification of the editing component follows the user's intuition about an interactive editor.

We have shown how user interface security is formalized and specified for the Verisoft email client. In the next Sections, we report our experiences from implementing and verifying the Verisoft email client specification.

3 Specification and Prototypical Implementation

The most direct way to develop a fully formally specified and verified application would be to start by writing a formal specification. From this, one would either write an implementation and proof its correctness, or refine the specification down to the implementation level, generating the correctness proofs on the way.

Because of the character of the Verisoft project, this approach could not be followed strictly. Since Verisoft started largely from scratch, all parts of Verisoft, including the specification languages, the calculus, and the system components the email client relies on, were developed in parallel. Over the course of the projects, more and more tools were finished, the calculus and languages got fixed, and specifications and implementations of other components became available. This led to a somewhat different design model.

At the beginning of the project, we started by informally defining the global design goals of the Verisoft email client. We developed a semi-formal specification using OCL and statecharts. Based on this specification, a prototype was developed. The prototype served two purposes. First, it allowed us to test the informal specification. Verifying software is even more costly in terms of time and money than normal software development. Therefore we wanted to ensure that the specification of the email clients did not contain design errors. The main functionality of the email client is to provide the user interface for other system parts, like the SMTP component and the signature component. The design of the user interface of the email client must not only comply to security requirements ("the email is shown correctly"), it must also comply to the user's expectation about "proper behavior" of a user interface. The prototype allowed us to test our email client design before finalizing the formal specification.

For the development of the prototype, we used ordinary C, not C0, and we used the *ncurses* library[Str91] instead of Verisoft's operating system functionality for screen output. The client compiles in a standard Linux environment. This has the advantage, that we could provide a working prototypical version of the email client without being

dependent on other system parts. C0 is a subset of C. By restricting our coding style to the constructs allowed by C0, we learned how to deal with the limitations inherent to the language. The prototype was more an evolutionary prototype than a throwaway prototype. After more and more libraries and the final definition of C0 became available, we adjusted the prototype step-by-step until it become the final implementation of Vericlient.

The development of a prototype was crucial. We gained several insights on the run-time behavior of the Vericlient and it helped to improve and even to correct the specification. With the help of the prototype, we detected glitches in the user-interface (for example, characters were inserted at the wrong place in email messages because of an off-by-one error) as well as errors in the statechart specification of Vericlient's overall behavior. Since these errors were in the specification, they would not have been noticed until Vericlient would have been fished.

Errors found at the earlier stages are easier to correct than errors found in a late stage. This is even more the case for a verification projects, where errors in the specification may require the component to be proved again. A prototype helps to identify errors or wrong decisions in the early design and specification phase of a project. Since it was possible to evolutionary develop the final version from the prototype, the work spent on the prototype was efficiently integrated into the project.

Over the course of the project, more functionality from other parts of the academic Verisoft project became available. The Vericlient prototype gradually turned into the final implementation of the Verisoft email client by integrating these modules once they became available. The prototypical code developed to run in a normal Linux environment was maintained in parallel to the code developed for the Verisoft environment. The advantage of this approach was that from the very beginning of the project, a working version existed and changes in the specification could be tested. From our experience, the little extra work required to develop two versions in parallel pays out enormously. Since a working version of the email client existed from the very beginning, we always knew precisely if the interfaces and specifications provided by other modules fit into the email client, or if changes were required.

4 Verification and Integration

The goal of Verisoft is the pervasive verification of both system hardware and software. In order to allow integration of the components developed in different parts of the project, a common set of formal methods and tools is required. As a general design decision, the Verisoft participants agreed to use Isabelle/HOL [NPW02] as the main verification tool. Norbert Schirmer developed an Isabelle theory for the verification of C0 programs in Isabelle/HOL. The core of this theory is a Verification Condition Generator for translation of specifications and code into HOL [Sch05]. C0 is a subset of C with some limitations for easier verification. Side effects are not allowed in expressions, and there can only be one return statement in each function. Pointers are typed. Pointer arithmetic is forbidden, and arrays can not be allocated dynamically. The verified compiler developed by Leinenbach et al.[LPP05] translates C0 programs into machine code and into a format suitable for input into the Isabelle system.

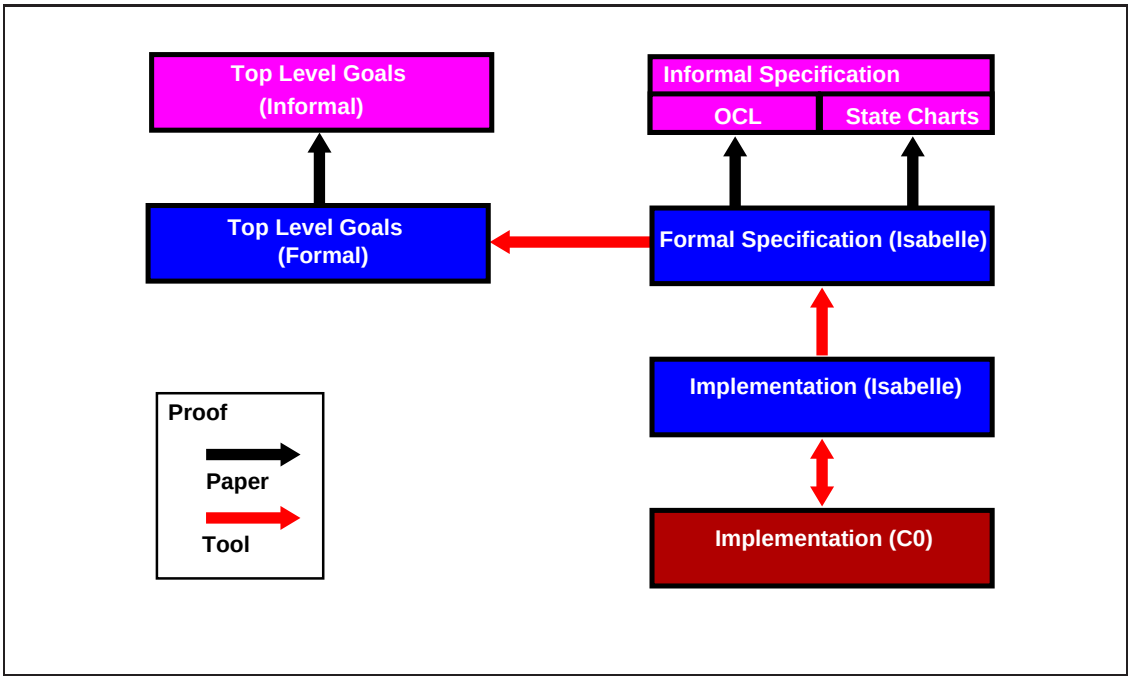

Figure 5: Types of models used in the specification of the Verisoft email client

The verification process of the Verisoft email client is depicted in Figure 5. Based on the informal specifications, formal Isabelle specifications were developed. The prototypical C implementation of the email client was adapted to C0, and correctness of the automatically generated Isabelle translation of the code was proven. Independent of this, formal definitions of the email client security requirements given in Section 2.1 were developed. From the functional correctness of the email client specification, compliance to the top level verification goals was deduced.

4.1 Verifying the Email Client

An important lesson from verifying the Verisoft email client is that *implementation follows verification*. When there was a problem in verifying a piece of code, we did not hesitate to change the implementation in order to make it easier to verify. Since the tool for automatic translation from C0 to the Isabelle representation of C0 was developed in Verisoft, and was therefore not available at the begin of the project, we started by manually translating parts of the code and verifying it. We learned that this goes very well with verification of the large and complex functions of the Verisoft email client. In order to verify a large function, we started with small pieces of the code and the specification. Once these fractions where verified successfully, more parts of the specification and of the code were added. The final implementation of the Verisoft email client consists of about 2800 lines of code. Verifying the Verisoft email client required an Isabelle proof script of twice the size of the code. Compared to typical text book examples, the proofs were rather simple. The Verisoft email client does not use data structures more complex than lists of strings, and operations on the data structures do

not involve recursion etc.

4.2 Interfaces to other components

The Verisoft email client has interfaces to a number of other components of the Verisoft system: The operating system provides system calls to the I/O devices. Remote procedure calls are used to pass mail to and from the SMTP server, and to the cryptographic module for signing and checking signatures. The C0 library provides essential data structures used by the email client. Since all of these components were developed in parallel to the Verisoft email client, their specifications and implementations became available over the course of the project.

In Section 3, we explained how we solved the problem of not having implementations of other components available at the begin of the project. For the specification and verification part, our approach to deal with this problem depended on the type of missing component.

Some of the components are essentially "black boxes" for the email client. For verifying the functional properties of the email client, the actual specification of the signature component and the SMTP server are irrelevant. For the email client, we just have to show that the SMTP and cryptographic functionality is executed at certain points during the execution of the email client. Specification of these components is needed only in the last step, when the email system is integrated and the top level goals of the email system are proven. Therefore, we were able to specify and verify the email client independent of these modules.

The situation was different for the data structures. For the specification and correctness proofs about the email client, we had to make use of the specification of the string and list data structures. Here, the Isabelle/HOL verification environment was beneficial. HOL provides native string and data structures. In the first phases of the project, we replaced the C0 data structures by their corresponding HOL data structures. Since the C0 data structures are defined as refinements of the native HOL data structures, integration of the real data structures was fairly straightforward. Only some additional pre-conditions had to be changed in order to take into account the cases where the behavior of the C0 data structure differs from the behavior of the HOL data structures. For example while HOL data structures do not have upper bounds, the length of C0 data structures is limited.

It turned out that this two-step approach did not cause a significant overhead, because the old proofs, conducted with the HOL data structures, did not become obsolete. They just had to be extended to deal with the additional constraints of the C0 data structures.

In conclusion, using non-verified, interface compatible libraries built-in into the tools and replacing them later on with their verified counterpart turned out to be a good approach. It allowed us to start verifying at a time when the final libraries were not available.

4.3 Integration

Different parts of the Verisoft academic system, and even different part of the Verisoft email client, use different kinds of formal methods. Parts are specified in terms of pre-

and post-conditions. These were verified in Hoare calculus. Other parts rely on temporal properties. These were specified in temporal logics and proven by model checking. In order to integrate both aspects, the specification and implementation of the email client was split in two parts: The temporal properties were modeled as the state transition diagram shown in Figure 4. The functional properties were embedded in this state transition diagram. Each state transition represented a function call specified in Hoare logics. Since all states are represented explicitly in Vericlient, and the main execution loop executes the statechart, integration was achieved by showing that each functional call executes the state transitions defined in the statechart.

5 Conclusions

Completely verifying the Verisoft email client posed a number of challenges:

- Other parts of the system, including the specification and implementation language and the calculus, were not available when the project started.
- Interactive user interfaces and secure human-computer interaction had to be specified and verified.
- Theorems proven in different formal methods had to be integrated.

All of these problems were solved. Implementing a prototype and using prototypical specifications of the other components proved to be a viable solution for the problem that not all system parts were available in the beginning, a situation quite typical for large-scale academic and industrial projects. Starting with a prototype, we could test the viability of our specification and start verifying without having all other system parts available. When more and more parts of the system became available, most of the work spent on the prototype could be reused.

Formally specifying secure human-computer interaction required genuine scientific work. By formalizing the user model and by adapting and formalizing secure computing for human-computer interaction, it was possible to verify user interface security with the formal methods employed in the Verisoft project. Here, it was beneficial that not all other parts of the Verisoft academic system were already available at the start of the project, because the requirements for a secure user interface directly affected the specification of other parts, like the I/O device interface of the operating system.

For the integration of results achieved by Hoare calculus verification with results achieved by model checking, it was a big advantage to have both methodologies integrated into Isabelle/HOL. This way, the same statechart specification could be used for model checking, and in the definition of pre- and post-conditions of the individual functions.

References

[ABKS05] Nathaniel Ayewah, Sven Beyer, Nikhil Kikkeri, and Peter-Michael Seidel. Challenges in the formal verification of complete state-of-the-art processors. In *International Conference on Computer Design*, San Jose, 2005.

[Bac05] Daniel Bachfeld. Nepper, Schlepper, Bauernfänger — Risiken beim Online-Banking. *c't magazin für Computertechnik*, pages 148–153, 2005.

[BB] Bernhard Beckert and Gerd Beuster. A method for formalizing secure user interfaces. In *Submitted to Eighth International Conference on Formal Engineering Methods (ICFEM 2006)*.

[BR02] L. Bussard and Y. Roudier. Authentication in ubiquitous computing. In *UBICOMP 2002, Workshop on Security in Ubiquitous Computing*, Göteborg, Sweden, September 2002.

[CC02] Patrick Cousot and Radhia Cousot. Modular static program analysis. In N. Horspool, editor, *Proceedings of the International Conference on Compiler Construction (CC 2002)*, LNCS 2304, pages 159–178, Grenoble, France, April 6–14 2002.

[DHO04] W. Damm, H. Hungar, and E.-R. Olderog. On the verification of cooperating traffic agents. In F.S. de Boer, M.M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. FMCO '03: Formal Methods for Components and Objects*, LNCS 3188, pages 78–110, 2004.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In *Proceedings, 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, LNCS 3603, pages 2–16. Springer, 2005.

[Gou01] Éric Goubault. Static analyses of floating-point operations. In P. Cousot, editor, *SAS'01*, LNCS 2126, pages 233–258, Paris, July 2001.

[GZ99] Gerhard Goos and Wolf Zimmermann. Verification of compilers. In *Correct System Design*, pages 201–230, 1999.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a c0 compiler. In *Proceedings, 3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)*, Koblenz, Germany, 5–9 September 2005.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pau05] Wolfgang Paul. Towards a worldwide verification technology. In *Proceedings of the Verified Software: Theories, Tools, Experiments Conference (VSTTE 2005)*, Zurich, Switzerland, October 2005.

[Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452, pages 398–414, 2005.

[Str91] John Strang. *Programming with curses*. O'Reilly & Associates, Inc., 1991.