1999

Formal Development of Secure Email

Dan Zhou
Syracuse University

Joncheng C. Kuo
Syracuse University

Susan Older
Syracuse University, sbolder@syr.edu

Shiu-Kai Chin
Syracuse University, skchin@syr.edu

Formal Development of Secure Email*

Dan Zhou Joncheng C. Kuo Susan Older Shiu-Kai Chin
Department of Electrical Engineering and Computer Science
Syracuse University
{*danzhou,ckuo,sueo,chin*}*@cat.syr.edu*

Abstract

Developing systems that are assured to be secure requires precise and accurate descriptions of specifications, designs, implementations, and security properties. Formal specification and verification have long been recognized as giving the highest degree of assurance. In this paper, we describe a software development process that integrates formal verification and synthesis. We demonstrate this process by developing assured sender and receiver C++ code for a secure electronic mail system, Privacy Enhanced Mail. We use higher-order logic for system-requirements specification, design specifications and design verification. We use a combination of higher-order logic and category theory and tools supporting these formalisms to refine specifications and synthesize code. Much of our work is applicable to other secure email protocols, as our development is parameterized, component-based, and reusable.

1. Introduction

Systems with security requirements typically must operate with a high degree of confidence; we must be able to demonstrate that these systems satisfy security requirements in addition to functional requirements. Formal methods are useful in high assurance design and implementation of secure software systems [7, 4], because they increase the clarity of requirements, identify hidden assumptions that the system must operate on, and certify the consistency of requirements and the correctness of designs, among other benefits [13]. The challenge is to combine formal analysis and code synthesis in a practical process acceptable to software engineers.

In this paper we address the problem of building a secure email system where the high-level security requirements are accounted for in even the lowest level of implementation. The particular secure email system we focus on is Privacy Enhanced Mail (PEM) [12]. It is representative of other email systems such as PGP [15] and NSA's MISSI system [3], and the methods we describe are applicable to those systems as well. We chose PEM because it has gone through rigorous review as an Internet standard, it is publicly available, and it is similar to MISSI.

We apply formal methods to all key phases of the software-development life cycle by integrating existing tools: the higher-order logic theorem prover HOL [8] and the synthesis tool SPECWARE which is based on higher-order logic and category theory [14]. We formally specify the system requirements, specify and verify the system design, perform stepwise refinement on the design specifications, and then compose these refinements to generate code that is correct by construction.

In this work, higher-order logic is used for specification, verification, and synthesis. Top-level security properties and protocols are defined in HOL. The protocols are verified to satisfy the required security properties. The protocols are instantiated by adding specific data structures and operations; these instantiations are verified to be correct within HOL. The verified design specifications are then translated into SPECWARE specifications. These specifications are refined to C++ code through stepwise refinements and through the composition of these refinements.

The rest of this paper is organized as follows. Section 2 describes our formal development process. Section 3 gives an overview of PEM and the security services that it provides. Section 4 shows an example of how a security property is defined and verified in HOL. Section 5 illustrates how the highly assured design of the previous example is refined into implementations. We conclude in Section 6.

2. High Assurance Development Process

Highly assured systems can be built using a formal development process. In any type of software development process, there are at least four key stages: requirement analysis, design, implementation and verification. To produce highly

*This research was sponsored in part by Air Force Research Contracts *F30602-97-C-0310* and *F30602-98-1-0063* and by the New York State Center for Advanced Technology in Computer Applications and Software Engineering.

assured software, we utilize formal support for each key stage. We outline the proposed formal development process next.

2.1. Formal development process

The ultimate goal of high-assurance system development is producing code that satisfies desired properties. Accomplishing this goal requires two items: (1) correct system specifications that satisfy the desired properties, and (2) the valid refinement of specifications into code. To this end, we employ a combination of higher-order logic and category theory. Higher-order logic is used for verification. Category theory provides the mechanism for refining specifications into assured code and (more generally) for component-based design and synthesis.

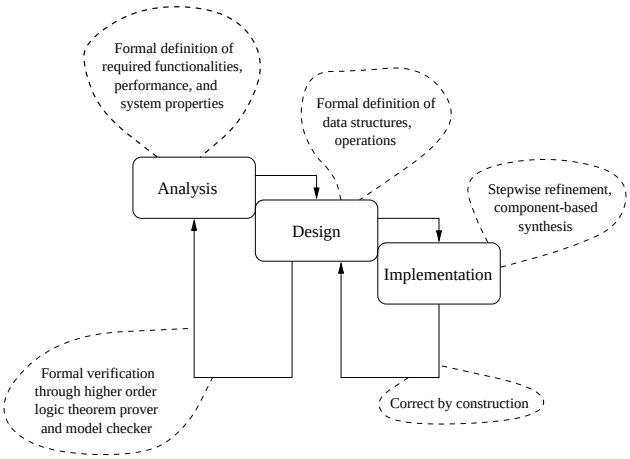

Figure 1. Software development process

Figure 1 illustrates the development process. We add formal support using higher-order logic for requirement analysis, design, implementation, and verification. The use of higher-order logic allows us to relate the products of each stage rigorously.

During the synthesis phase, we use stepwise refinements from the verified design specifications to yield lower-level specifications. These lower-level specifications are in turn refined until we arrive at a specification that maps directly to code.

Figure 2 shows the steps involved in synthesis phase.

A design specification is typically composed from smaller specifications. We refine each of the component specifications using stepwise refinement and then compose the individual refinements to arrive at an implementation for the composite design specification. The decomposition

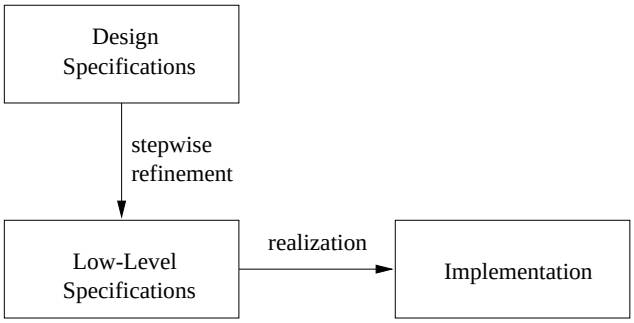

Figure 2. Synthesis phase

of a problem into smaller pieces is done by software engineers; our process does not replace this part of the human input. Because system specifications and refinements are composed through basic specifications and refinements respectively, this paradigm supports component based design and synthesis.

2.2. Tools

The process is instantiated into a concrete process by using specific tools.

We use the higher-order logic theorem prover HOL [8] for system-requirements specification, system-design specification, and design verification. Higher-order logic provides a version of predicate calculus that allows variables to range over functions and predicates. We choose HOL because of its expressiveness, extensive libraries, open construction, and strong typed implementation that lends itself to being trustworthy.

We use Kestrel's synthesis tool SPECWARE [14] for code generation. SPECWARE is a tool that supports the design, development and automated synthesis of correct-by-construction software. It is based on category theory, the theory of algebraic specifications, refinements, and composition of refinements. We choose SPECWARE because of its use of higher-order logic and categorical composition and its code-generation capabilities.

Our formal development process does not limit our choice to either HOL or SPECWARE. Any higher-order logic theorem prover could be used in place of HOL. Likewise, a different synthesis tool based on category theory and algebraic specifications could be substituted for SPECWARE.

```
-----BEGIN PRIVACY-ENHANCED MESSAGE-----
Proc-Type: 4,ENCRYPTED
Content-Domain: RFC822
DEK-Info: DES-CBC,BFF968AA74691AC1
Originator-Certificate:
 MIIBlTCCAScCAWUwDQYJKoZIhvcNAQECBQAwUTELMAkGA1UEBhMCVVMxIDAeBgNV
  ......
Issuer-Certificate:
 MIIB3DCCAUgCAQowDQYJKoZIhvcNAQECBQAwTzELMAkGA1UEBhMCVVMxIDAeBgNV
  ......
MIC-Info: RSA-MD5,RSA,
 UdFJR8u/TIGhfH65ieewe2lOW4tooa3vZCvVNGBZirf/7nrgzWDABz8w9NsXSexv
  ......
Recipient-ID-Asymmetric:
 MFExCzAJBgNVBAYTAlVTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cml0eSwgSW5j
  ......
Key-Info: RSA,
 O6BS1ww9CTyHPtS3bMLD+L0hejdvX6Qv1HK2ds2sQPEaXhX8EhvVphHYTjwekdWv
  ......

qeWlj/YJ2Uf5ng9yznPbtD0mYloSwIuV9FRYx+gzY+8iXd/NQrXHfi6/MhPfPF3d
jIqCJAxvld2xgqQimUzoS1a4r7kQQ5c/Iua4LqKeq3ciFzEv/MbZhA==
-----END PRIVACY-ENHANCED MESSAGE-----
```

Figure 3. A sample PEM message

3. Overview of PEM

In this paper, we describe the application of our development process to the development of a Privacy Enhanced Mail (PEM) system. This system is representative of other secure email systems such as PGP [15] and NSA's Multilevel Information Systems Security Initiative (MISSI) [3].

PEM adds privacy, source authentication, message integrity, and non-repudiation to plaintext email. It provides end-to-end security, assuming the underlying communication network is insecure. It is documented in four *Request for Comments* (RFC) documents: RFC 1421 [12] describes message encryption, authentication procedures, and formats; RFC 1422 [11] describes certificate-based key management; RFC 1423 [1] describes algorithms; and RFC 1424 [10] describes key certification.

PEM supports several common security properties [2]: **privacy**, the assurance to the sender and recipient that no one but the intended recipient can read the message; **authentication**, the assurance to the recipient of the sender's identity; **integrity**, the assurance to the recipient that the message has not been altered since being transmitted by the sender; and **non-repudiation**, the assurance to the recipient that she can prove to a third party that the sender was indeed the originator of the message (i.e., the sender cannot deny sending the message). We have previously defined all these properties in higher-order logic [5, 6].

Figure 3 shows an example of a PEM message; each message is encapsulated in a plaintext email message. There are five types of PEM messages: (1) *ENCRYPTED*, (2) *MIC-CLEAR*, (3) *MIC-ONLY*, (4) *CRL*, and (5) *CRL-RETRIEVAL-REQUEST*. The type of a PEM message determines the structure of the message as well as the protocol for processing the message. The format for each of these messages varies slightly depending on whether public- or secret-key cryptography is being used.

Each PEM message contains a header in addition to the text message itself. The header contains several fields that identify the message type and provide information about the message and the cryptographic functions applied to the message.

Among the header fields of interest is MIC-Info, the message integrity field. MIC-Info provides information necessary for checking the integrity of a message. This field has three subfields: in order, they contain (1) the (name of the) hash algorithm used to generate the message digest; (2) the (name of the) algorithm used to sign or encrypt the digest, depending on whether the protocol is using public-key

or secret-key cryptography; and (3) the message integrity code (MIC). The MIC functions like a secure checksum on the message text.

For the remainder of this paper, we shall focus only on the public-key variant of MIC-CLEAR messages, where the message text is sent in the clear (i.e., unencrypted and unencoded) with its associated message integrity code.

The processes that senders use to create MIC-CLEAR messages and that receivers use to check the integrity of MIC-CLEAR messages are given by a security protocol. Figure 4 shows the sequence of operations used to create messages to send and to check the received messages. For example, to create a MIC-CLEAR message, the sender combines the plaintext message with the MIC, where the MIC is the signed message digest of the mail-message content. To check the integrity of a MIC-CLEAR message, the recipient must determine the appropriate hash and signature verification algorithms to use, apply them to the message text, and verify the result against the MIC. This security protocol is concerned only with the sequence of operations, not with the actual structure of messages.

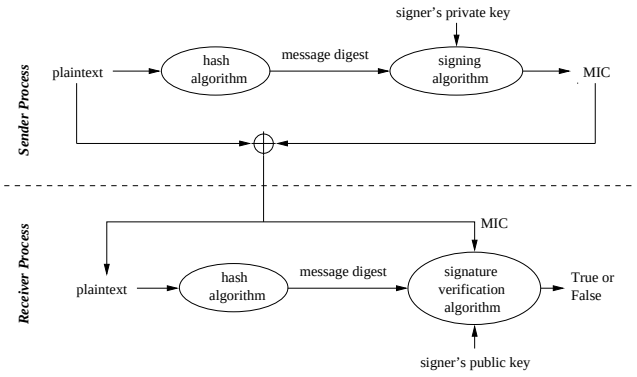

Figure 4. Protocol for message-integrity checking

In the next two sections, we describe how to design and synthesize assured code for implementing the MIC-CLEAR message structures and protocols.

4. Specification and Verification of Message Integrity

In previous work, we formally specified the security properties desired by PEM using HOL theories [5]. We formally specified mail message structures and operations for PEM ENCRYPTED and MIC-CLEAR messages. We also formally verified that PEM provides privacy, integrity, source authentication and source non-repudiation. Because that work provides a necessary input to the synthesis phase, we reiterate the basis approach to specification and verification in this section. We focus on the property of message integrity for MIC-CLEAR messages.

We use standard predicate calculus notation. The symbols $\wedge, \vee, \supset$ denote *and, or*, and *implication*, respectively, while $\forall$ and $\exists$ denote the *universal* and *existential quantifiers*. The notation $cond \rightarrow t_1 | t_2$ denotes the conditional *if cond then* t_1 *else* t_2, and $\vdash t$ indicates that the formula t is a *theorem*. Definitional extensions to HOL are denoted by $\vdash_{def}$.

4.1. Specification of MIC-CLEAR messages

A MIC-CLEAR message is specified simply as a tuple $\langle Pkey, MIC\text{-}Info, Message \rangle$ comprising the sender's public key (*Pkey*), additional MIC information (*MIC-Info*), and the message itself. In turn, *MIC-Info* is a tuple $\langle Hash\text{-}ID, Sign\text{-}ID, MIC \rangle$ containing a hash-algorithm id, a signing-algorithm id, and the MIC. This simplification retains the essential information needed to retrieve a message with security protection and is still complex enough to exemplify component-based design and synthesis concepts.

The specification also defines accessor functions that retrieve the values of the individual fields of a MIC-CLEAR message, as well as selector functions that select the hash function, the signature-generation function, and the signature-verification function to be used. A portion of the specification for MIC-CLEAR messages appears in Figure 5.

4.2. Generic integrity checking

Functionally, the integrity checking of mail messages is a procedure that takes the message digest of a received message and uses the sender's public key to verify the received MIC against the message digest. At this level of description, the integrity check is independent of the message structure and thus can be specified by the following definition in HOL:

$\vdash_{def}$ $\forall$*verify hash message mic ekey.*
is_Intact verify hash message mic ekey =
verify (*hash message*) *mic ekey*

Intuitively, the predicate *is_Intact* should evaluate to **true** if and only if the transmitted and received messages are deemed to be the same, according to the hash function. This property holds under the following assumptions:

- The MIC field of the transmitted message is the encrypted message digest.
- The received MIC is the same as the transmitted MIC.

```
Definitions:
  get_MIC_hash ⊢def ∀x. get_MIC_hash x = FST (REP_MIC_info x)
  get_MIC_sign ⊢def ∀x. get_MIC_sign x = FST (SND (REP_MIC_info x))
  get_MIC_mic ⊢def ∀x. get_MIC_mic x = SND (SND (REP_MIC_info x))

  MIC_hash_select
  ⊢def ∀x.
     MIC_hash_select x =
     ((get_MIC_hash x = RSA_MD2) → fRSA_MD2 | fRSA_MD5)
  MIC_sign_select
  ⊢def ∀x.
     MIC_sign_select x =
     ((get_MIC_sign x = DES_EDE)
      → sDES_EDE
      | ((get_MIC_sign x = DES_ECB) → sDES_ECB | sRSA))

  (* retrieves MIC−Info field from a MIC−CLEAR message *)
  get_MIC_Info
  ⊢def ∀x. get_MIC_Info x = FST (SND x)
  (* retrieves sender's public key from a MIC−CLEAR message *)
  get_public_key
  ⊢def ∀x. get_public_key x = FST x
  (* retrieves plaintext message from a MIC−CLEAR mesasge *)
  get_message
  ⊢def ∀x. get_message x = SND (SND x)

Theorems:
  get_MIC_hashid_CASES
  ⊢ ∀x. (get_MIC_hash x = RSA_MD2) ∨ (get_MIC_hash x = RSA_MD5)
  get_MIC_signid_CASES
  ⊢ ∀x.
     (get_MIC_sign x = DES_EDE) ∨
     (get_MIC_sign x = DES_ECB) ∨
     (get_MIC_sign x = RSA)
```

Figure 5. HOL specification for MIC-CLEAR messages

- The signature of a specific message can be verified through the signer's public key *ekey*.

The following correctness theorem shows that the integrity check satisfies the proceeding property. Note that the assumptions appear as antecedents in the implication, and *dkey* represents the sender's private key.

```
is_Intact_Correct =
⊢ ∀verify sign hash txmessage rxmessage
   txmic rxmic ekey dkey.
   (txmic = sign (hash txmessage) dkey) ⊃
   (rxmic = txmic) ⊃
   (∀m1 m2. verify m1 (sign m2 dkey) ekey
         = (m1 = m2)) ⊃
   ((hash rxmessage = hash txmessage)
    = is_Intact verify hash rxmessage rxmic ekey)
```

This theorem is easily proved using the definition of *is_Intact* and the antecedents of the implication.

4.3. Integrity checking of MIC-CLEAR messages

To define message integrity checking for a particular message structure, we instantiate the parameters in the preceding generic integrity check with information contained in the header of a particular message. We define accessor functions to retrieve particular fields of a message and selector functions to select cryptographic functions given algorithm IDs. For example, the integrity checking function for MIC-CLEAR messages is as follows:

```
⊢def MIC_CLEAR_is_Intact mic_clear_msg =
  let micInfo = get_MIC_Info mic_clear_msg
     and ekey = get_public_key mic_clear_msg
  in
  is_Intact (MIC_sign_select micInfo)
   (MIC_hash_select micInfo)
   (get_message mic_clear_msg)
   (get_MIC_mic micInfo)
   ekey
```

The integrity check for MIC-CLEAR message (*MIC_CLEAR_is_Intact*) is verified to satisfy a similar correctness theorem as the generic integrity check *is_Intact*:

```
MIC_CLEAR_is_Intact_Correct =
 ⊢ ∀mic_clear_msg txmessage txmic dkey.
   let micInfo = get_MIC_Info mic_clear_msg
      and ekey = get_public_key mic_clear_msg
   in
   let hash = MIC_hash_select micInfo
      and verify = MIC_verify_select micInfo
      and sign = MIC_sign_select micInfo
      and rxmessage = get_message mic_clear_msg
      and rxmic = get_MIC_mic micInfo
   in
   (txmic = sign (hash txmessage) dkey) ⊃
   (rxmic = txmic) ⊃
   (∀m1 m2. verify m1 (sign m2 dkey) ekey
         = (m1 = m2)) ⊃
   ((hash rxmessage = hash txmessage)
    = MIC_CLEAR_is_Intact mic_clear_msg)
```

This theorem (*MIC_CLEAR_is_Intact_Correct*) is identical to the general correctness theorem *is_Intact_Correct*, except that (1) the received mail's plaintext message content, the MIC, and the sender's public key are retrieved from the received MIC-CLEAR mail message (*mic_clear_msg*), and (2) the hash function, and the signature generation and verification functions are selected bases on the information provided in *mic_clear_msg*.

This theorem is proved using the definition *MIC_CLEAR_is_Intact* and the theorem *is_Intact_Correct*.

HOL Specification

```
Type constants:
  algid 0
  asymsignmic 0
  MIC_info 0

Term constants:
  is_MIC_info (Prefix)   :algid # algid # asymsignmic -> bool
  REP_MIC_info (Prefix)
    :MIC_info -> algid # algid # asymsignmic
  MIC_Info (Prefix)
    :algid # algid # asymsignmic -> MIC_info
  get_MIC_hash (Prefix)   :MIC_info -> algid

Definitions:
  is_MIC_info
  ⊢ ∀x.
      is_MIC_info x =
      ((FST x = RSA_MD2) ∨ (FST x = RSA_MD5)) ∧
      ((FST (SND x) = DES_EDE) ∨
       (FST (SND x) = DES_ECB) ∨
       (FST (SND x) = RSA))

  MIC_info_TY_DEF
  ⊢ ∃rep. TYPE_DEFINITION is_MIC_info rep

  MIC_info_ISO_DEF
  ⊢ (∀a. MIC_Info (REP_MIC_info a) = a) ∧
    (∀r. is_MIC_info r = REP_MIC_info (MIC_Info r) = r)

  get_MIC_hash
  ⊢ ∀x. get_MIC_algid x = FST (REP_MIC_info x)
```

SPECWARE Specification

```
spec MIC_INFO is
  sorts Algid, Asymsignmic, Temp_MIC, MIC_info
  sort-axiom Temp_MIC = (Algid, Algid, Asymsignmic)
  sort-axiom MIC_info = Temp_MIC | is_MIC_info?

  % define is_MIC_info check
  op is_MIC_info? : Temp_MIC -> Boolean
  definition of is_MIC_info? is
    axiom (iff (is_MIC_info? x)
           (and (or (equal ((project 1) x) RSA_MD2)
                    (equal ((project 1) x) RSA_MD5))
            (or (equal ((project 2) x) DES_EDE)
             (or (equal ((project 2) x) DES_ECB)
              (equal ((project 2) x) RSA)))))
  end-definition

  % define get_MIC_hash
  op get_MIC_hash : MIC_info -> algid
  definition of get_MIC_hash is
    axiom (equal (get_MIC_hash x)
           ((project 1) ((relax is_MIC_info?) x)))
  end-definition
  ...
end-spec
```

Figure 6. Comparison of HOL and SPECWARE specifications for MIC-Info

5. Synthesis of PEM MIC-CLEAR Messages

Having verified that the specifications for (the design of) the data structures and operations satisfy the required integrity property, we turn to the synthesis phase of system development. The previous analysis is legitimate for the final system only if the synthesized code can be related formally to the specifications. To this end, we specify the PEM system in SPECWARE and then refine it to code. The HOL specification serves as a road map for the SPECWARE specification, as the two specifications are very similar. Figure 6 illustrates the syntactic similarity of the HOL and SPECWARE specifications for the MIC-Info structure.

5.1. Theoretical basis of SPECWARE

The implementation phase relies on SPECWARE's support for both the composition of specifications and the refinement of specifications into C++ code. These composition and refinement processes are based on categorical constructions involving categories of algebraic specifications. Roughly speaking, a *specification* comprises a signature (i.e., a collection of sorts (or types) and a collection of operators over those sorts) and a collection of axioms over those sorts [9]. A *specification morphism* between two specifications is a mapping between their signatures that preserves theorems. Intuitively, a specification morphism from *A* to *B* indicates how *A* can be extended to *B* (equivalently, how every model of *B* can be viewed as a model of *A*).

Whenever a specification *A* can be extended to two different specifications *B* and *C*, there is a canonical composite specification that exhibits all the properties of both *B* and *C*. This specification can be obtained as a quotient of the disjoint union of the two specifications, where individual sorts and operators of *B* and *C* are unified exactly when they are the extensions of the same sort or operator in *A*. This construction is based on categorical pushouts (or, more generally, finite colimits).

Pushouts and other finite colimits form the basis for instantiation of parameterized specifications. For example, we can compose a specification `HASH` for hash functions with a specification `SIGN` for signature-generation functions to yield a specification for generating MICs on the messages, as shown in Figure 7. In this diagram, dotted

lines represent element mappings, while solid lines represent specification morphisms. Thus the single sort *E* in ONE_SORT is mapped to both *md* in HASH and *plaintext* in SIGN. As a result, *md* and *plaintext* are identified as the single sort *md*[1] in the resulting specification SIGN_HASH, as evidenced by the types of the operators *hMD2* and *sRSA*.

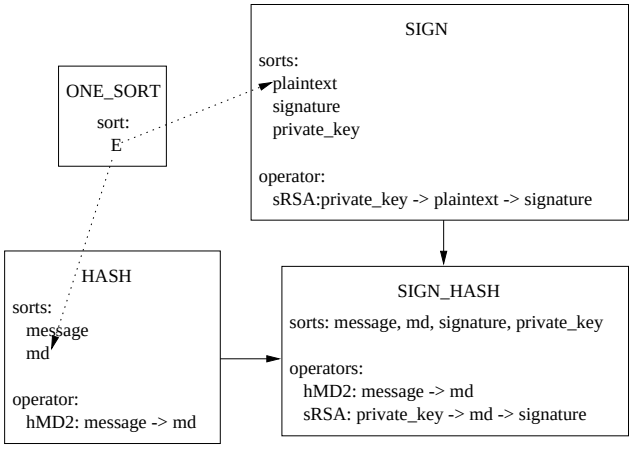

Figure 7. Composition of specifications for hash and for digital signature

Refinement of specifications—the mechanism by which code is synthesized— also occurs via colimits, in a category of specifications and *interpretations*. An interpretation from *A* to *B* can be viewed as a specification morphism from *A* to a definitional extension of *B*, which is a specification that expands *B*'s collection of sorts, operators, and axioms without altering its collection of models.

These interpretations serve as refinements. For example, suppose we have a source specification for *traffic light* that has one sort *color*, and three operators (or constants) *green*, *red* and *yellow*. We can implement *traffic light* using a pair of booleans through a mediating specification color-as-bool-pair. In color-as-bool-pair we introduce a new sort *med-color* whose elements are defined in terms of a subset of (the constructed) sort *bool* $\times$ *bool*. We then map the sort *color* to *med-color* and the operators of sort *color* to operators of sort *med-color*. The interpretation *color-to-bool-pair* is illustrated in Figure 8; in this diagram, dotted lines represent element mappings, and solid line represents isomorphic mapping for introducing new type.

Refinements can themselves be composed, in what are termed *sequential compositions* and *parallel compositions*. Sequential composition can be viewed as transitivity of refinements: a refinement from *A* to *B* can be composed with

[1]The selection of the (overloaded) name *md* for the unified sort is a design decision.

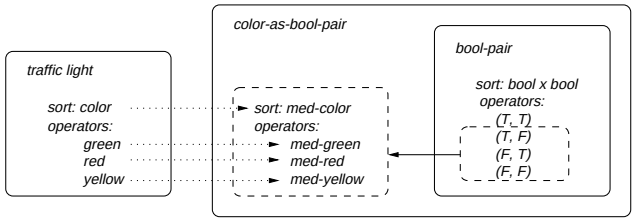

Figure 8. Interpretation *color-to-bool-pair*: implementation of *color* with a boolean pair

a refinement of *B* to *C* to yield a refinement from *A* to *C*. Parallel composition is based on colimits of interpretations. In particular, the refinement of a system obtained by composing several components can be obtained by a parallel composition of the individual components' refinements. As a result, a library of relatively small specifications can be used to generate code for a large system: the small specifications can be composed to create a large specification whose refinement into code is obtained by the composition of the refinements of the small specifications.

5.2. Specification for PEM MIC-CLEAR messages

During the specification process, we build specifications via the composition of basic specifications.

We create a specification SECURE_MAIL to specify a mail system with integrity protection (see Figure 9). This specification does not impose any particular message structure

```
spec SECURE_MAIL is
  sorts Message, Md, Hash, MIC, Pkey, Verify
  sort-axiom Hash = Message -> Md
  sort-axiom Verify = (Pkey, Md, MIC)
                      -> boolean

  op is_Intact : (Verify, Hash, Message,
                  MIC, Pkey) -> boolean
  definition of is_Intact is
    axiom (equal (is_Intact v h msg mic ekey)
                 (v ekey (h msg) mic))
  end-definition
end-spec
```

Figure 9. SPECWARE specification SECURE_MAIL

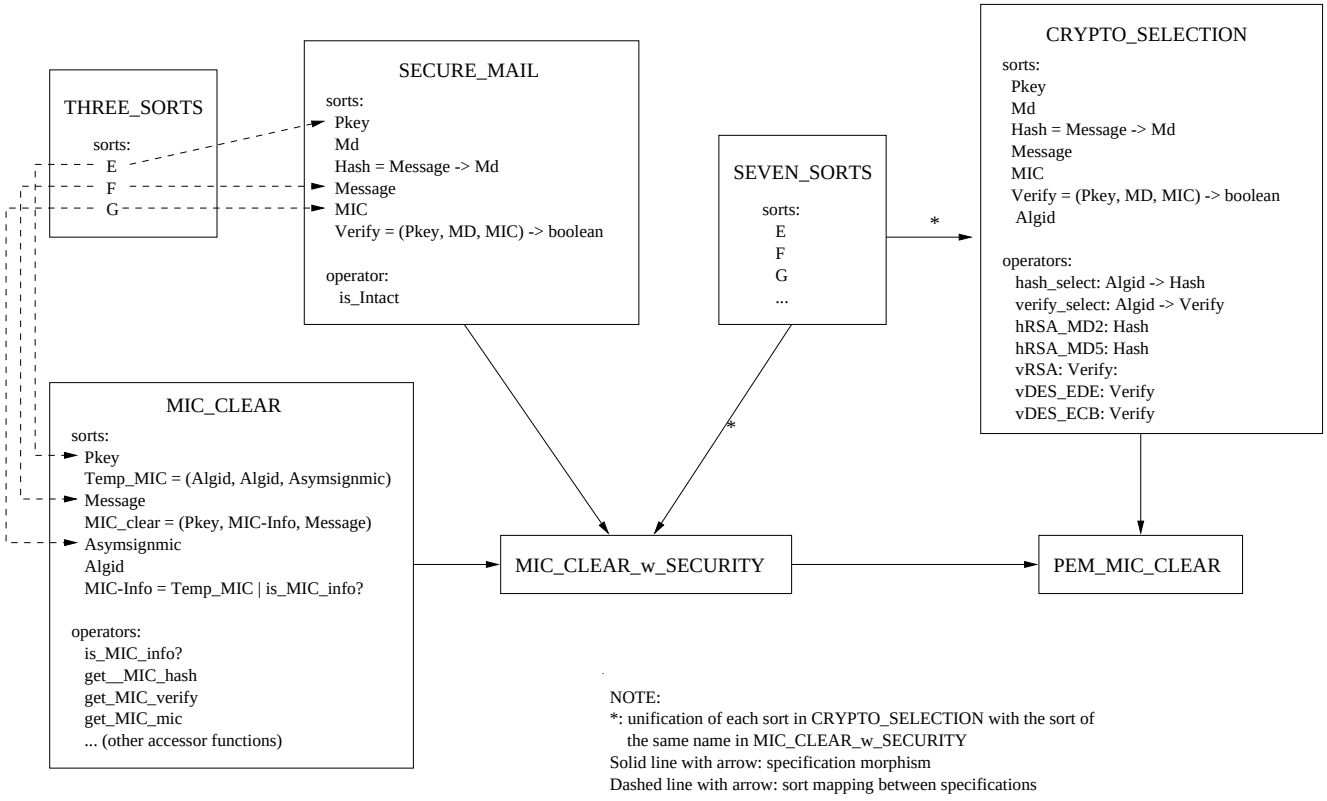

Figure 10. Specification for PEM MIC-CLEAR messages

on the mail; the integrity check *is_Intact* is independent of message structures and protocols. We can reuse this specification for different mail systems with different message structures.

We build a specification for PEM MIC-CLEAR messages by composing SECURE_MAIL with following specifications:

- MIC_CLEAR defines a PEM MIC-CLEAR message structure, together with accessor functions that retrieve the fields from mail messages.

- CRYPTO_SELECTION defines types for hash functions, signature-verification functions, and algorithm IDs, and also defines selector functions that map algorithm IDs to cryptographic functions.

The composition is shown in Figure 10. In this figure, the boxes represent individual specifications, while the solid arrows represent specification morphisms. The dotted arrows from THREE_SORTS to MIC_CLEAR and to SECURE_MAIL indicate the individual sort mappings of two specification morphisms and illustrate how the sorts of MIC_CLEAR and SECURE_MAIL are unified.

The ultimate result of composing these specifications is a specification PEM_MIC_CLEAR for a PEM MIC-CLEAR mail system with an integrity check. Replacing the specification MIC_CLEAR in this composition with a specification for a PEM ENCRYPTED message would yield a specification for a PEM ENCRYPTED system with an integrity check. Likewise, replacing MIC_CLEAR with a specification for a MISSI message structure and replacing CRYPTO_SELECTION with a MISSI specification for cryptographic algorithms would yield a specification for a MISSI implementation with an integrity check.

5.3. Refinement of specifications

To refine the composite specification PEM_MIC_CLEAR, we refine its components and then compose the resulting refinements. When the refinements become sufficiently low level, SPECWARE supports the translation of the lowest-level specifications into C++ code through the use of built-

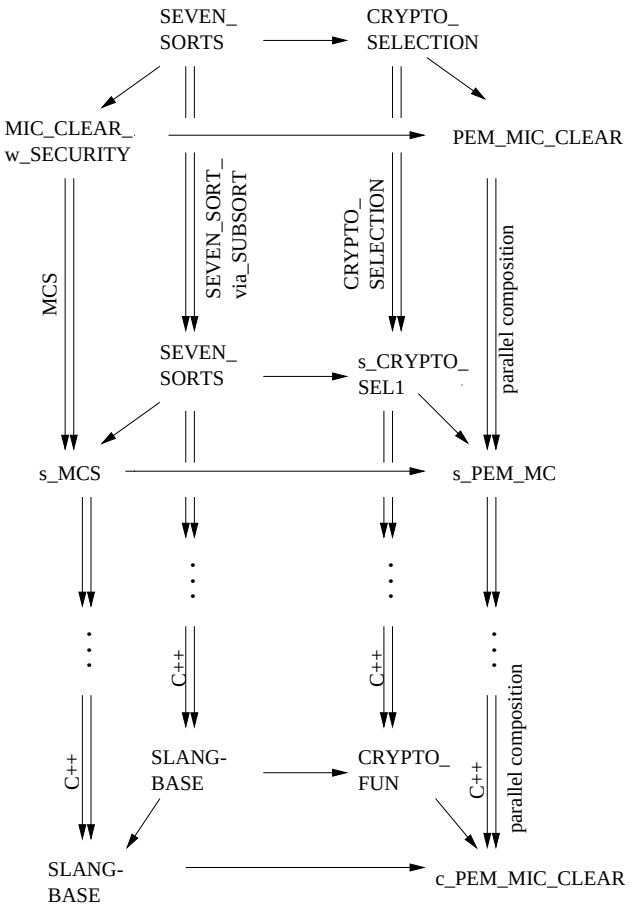

Figure 11. Refinement of composite specification `PEM_MIC_CLEAR`

in theories. Figure 11 sketches the refinement process; the refinements (i.e., interpretations) appear as the vertical double arrows.

A portion of the resulting code appears in Figure 12.

6. Conclusions

The purpose of this work was to demonstrate an integrated verification and synthesis process on an engineering application. Higher-order logic bridges the two systems used for verification and for synthesis; it is a useful intermediate language for relating formal tools.

The automatically generated code was not as concise as custom designed code. Nevertheless, it was assured code that worked.

In constructing this system, we developed an algebraic specification for each component of PEM. The use of abstract data type helps partition the system into modules, which should increase system maintainability. We have benefited from the emphasis on modularity and composition: we were able to rebuild the system easily when components were changed.

The formal specification and verification, together with the use of component-based design, helped us identify a secure core protocol that is common to many secure email systems. Once the details of the mail-message structures of different mail systems have been abstracted away, the underlying core protocol appears the same. We are in the process of formally specifying and implementing this core protocol. We will (re)use the core protocol to specify and synthesize both PEM and PGP formally and to relate these two secure email systems.

References

[1] D. Balenson. Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers. RFC 1423, TIS, February 1993. ftp: ds.internic.net.

[2] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security Private Communication in a Public World*. Prentice Hall, New Jersey, 1995.

[3] John P. Van Tassel D. Randolf Johnson, Fay F. Saydjari. MISSI Security Policy: A Formal Approach. Technical Report R2SPO-TR001-95, INFOSEC Research and Technology Group, National Security Agency, July 1995.

[4] Dan Craigen, Susan Gerhart, and Ted Ralston. An international survey of industrial applications of formal methods; volume 1: Purpose, approach, analysis and conclusions; volume 2: Case studies. Technical Report NIST GCR 93/626, National Institute of Standards and Technology, 1993.

[5] Dan Zhou and Shiu-Kai Chin. Formal Verification of Security Properties of Privacy Enhanced Mail. Technical report, Rome Lab, 1998.

[6] Dan Zhou and Shiu-Kai Chin. Verifying Privacy Enhanced Mail Functions with Higher Order Logic. *Network Threats, DIMACS Series in Discrete Mathematics*, 38:11–20, 1998.

[7] Edmund Clarke and Jeannette Wing. Formal methods: State of the art and future directions. *Report of the ACM Workshop on Strategic Directions in Computing Research, Formal Methods Subgroup*, August 1996. Available as CMU Computer Science Technical Report CMU-CS-96-178.

[8] M.J.C. Gordon. A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subramanyam, editors, *VLSI specification, verification and synthesis*. Kluwer, 1987.

[9] J. A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R.T.Yeh, editor, *Current Trends in Programming Methodology, Volume IV*, pages 80–149. Prentice Hall, 1978.

```
boolean is_intact (verify_1 v, hash_1 h, string msg, string mic, string ekey)
{
  return v (ekey, h (msg), mic);
}

boolean mic_clear_is_intact (mic_clear_2 x)
{
  string _x_1 = project(1,x);
  e _x_2 = project(2,x);
  string _x_3 = project(3,x);

  return
    is_intact
    (verify_select (get_mc_verify ( product(_x_1, _x_2, _x_3) )),
     hash_select (get_mc_hash ( product(_x_1, _x_2, _x_3) )),
     get_mc_message ( product (_x_1, _x_2, _x_3) ),
     get_mc_mic ( product (_x_1, _x_2, _x_3) ),
     get_mc_pkey ( product (_x_1, _x_2, _x_3) ));
}
```

Figure 12. SPECWARE-generated C++ code for MIC-CLEAR integrity checking

[10] B. Kaliski. Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services. RFC 1424, RSA Laboratories, February 1993. ftp: ds.internic.net.

[11] S. Kent. Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. RFC 1422, BBN, February 1993. ftp: ds.internic.net.

[12] J. Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. RFC 1421, DEC, February 1993. ftp: ds.internic.net.

[13] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993. A book based on this material will be published by Cambridge University Press in 1999.

[14] Yellamraju V. Srinivas and Richard Jullig. Specware: Formal Support for Composing Software. In *Conference on Mathematics of Program Construction*, Kloster Irsee, Germany, July 1995.

[15] P.R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, Massachusetts, 1995.